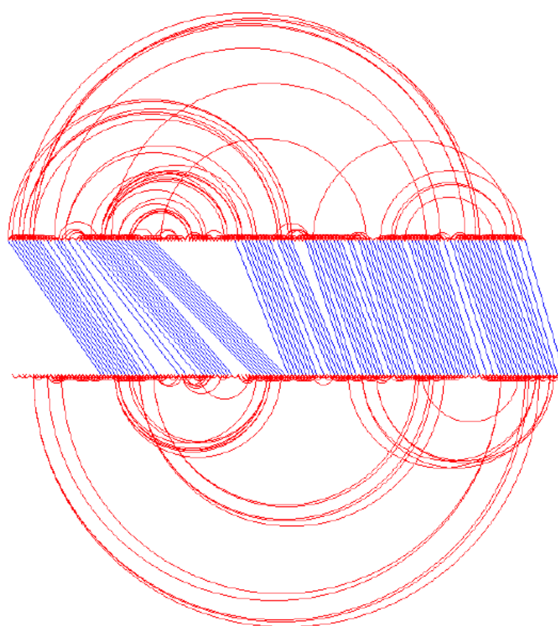
Хабилитационен труд за професор

специалност Изследване на операциите

ПРАКТИЧЕСКА ОПТИМИЗАЦИЯ

Никола Янев



Софийски университет "св. Кл. Охридски"

1 август 2007 г.

УВОД

Практическата оптимизация е изкуство и наука за разпределяне на ограничен ресурс по най-добър начин. Чрез тази често използвана дефиниция, почти безпроблемно могат да се идентифицират като практически различните оптимизационни техники, използвани в индустриалното планиране, разпределение на ресурси, разписания, вземане на решения и т. н. Например, как петролна компания решава откъде да купува петрол, къде да го рафинира, какви продукти да произвежда и на какви цени ? Отговорът е в използване на оптимизационен модел за максимална печалба. От друга страна, оптимизационен модел за минимален разход е удачно да се използва за планиране на полети и разписание за екипажите в практиката на самолетни компании. Понятието ресурс в тези модели може да бъде: пари, работни часове, гориво и пр. Как обаче можем да класифицираме, например, задачите разглеждани в тази работа, които формално са за намиране на оптимално (в определен смисъл) сдвояване в двуделен граф? За какъв ресурс може да става дума в такъв контекст? Една възможност е ребрата на двуделния граф да бъдат съпоставени на върхове в друг граф, път в който съответства на сдвояване. Въпрос на не толкова богата фантазия е задачата за намиране на път да се имитира като задача за разпределение на ресурс. Не толкова лесно е използването на тази дефиниция за класифициране на другите, разгледани тук, задачи, но в тези случаи просто ще приемем, че става въпрос за оптимизационни задачи с реален практически ефект.

По-долу ще се спрем на основните оптимизационни техники за решаване на реални задачи с големи размери. Като начало ще отбележим, че цялата история на практическата оптимизация(тук и по-нататък акцентът е само върху задачи с големи размери, или по-удачния английски термин large-scale optimization) е много кратка. Откривателят на симплекс-метода, Джордж Данциг почина през 2005 г. Самият метод е все още в основата на най-ефективните алгоритми за решаване на задачите на **линейното оптимиране** и в зората на елктронното смятане огромния дял на смятанията с компютри се пада именно на такива задачи. Сега, благодарение на значителните инвестиции в създаване на мощен софтуер (напр. CPLEX на фирмата ILOG) задачи с милиони променливи и стотици хиляди ограничения се решават за приемливо време и в редица случаи (при отсъствие на специализирани алгоритми) позволяват решаването на важни задачи да става още преди важността на самата задача да привлече вниманието на съответни специалисти (математици,информатици, оптимизатори) .

Поучителен пример за това е демонстриран в първа глава, където се разглежда изключително актуалната за биолозите задача за свиване на протеини. (Един популярен *in silico* подход, наречен Protein threading problem (**PTP**) се разглежда в Глава 1.) Благодарение на предложения математически модел на задачата, стана възможно решаването на огромни по размери задачи със не специализиран софтуер (CPLEX) да става за приемливо време и като резултат да се повиши прогностичната възможност на използващата го система. Като страничен (но не маловажен) резултат се

откриха неочаквани връзки между целевата функция и многостена на ограниченията на линейната задача, които стимулираха създаването на алгоритъм за решаване на същата задача със невероятно (за размерите на задачите) бързодействие. Осветяването на тази връзка, обаче, е тясно свързано с оптимизационни техники, принадлежащи на **целочислената оптимизация.**

Докато в линейното оптимиране, променливите имат реални стойности (напр. 3. 7 или 101. 1123) в целочислените оптимизационни задачи те могат да имат само целочислени стойности и в много случаи само 0 или 1. Много от практическите задачи са целочислени по своята природа и въпреки че изглеждат лесно решими ($0/1$ задача с n променливи може да се реши с 2^n пресмятания на целевата функция) в общия случай е невъзможно да се генерира всяка допустима точка и да се избере най-добрата. Много лесно е да се конструират скромни по размер целочислени задачи, чието множество от допустими решения е по-голямо от броя на атомите във вселената. Именно тази "комбинаторна експлозия "е причината за възникване на специфична теория и техники обединени под името целочислено линейно оптимиране (накратко целочислено оптимиране) . Целочислената оптимизация е най-успешния подход за точно решаване на трудни оптимизационни задачи, възникващи в стотици реални ситуации. Този подход се състои в формулиране на проблема като максимизация (минимизация) на линейна функция на целочислени променливи и решаването му чрез метода "ограничаване и граници където границите идват от линейната релаксация на задачата (ЛО релаксация) .

В ЛО релаксацията се оптимизира същата функция, но без ограничение за целочисленост на променливите. Колкото по-добри са границите, толкова по-успешна е формулировката на задачата. Казано иначе, колкото по-близко е стойността на целевата функция върху разширената област до стойността и върху целочислените точки, толкова по-ефективно е редуцирането на допустимото множество. Много често по-добра формулировка се постига чрез добавяне на "отсичания"(линейни неравенства) , което резултатира в подхода известен като "разклонение и отсичания". Отсичанията са ограничения, които не елиминират допустими точки, но намаляват обема на множеството от нецелочислени точки и като резултат подобряват границите, получавани от ЛО решението. Линейното оптимиране е важен инградиент на целочислената оптимизация и за някои задачи (РТР е интересен пример за това) е успешен заместител на специфични други техники за решаване на целочислени оптимизационни задачи. Формално, класът задачи, в които изискването за целочисленост на променливите е излишно, е нетрудно да бъде специфизиран благодарение на основната теорема на линейното оптимиране, която гласи че: решимите линейни задачи достигат оптимума си във връх на многостена на ограниченията. От тук следва, че ако този многостен е целочислен (върховете му са целочислени вектори) , то за произволна линейна целева функция, решението получено с използване на симплексно-подобни алгоритми ще бъде целочислено.

Един богат клас задачи с подобно свойство на ограничителния многостен са задачите за оптимизация в **мрежи**. Мрежите са графи с тегла на върховете и (или) ребрата и като такива са естествени модели на много реални ситуации. Това разбира

се не означава, че всяка задача, чиито математически модел е на мрежова основа, е лесно (полиномиално) решима. Класически пример за това е прочутата задача за търговския пътник и не толкова прочутите задачи разглеждани в този труд, които обаче са поне толкова трудни колкото и първата задача. Затова, в повечето случаи, твърдения от рода "предложен е ефективен алгоритъм за решаване на . . . . "се отнасят до емпиричен анализ на ефективността, чрез сравнение с други алгоритми, решаващи същата задача и не на последно място използваемостта му в конкретна практика.

Друга често използвана оптимизационна техника е **динамичното опимиране**, което обикновено се представя като подход за взимане на решения, свързани върху последователни моменти от времето. Понятието "време"е само исторически обусловено и в многобройните съвременни използвания на тази техника "етапната "оптимизация се замества с попълване на 2-мерна таблица с използване на рекурсивни връзки. Много убедителна демонстрация за ефективността на алгоритми, базирани върху динамичното оптимиране, са непрекъснато решаваните задачи за сравнение на нуклеотидни редици като разбира се не може да се пропусне сензационното секвениране на човешкия геном. Динамичното оптимиране е често използвана техника и в настоящия труд в два основни аспекта.

Първият е във всички алгоритми, базирани върху Лагранжевата релаксация и двойственост. Лагранжевата релаксация е средство за генериране на по-добри граници от ЛО релаксацията и поне за разглежданите тук практически задачи е свързано със създаване и използване на алгоритми на динамичното оптимиране. Същността на Лагранжевия подход е следната: част от ограниченията на задачата се дуализират (прибавят се към целевата функция със тегла, зададени от стойността на двойствени променливи, съответстващи на дуализираните ограничения) . За всяка стойност на двойствените променливи, релаксираната задача се решава върху множеството на оригиналните променливи, което съответства на пресмятане на стойността на Лагранжевата функция (дефинирана върху двойствените променливи. Имено тук е използването на динамичното оптимиране в описаните в Глава 1 алгоритми.) Минимума (максимума) на тази функция е границата, която се търси. Некласическият момент е в пресмятането на стойността на Лагранжевата функция (алгоритъмът зависи от задачата и от дуализираните ограничения) , а класическият е в намирането на оптимума на същата функция и в повечето случаи е т. н. субградиентна оптимизация.

Вторият аспект, свързан с динамичното оптимиране е разгледан в Глава 3, където известната задача за раницата се решава чрез хибридизация на динамичното оптимиране с "разклоняване и граници".

Относно съдържанието на този труд

Глава 1 е посветена изцяло на известни биоинформатични задачи и по-скоро на тяхното решаване. Първата задача е от класа "сравнение от тип редица-структура а втората от класа "сравнение от тип структура-структура". В секция 1 е представен мрежов подход за първата задача, позволяващ създаването на различни линейно целочислени модели. Тези модели дават възможност за прилагане на стратегия "раз-

клоняване и отсичане чиято ефективност е демонстрирана върху широк клас реални задачи. В секция 2 е разгледан нов подход за решаване на същата задача, основан на Лагранжева релаксация от тип "разделяне на разходите (cost-splitting) ". Този алгоритъм има стотици пъти по-голямо бързодействие от най-качествения пакет за решаване на линейни (целочислени линейни) задачи (става въпрос за задачи на линейното оптимиране с милиони променливи и стотици хиляди ограничения) . Това бързодействие, комбинирано с нулевата двойствена дупка (duality gap) в случай на реални задачи, превръща най-тежката операция в системи за предсказване свиването на протеини в рутинна операция. В секция 3 е представен още един точен алгоритъм с използване на двойственост по Лагранж, като цялата проблематика е разгледана върху обща платформа- сдвояване в двуделни графи. Доказани са интересни свойства на многостена на допустимите сдвоявания. В секция 4 се разглежда задача за оптимално сравнение на тримерни структури на протеини, известна като CMO (contact map overlap) . Задачата, която възниква при този подход е еквивалентна (в общия случай) на задача за намиране на максимална по мощност клика в произволен граф, а в реалния случай това е задача за намиране на максимален общ подграф на два графа (описващи структурите на два протеина чрез релацията "близо-далече") . Описан е алгоритъм, с използване на двойственост по Лагранж и е показано неговото тотално доминиране над най-ефективния (публикуван в 2004 г.) известен алгоритъм.

Секция 5 е посветена на задача за сегментиране на бактериален геном (този път не NP-трудна) , свеждаща се до две нови задачи за най-къс път в ацикличен ориентиран граф. В първата задача (и по-лесната) се търси път минимизиращ отклонението на дължините на ребрата от зададена "идеална"дължина. Във втората задача последната дължина е неизвестна. В приложение А се разглеждат проблемите, възникващи при създаване на системи за автоматично предсказване свиването на протеини. Разгледана е реално действаща система FROST, в която са интегрирани споменатите по-горе алгоритми. Описан е и алгоритъм за нормализиране на целевата функция (виж описанието на score function) и неговата реализация върху многопроцесорна система (става въпрос за решаване на милион NP-трудни задчи с използване на т. н. grid computing) .

Разглежданията в Глава 2 са върху незатихващата тематика, свързана с разпаралелване на перфектни вмъкнати цикли, пресмятащи регулярни изрази, срещащи се в почти всяка компютърна програма за научни или инженерни смятания. Използваната техника известна като "павиране"(tiling) е основана на покриване на индексното пространство на индексите на цикъла с "плочки"(tiles) , позволяващи на процесора, работещ с плочката да извършва пресмятането на израза, при положение, че са му известни стойностите по границата (границите) . При зададен модел на паралелната архитектура (в повечето изследвания това са архитектури с разпределена памет) се търси формата (хипер правоъгълник или паралелепипед) и неговия обем, а също и разпределението на плочките по процесорите, така че сумарното време за изпълнение на цикъла да е минимално. Основна особеност на възникващите оптимизационни модели е тяхната нелинейност и целочисленост, а основният проблем при тяхното решаване е необходимостта от намиране на оптималното решение в затворена фор-

ма (аналитично) . В първата секция на тази глава, формата е фиксирана на хипер правоъгълник, а дълбочината на цикъла е произволна. Във втората секция, формата е паралелепипед (oblique tiling) , като това разбира се е определено от векторите на зависимост, участващи в регулярния израз. И в двете секции са получени точни решения на съответните оптимизационни задачи, като в случая на паралелепипед е получен и страничен резултат върху тематиката на първа глава. За една от най-често решаваните задачи за оптимално сравнение на редици е получено разпаралелване на съответен алгоритъм (динамично оптимиране) с доказани оптимални свойства.

Глава 3 е посветена на два класа класически оптимизационни задачи: многомерна 0/1 задача за раницата и задача за раницата. И двете задачи са NP-трудни, но докато за първата, примери с 100 променливи и 5 ограничения могат да затруднят произволен съществуващ алгоритъм, то за втората намирането на трудни примери вече е проблем. За многомерната задача е описана процедура, позволяваща фиксиране на променливите в началната фаза на алгоритъма. Процедурата е основана на построяване на намаляваща редица от горни граници и растяща редица от долни граници (задачата е за максимум) . Сравняването на двете редици позволява или доказване на оптималността на най-доброто получено допустимо решение или фиксиране на променливи. Процедурата е и основа на евристика, качеството, на която е демонстрирано върху известни тестови примери с неизвестно оптимално решение. "По-лесната"задача за раницата е разгледана във втората секция на тази глава, където акцентите са върху: представяне на хибриден алгоритъм за тази задача, който е по-добра версия на най-бързия в момента алгоритъм, основан на техниката на динамичното оптимиране. Получена е интересна нова горна граница на оптималното решение и е въведен нов клас задачи за раницата, разширяващ известния клас (трудни) на силно корелираните задачи. Преимуществото на новият алгоритъм е демонстрирано върху най-различни класове от примери, като едновременно са демонстрирани нови тестови примери на все още трудни задачи.

В последната глава е разгледан проблема за намиране на линеен класификатор (сепаратор) на две групи от точки в n-мерното пространство. Критерият за намиране на оптималната разделяща хиперравнина е минимума на броя на грешно класифицираните точки. Представен е подходящ математически модел на оптимизационната задача с използване на класическата теорема на Фаркаш. На базата на този модел е създаден алгоритъм с добри възможности за реално използване. Развитата теория по-късно е използвана в редица мета-евристики за атакуване на същата задача, но те не са дискутирани в тази глава.

В този хабилитационен труд са разгледани резултати, получени след 1998 г. , която (поне за нас) бележи началото на активното използване на интернет и за бързо разпространение на научни резултати. Все по-далече остава времето, когато се губеше много енергия за запознаване със състоянието на нещата в дадена област, особено ако си новодошъл. Сега, с популярни търсачки, това става бързо и лесно. Обичайната практика в момента е докато чакаш ред в някое списание да качиш т. н. technical report в интернет (в много научни институции това се прави автоматично) и сравни-

телно спокойно да чакаш овъзмездяването на своя труд с появата му в списание (за предпочитане с висок импакт фактор. Това понякога може да създаде проблеми от друго естество, какъвто е случаят с резултатите в Глава 3. За по-малко от половин година от качването на съответните препринти в интернет, с изненада ги срещнахме в новопубликувана монография на известни специалисти по задачи за раницата (със съответните цитирания разбира се) . Това приятно на пръв поглед признание, по-късно създаваше проблеми с публикуването на същите резултати в специализирани списания.

Преди да пожелаем приятно четене трябва да отбележим следното: Невключването в този труд на резултатите получени в прединтернетската епоха, в никой случай не означава омаловажаване. Много от тях се отнасят до задачи, чиято актуалност ще продължава докато $P \neq NP$? Попитайте, например, Google за lot-size problem, bin-packing problem. Една от причините за тяхното оставяне "извън борда"е отекчението от четене на дебели книги, за което допринася и този дълъг предговор.

PRACTICAL OPTIMIZATION

Nicola Yanev

August 1, 2007

Contents

1 Models and Algorithms for Computational Biology Problems **7**
1.1 High Performance Alignment Methods for Protein Threading 9
1.1.1 Introduction 9
1.1.2 Formal definition 13
1.1.3 Mixed integer programming models 16
1.1.4 A model using vertices, x-arcs, and z-arcs 22
1.1.5 The **RAPTOR** Model 24
1.1.6 Experimental comparison of the MIP models 24
1.1.7 Divide and conquer 26
1.1.8 Parallelization 30
1.1.9 Future research directions 35
1.1.10 Conclusion 36
1.2 Optimal protein threading by cost-splitting 37
1.2.1 Motivation 37
1.2.2 Protein threading problem revisited 38
1.2.3 Special cases 39
1.2.4 All edges have their left end tied to a common vertex 40
1.2.5 Relaxation through decomposition 42
1.2.6 Experimental results 44
1.2.7 Conclusion 47
1.3 Lagrangian approaches for a class of matching problems in computational biology 48
1.3.1 Preliminaries 48
1.3.2 Integer programming formulation 52
1.3.3 Complexity results 53
1.3.4 Lagrangian approaches 57
1.3.5 Lagrangian relaxation 58
1.3.6 Cost splitting 59
1.3.7 Experimental results 60
1.3.8 Conclusion 64
1.4 A Novel Algorithm for Finding Maximum Common Ordered Subgraph 66
1.4.1 Introduction 66
1.4.2 The mathematical model 68

1.4.3 Lagrangian relaxation approach . . . 70
1.4.4 Computational results . . . 73
1.4.5 Conclusion . . . 79
1.5 Optimal Segmentation of Bacterium Genomes . . . 80
1.5.1 Introduction . . . 80
1.5.2 Graph problem formulation . . . 82
1.5.3 The case when the segment length is given . . . 83
1.5.4 The case when the segment length is unknown . . . 84
1.5.5 Computational experiments . . . 88
1.5.6 Conclusion . . . 89

Bibliography **90**

2 Tiling problems **97**
2.1 Optimal Orthogonal Tiling . . . 98
2.1.1 Introduction . . . 98
2.1.2 Abstract Model Building . . . 99
2.1.3 Machine and Program Specific Model . . . 101
2.1.4 Solution for the simple models . . . 103
2.1.5 Solution for the BSP cost function . . . 105
2.1.6 Conclusions . . . 108
2.2 Optimal Semi-Oblique Tiling . . . 109
2.2.1 Introduction . . . 109
2.2.2 Problem Formulation . . . 111
2.2.3 Optimizing the tile size . . . 116
2.2.4 Optimizing the number of processors . . . 122
2.2.5 Optimizing Tile Shape . . . 124
2.2.6 The sequence global alignment application: Fickett's algorithm . . . 125
2.2.7 Experimental Validation . . . 127
2.2.8 Related Work . . . 130
2.2.9 Conclusions . . . 133

Bibliography **134**

3 Knapsack Problems **139**
3.1 A Dynamic Programming Based Reduction Procedure for the Multidimensional 0-1 Knapsack Problem . . . 139
3.1.1 Introduction . . . 139
3.1.2 Related work . . . 140
3.1.3 Heuristic methods . . . 141
3.1.4 Preproccesing techniques . . . 141
3.1.5 Dynamic programming approaches . . . 142
3.1.6 General method . . . 142

3.1.7 List representation . . . 143
3.1.8 A new procedure to fix variables . . . 144
3.1.9 Computing the bounds . . . 146
3.1.10 Experimental results . . . 150
3.1.11 Conclusion . . . 154
3.2 A Hybrid Algorithm for the Unbounded Knapsack Problem . . . 155
3.2.1 Introduction . . . 155
3.2.2 A summary of known dominance relations and bounds . . . 156
3.2.3 A new general upper bound for UKP . . . 157
3.2.4 The new algorithm `EDUK2` . . . 159
3.2.5 Performance evaluation experiments . . . 161
3.2.6 Conclusion . . . 173

Bibliography **173**

4 The classification problem **181**
4.1 Overview of approaches to the classification problem . . . 182
4.2 Problem formulation . . . 184
4.3 Exact algorithm . . . 186
4.3.1 Branching procedure . . . 186
4.3.2 Lower bounds . . . 187
4.3.3 Algorithm Description . . . 188
4.4 Upper bounds and heuristic algorithms . . . 189
4.5 Computational experiment . . . 193
4.6 Conclusions . . . 196

Bibliography **197**

Chapter 1

Models and Algorithms for Computational Biology Problems

The comparison of protein structures is a problem of paramount importance in structural genomics, with applications to drug design, fold prediction, protein clustering, and evolutionary studies. An increasing number of approaches for its solution were proposed over the past years, but despite the extraordinary international research effort, progress is slow. A fundamental dimension of this bottleneck is the absence of rigorous algorithmic methods.The situation is due to several reasons, which include the following.

1. By their nature, three-dimensional computational problems are inherently more complex than the similar one-dimensional ones for which we have effective solutions. The mathematics that can provide rigorous support in understanding models for structure prediction and analysis is almost nonexistent, as the problems are blend of continuous, geometric and combinatorial, discrete mathematics.

2. Various simplified versions of the problems were shown NP-hard.

3. There is a dramatic difference between sequence alignment and structure alignment. As opposed to the protein sequence alignment, where we are certain that there is unique alignment to a common ancestor sequence, in structure comparison the notion of a common ancestor does not exist. Similarity in folding structure is due to a different balance in folding forces, and there is not necessarily a one-to-one correspondence between positions in both proteins.

In this chapter, we describe our work on two major problems, namely the protein folding prediction, based on a sequence-to-structure alignment and the structure-to-structure comparison, based on Contact Map Overlap scoring scheme. The remainder of the chapter is organized as follows. In Section 1, we present a new network flow formulation of the problem of predicting 3D protein structures using threading. Several integer programming models based on this formulation are proposed and compared. These models allow for an efficient decomposition and for the application of a parallel branch-and-cut algorithm, significantly reducing the running time. The

efficiency of the approach has been confirmed by extensive computational experiments.In Section 2, we use integer programming approach for solving a hard combinatorial optimization problem, namely protein threading problem. For this sequence-to-structure alignment problem we apply cost-splitting technique to derive a new Lagrangian dual formulation. The optimal solution of the dual is sought by an algorithm of a polynomial complexity. For most of the instances the dual solution provides an optimal or near-optimal (with negligible duality gap) alignment. The speed-up with respect to the broadly advertised approach for solving the same problem in [37] is from 100 to 250 on computationally interesting instances. Such a performance turns computing score distributions, the heaviest task when solving PTP, into a routine operation. In Section 3, we present efficient algorithms for solving the PTP, by considering the problem as a special case of graph matching problem. We give formal graph and integer programming models of the problem. After studying the properties of these models, two kinds of Lagrangian relaxation for solving them are proposed. We present experimental results on real life instances showing the efficiency of our approaches. In Section 4, we present an algorithm for exact solving the contact map overlap problem. This problem has been found to be very useful for measuring protein similarity,i.e. for structure-to-structure alignment. Section 5 is devoted to the studying of Bacterium genome plasticity by Long-Range PCR: genomes of different strains are split into hundreds of short segments which, after LR-PCR amplification, are used to sketch profiles. The segments have: (1) to cover the entire genome, (2) to overlap each other, and (3) to be of nearly identical size. Here, we address the problem of finding a list of segments satisfying these constraints "as much as possible". Two algorithms based on dynamic programming approach are presented. They differ on the optimization criteria for measuring the quality of the covering. The first one considers the maximal deviation of the segment lengths relatively to an *ideal* length. The second one automatically *finds* a segment length which minimizes the maximal deviation.

1.1 High Performance Alignment Methods for Protein Threading

> Recombinant DNA techniques have provided tools for the rapid determination of DNA sequences and, by inference, the amino acid sequences of proteins from structural genes. The number of such sequences is now increasing almost exponentially, but by themselves these sequences tell little more about the biology of the system than a New York City telephone directory tells about the function and marvels of that city.
>
> —C. Branden and J. Tooze, [7]

1.1.1 Introduction

Genome sequencing projects generate an ever increasing number of protein sequences. For example, the Human Genome Project has identified over 30,000 genes which may encode about 100,000 proteins. One of the first tasks when annotating a new genome, is to assign functions to the proteins produced by the genes. To fully understand the biological functions of proteins, the knowledge of their structure is essential.

Unlike other biological macromolecules (e.g., DNA), proteins have complex, irregular structures. They are built up by amino acids that are linked by peptide bonds to form a . The amino acid sequence of a protein's polypeptide chain is called its primary or one-dimensional (1D) structure. Different elements of the sequence form local regular secondary (2D) structures, such as α-helices or β-strands. The tertiary (3D) structure is formed by packing such structural elements into one or several compact globular units called domains. The final protein may contain several polypeptide chains arranged in a quaternary structure. By formation of such tertiary and quaternary structure, amino acids far apart in the sequence are brought close together to form functional regions (active sites). The interested reader can find more on protein structure in [7].

Protein structures can be solved by experimental (*in vitro*) methods, such as or nuclear magnetic resonance (NMR) spectroscopy. Despite the recent advances in these techniques, they are still expensive and slow, and cannot cope with the explosion of sequences becoming available. That is why molecular biology resorts to computational (*in silico*) methods of structure determining.

The protein folding problem can be simply stated in the following way. Given a protein 1D sequence, which is a string over the 20-letter amino acid alphabet, determine the coordinates of each amino acid in the protein's 3D folded shape. Surprisingly, in many cases the input information is completely sufficient to produce the desired output.

Although simply stated, the protein folding problem is quite difficult. The natural folding mechanism is complicated and poorly understood. Most likely, it is a global result of many local, weak interactions. The protein folding problem is widely recognized as one of the most important challenges in computational biology today [7, 13, 14, 17, 20, 32]. Sometimes it is referred as the "holly grail of molecular biology." The progress of molecular biology depends on the existence of reliable and fast computational structure prediction methods.

The computational methods of structure prediction fall roughly into two categories. The

direct methods [8] use the principles of quantum mechanics. They seek a fold conformation minimizing the free energy. The main obstacles to these methods are the large number of interacting atoms, as well as many technical complications related to cumulative approximation errors, modeling surrounding water, etc.

An important alternative to the direct techniques are the methods using information about proteins of already known structure, stored in databases. These methods are based on the concept of homology, which plays a central role in biology. Two proteins are homologous if they are related by descent from a common ancestor. Homologous proteins have similar 3D structures and often, similar functions.

The easiest way to detect homology between two proteins is to compare their amino acid sequences. If the sequences are "sufficiently" similar then the two proteins are homologous. A number of sequence comparison methods (e.g., BLAST [2], FASTA [26], PSI-BLAST [3]) are available and can be used to detect homology. But if two amino acid sequences are not sufficiently similar, can we conclude that the corresponding proteins are not homologous? In the case of remote homologs, the amino acid sequences have had a plenty of time to diverge. They are no longer similar and lie beyond the sequence comparison recognition threshold, in the so-called twilight zone. Nevertheless, their 3D structures are still similar. In such a case, one of the most promising approaches to the protein folding problem is protein threading. This method relies on three basic facts:

- The 3D structures of homologous proteins are much better conserved than their 1D amino acid sequences. Indeed, many cases of proteins with similar folds are known, though having less than 15% sequence identity.
- There is a limited, relatively small number of protein structural families (between 1,000 and 10,000 according to different estimations [9, 25]). Each structural family defines an equivalence class and the problem reduces to classifying the query sequence into one of these classes. According to the statistics of Protein Data Bank (PDB) [6], 90% of the new proteins submitted in the last three years belong to already known structural families[1].
- Different types of amino acids have different preferences for occupying a given (for example, being in α-helix or β-sheet, being buried in the protein interior of exposed on the surface). In addition, there are different preferences for , or more generally, for spatial proximity, as a function of those environments. These preferences have been estimated statistically and used to produce score functions distinguishing between native and non-native folds.

The process of aligning a sequence to a structure, thereby guiding the spatial placement of sequence amino acids, is known as threading. The term "threading" is used to specialize the more general term "alignment" of a and a structure template.

The fold recognition methods based on threading are complex and time consuming computational techniques consisting of the following main components:

[1] http://www.rcsb.org/pdb/holdings.html

1. a database of potential core folds or structural templates;
2. an objective function () which evaluates any alignment of a sequence to a structure template;
3. a method of finding the best (with respect to the score function) possible alignment of a sequence and a structure template;
4. a method to select the most appropriate among the best alignments of a query sequence and each template from the database.

Components 1, 2 and 4 use mainly statistical methods incorporating the biological and physical knowledge on the problem. These methods are beyond the scope of the present chapter. Component 3 is the most time consuming part of the threading methods. It is the most challenging and the most interesting one from computer scientist's point of view. The problem of finding the optimal sequence-to-structure alignment is referred as protein threading problem (PTP) throughout this chapter. PTP is solved many times in the threading process. The query sequence is threaded to all (or at least to a part of) templates in the database. Component 4 of some methods uses a score normalization procedure which involves threading a large set of queries against each template [22]. The designers of score functions make experiments involving milions of threadings in order to tune their parameters. That is why a really efficient threading algorithms are needed.

As we will see in the next sections, PTP is a hard combinatorial optimization problem. Till recently, it was the main obstacle to the development of efficient and reliable fold recognition methods. In the general case, when variable-length alignment gaps are allowed and pairwise amino acid interactions are considered in the score function, PTP is NP-hard [16]. Moreover, it is MAX-SNP-hard [1], which means that there is no arbitrary close polynomial approximation algorithm, unless P = NP. These complexity results have guided the research of threading methods to three different directions.

The first approach consists of simplifying the problem by ignoring the pairwise amino acid interactions. In this case the optimal alignment can be found by polynomial dynamic programming algorithms [30, 31]. The methods based on this approach ignore potentially rich source of structural information and, consequently, cannot recognize distant structural homologies.

The second direction is to use approximate algorithms which are relatively fast and capable of finding a good but not necessarily the optimal alignment. These methods include several algorithms based on dynamic programming [11, 24, 34], statistical sampling [19], and genetic algorithms [15, 29, 33]. Since these methods do not guarantee optimality, their use risks to worsen the fold recognition sensitivity and quality.

The third way to attack PTP is to develop dedicated exact methods which are exponential in the worst case, but efficient on most of the real-life instances. This chapter traces the later direction of research and presents recently developed high performance exact algorithms for solving PTP, which use advanced mathematical programming techniques, as well as parallel and distributed computing methods. Below we summarize what we feel to be the most important steps in this direction.

Lathrop and Smith [18] designed the first practical branch-and-bound (B&B) algorithm for PTP. This algorithm became the kernel of several structure prediction software packages [21, 22]. Lathrop and Smith's work has shown that the problem is easier in practice than in theory and that it is possible to solve real-life (biological) instances in a reasonable amount of time. However, practical applications of this B&B algorithm remained limited to moderate-size instances. These results have drawn the attention of many researchers to the problem, the authors included.

The second main step involves using mathematical programming techniques to solve PTP. Two teams have been working independently in this direction and their results were published almost simultaneously. Andonov et al. [41, 42, 4] proposed different mixed integer programming (MIP) models for PTP. The content of this chapter is based essentially on the results from [4]. Xu et al. [36, 37] also reported successful use of a MIP model in their protein threading package RAPTOR. The main drawback of mathematical programming approaches is that the corresponding models are often very large (over 10^6 variables). Even the most advanced MIP solvers cannot solve instances of such size in reasonable time and, in some cases, even to stock them in computer memory. Different divide-and-conquer methods and parallel algorithms are used to overcome this drawback.

Xu et al. [39, 38] were the first to use divide-and-conquer in their package PROSPECT-I. Their algorithm performs well on simple template interaction topologies, but is inefficient for protein templates with dense pairwise amino acid interactions. The ideas from PROSPECT-I were used in the latest version of RAPTOR [35]. Andonov et al. [42, 4] proposed a different divide-and-conquer strategy. While the splits in [39] occur along the interactions between template blocks, the splits in [42, 4] are along the possible positions of a given template block. In addition, in the latter works, the solutions of already solved subproblems are used as "cuts" for the following subproblems. Andonov et al. [42, 4] also proposed an efficient parallel algorithm which solves simultaneously the subproblems generated by their divide-and-conquer technique.

The rest of this chapter is organized as follows. In Section 1.1.2 we give a formal definition of PTP and introduce some existing terminology. Section 1.1.3 introduces a network formulation of PTP. Based on this formulation, we present the existing MIP models for PTP in a unified framework and compare them. We show that choosing an appropriate MIP model can lead to considerable decrease in solution time for PTP. We demonstrate that PTP is easier in practice than in theory because the linear programming (LP) relaxation of the MIP models provides the optimal solution for most real-life instances of PTP. Section 1.1.7 presents and analyzes two divide-and-conquer strategies for solving the MIP models. We show that these strategies are a way to overcome the main drawback of the MIP models, their huge size, and lead to significant reduction of the solution time. These strategies are used in Section 1.1.8 to design an efficient parallel algorithm for PTP. The performance of this algorithm is experimentally evaluated and the question of choosing a good granularity (number of subproblems) for a given number of processors is discussed. In Section 1.1.9 we discuss some open questions and future research directions and in Section 1.1.10 we conclude.

1.1.2 Formal definition

In this section we give a more formal definition of PTP and simultaneously introduce some existing terminology. Our definition is very close to the one given in [1, 17]. It follows a few basic assumptions widely adopted by the protein threading community [4, 17, 18, 32, 37, 39]. Consequently, the algorithms presented in the next sections can be easily plugged in most of the existing fold recognition methods based on threading.

Query sequence A query sequence is a string of length N over the 20-letter amino acid alphabet. This is the amino acid sequence of a protein of unknown structure which must be aligned to structure templates from the database.

Structure template All current threading methods replace the 3D coordinates of the known structure by an abstract template description in terms of core blocks or segments, neighbor relationships, distances, environments, etc. This avoids the computational cost of atomic-level mechanics in favor of more abstract, discrete alignment between sequence and structure.

We consider that a structure template is an ordered set of m segments or blocks. Segment i has a fixed length of l_i amino acids. Adjacent segments are connected by variable length regions, called loops (see Fig. 1.1(a)).

Segments usually correspond to the most conserved parts of secondary structure elements (α-helices and β-strands). They trace the backbone of the conserved fold. Loops are not considered as part of the conserved fold and consequently, the pairwise interactions between amino acids belonging to loops are ignored. It is generally believed that the contribution of such interactions is relatively insignificant. The pairwise interactions between amino acids belonging to segments are represented by the so-called contact map graph (see Fig. 1.1(b)). It is common to assume that two amino acids interact if the distance between their C_β atoms is within p Å and they are at least q positions apart along the template sequence (for example $p = 7$ and $q = 4$ in [37]). However, arbitrary contact map graphs can be considered. We say that there is an interaction between two segments, i and j, if there is at least one pairwise interaction between amino acid belonging to i and amino acid belonging to j. Let $L \subseteq \{(i, j) \mid 1 \leq i < j \leq m\}$ be the set of segment interactions. The graph with vertices $\{1, \ldots, m\}$ and edges L is called generalized contact map graph (see Fig. 1.1(c)).

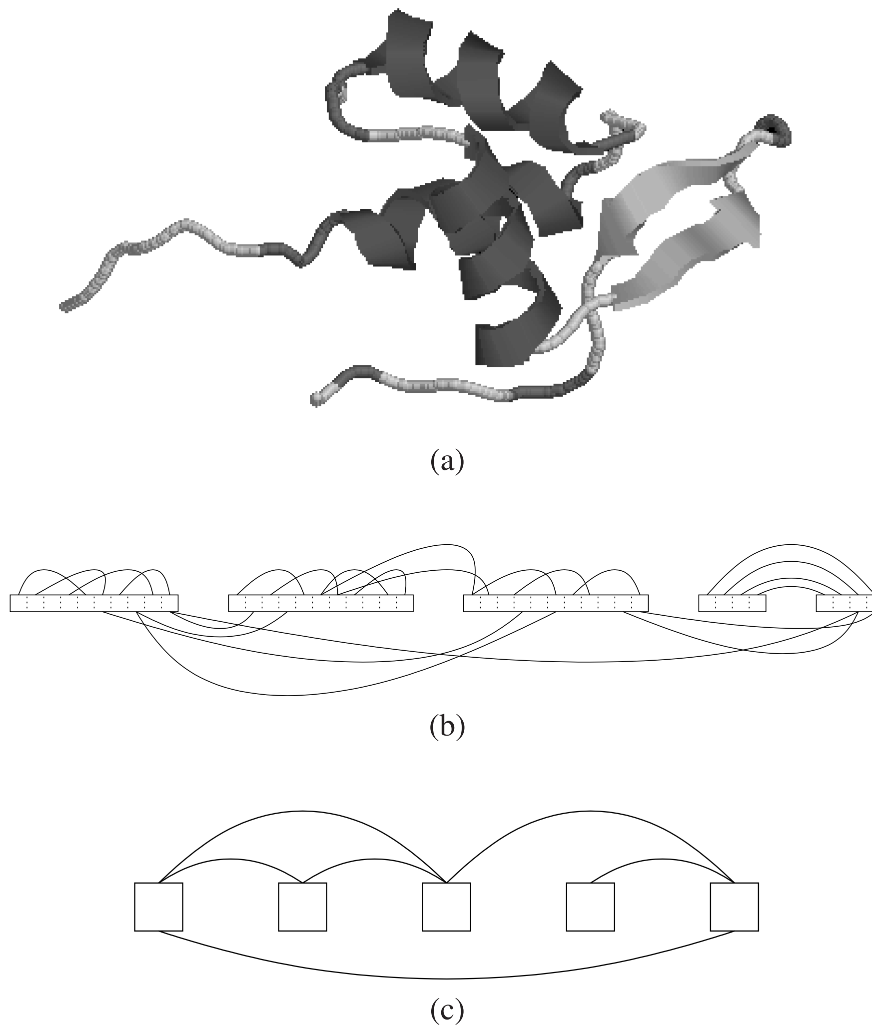
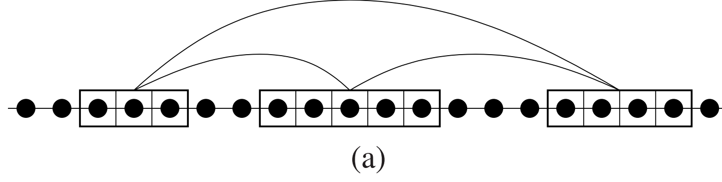


Figure 1.1: (a) 3D structure backbone showing α-helices, β-strands and loops. (b) The corresponding contact map graph. (c) The corresponding generalized contact map graph.

Threading An alignment or threading of a query sequence and a structure template is covering of contiguous query areas by the template segments. A threading is called feasible if the segments preserve their original order and do not overlap (see Fig 1.2(a)).



(a)

abs. position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
rel. position block 1	1	2	3	4	5	6	7	8	9											
rel. position block 2				1	2	3	4	5	6	7	8	9								
rel. position block 3									1	2	3	4	5	6	7	8	9			

(b)

Figure 1.2: (a) Example of alignment of query sequence of length 20 and template containing 3 segments of lengths 3, 5 and 4. (b) Correspondence between absolute and relative block positions.

A threading is completely determined by the starting positions of all segments. For the sake of simplicity we will use relative positions. If segment i starts at the kth query character, its relative position is $r_i = k - \sum_{j=1}^{i-1} l_j$. In this way the possible (relative) positions of each segment are between 1 and $n = N + 1 - \sum_{i=1}^{m} l_i$ (see Fig. 1.2(b)). The set of feasible threadings is

$$\mathcal{T} = \{(r_1, \ldots, r_m) \mid 1 \leq r_1 \leq \cdots \leq r_m \leq n\}.$$

It is easy to see that the number of possible threadings (the search space size of PTP) is $|\mathcal{T}| = \binom{m+n-1}{m}$, which is a huge number even for small instances (for example, if $m = 20$ and $n = 100$ then $|\mathcal{T}| \approx 2.5 \times 10^{22}$).

Most of the threading methods impose an additional feasibility condition, upper and lower bounds on the lengths of the uncovered query areas (loops). This condition can be easily incorporated by a slight modification in the definition of relative segment position.

In the above definition, alignment gaps are not allowed within segments. They are confined only to loops. The biological justification is that segments are conserved so that the chance of insertion or deletion within them is very small.

Score function The score function is used to evaluate the degree of compatibility between the sequence amino acids and their positions in the template in a given threading. This evaluation is based on statistically estimated amino acid preferences for occupying different environments. The choice of an adequate score function is essential for the quality of the threading method. The form of the score function varies from method to method. Here we give a general definition, compatible to most of the threading methods. We only assume that the score function is additive and can be computed considering interactions between at most two amino acids at a time. These assumptions allow to represent the score function by two groups of coefficients:

- c_{ik}, $i = 1, \ldots, m$, $k = 1, \ldots, n$, the score for placing segment i on position k;

- c_{ikjl}, $(i,j) \in L$, $1 \le k \le l \le n$, the score induced by the pairwise interaction between segments i and j when segment i is on position k and segment j is on position l.

The coefficients c_{ik} are some function (usually sum) of the preferences of each query amino acid placed in segment i for occupying its assigned position, as well as the scores of pairwise interactions between amino acids belonging to segment i. The coefficients c_{ikjl} include the scores of interactions between pairs of amino acids belonging to segments i and j. Loops may also have sequence specific scores, included in the coefficients $c_{i,k,i+1,l}$.

Alternatively, we can represent the score function only by the second set of coefficients. To do this, it is sufficient to add c_{ik} to all coefficients $c_{i,k,i+1,l}$, $l = k, \ldots, n$. In the next sections we will use one of these representations, depending on which one is more convenient.

Protein threading problem Using the above definitions, PTP is simply to find the feasible threading of minimum score, or formally,

$$\min \left\{ \sum_{i=1}^{m} c_{ir_i} + \sum_{(i,j) \in L} c_{ir_i jr_j} \mid (r_1, \ldots, r_m) \in \mathcal{T} \right\}.$$

1.1.3 Mixed integer programming models

In this section we restate PTP as a network optimization problem. Based on this reformulation, we present different mixed integer programming programming models for PTP in a unified framework. At the end of the section we compare the efficiency of these models by experimental results. Below we make a brief introduction to mixed integer programming and linear programming, necessary to understand the rest of this section. The reader familiar with these topics can skip directly to Section 1.1.3. For a more detailed and consistent introduction, the reader is referred to any good integer programming textbook, for example [23].

Mixed integer programming (MIP) deals with problems of optimizing (maximizing or minimizing) a linear function of many variables subject to linear equality and inequality constraints and integrality restrictions on some or all of the variables. Because of the robustness of the general MIP model, a remarkably reach variety of optimization problems can be represented by mixed integer programs. The general form of MIP is

$$z_{\text{MIP}} = \min \left\{ cx + dy \mid Ax + By \le b,\ x \in Z_+^n,\ y \in R_+^p \right\},$$

where Z_+^n is the set of nonnegative integral n-dimensional vectors, R_+^p is the set of nonnegative real p-dimensional vectors, $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_p)$ are variables, c is an n-vector, d is a p-vector, A is an $m \times n$ matrix, B is an $m \times p$ matrix, and b is an m-vector. The function $z = cx + dy$ is called objective function and the set $\{(x,y) \mid Ax + By \le b,\ x \in Z_+^n,\ y \in R_+^p\}$ is called feasible region. In many models, the integer variables are constrained to equal 0 or 1. Thus we obtain 0-1 MIP, where $x \in Z_+^n$ is replaced by $x \in B^n$, where B^n is the set of n-dimensional binary vectors.

Although the general MIP problem is NP-hard, it can be solved efficiently in many particular cases. Even in the general case, there exist different efficient solution techniques. Most of them use the linear programming (LP) relaxation of MIP

$$z_{\mathrm{LP}} = \min \left\{ cx + dy \mid Ax + By \le b,\ x \in R^n_+,\ y \in R^p_+ \right\},$$

where the integrality constraints are relaxed. LP is much easier than MIP. It can be solved in polynomial time. The most commonly used method to solve LP is the simplex method which, although exponential in the worst case, performs well in practice.

The most important relations between MIP and LP are: (a) $z_{\mathrm{LP}} \le z_{\mathrm{MIP}}$, i.e. the optimal objective value of LP is a lower bound on the optimal objective value of MIP, and (b) if (x^*, y^*) is an optimal solution of LP and $x^* \in Z^n_+$ then (x^*, y^*) is an optimal solution of MIP. The branch-and-bound algorithms for MIP are based on these relations. They partition MIP into subproblems by fixing some of the x-variables to integer values until either (1) the optimal solution of the LP relaxation of a subproblem becomes feasible for MIP, or (2) the optimal objective value of the LP relaxation of a subproblem becomes greater than the objective value of the best known solution of MIP.

Most optimization problems can be formulated as MIP in several different ways. Choosing a "good" model is of crucial importance to solving the model. The pruning conditions (1) and (2) will work earlier for a model with "tighter" LP relaxation.

Network flow formulation

In order to find the most appropriate MIP model for PTP, we start by restating it as a network optimization problem. Let $A = \{(i,j) \in L \mid j - i = 1\}$ be the set of interactions between adjacent segments and let $R = L \setminus A$ be the set of remote links. We introduce a digraph $G(V, E)$ with vertex set $V = \{(i,k) \mid i = 1, \ldots, m,\ k = 1, \ldots, n\}$ and arc set $E = E_L \cup E_x$, where

$$E_L = \{((i,k),(j,l)) \mid (i,j) \in L,\ 1 \le k \le l \le n\},$$
$$E_x = \{((i,k),(i+1,l)) \mid i = 1, \ldots, m-1,\ 1 \le k \le l \le n\}.$$

A vertex $(i,k) \in V$ corresponds to segment i placed on its kth relative position. The set E_L corresponds to the set of links L between the segments. The arcs from $E_x \setminus E_L$ are added to ensure the order of the segments. Depending on the situation, a set of arcs E_z is defined as either $E_L \setminus E_x$, corresponding in this case to the links from R, or as E_L. The default value is the first definition, but in either case, $E = E_x \cup E_z$. The arcs from E_x are referred to as x-arcs, and the arcs from E_z as z-arcs.

By adding two extra vertices, S and T, and arcs $(S, (1,k))$, $k = 1, \ldots, n$ and $((m,l), T)$, $l = 1, \ldots, n$, (considered as x-arcs), it is easy to see the one-to-one correspondence between the set of the feasible threadings and the set of the S-T paths in (V, E_x). A threading $(r_1, \ldots, r_m)$ corresponds to the S-T path $S - (1, r_1) - \cdots - (m, r_m) - T$, and vice versa. Fig. 1.3 illustates this correspondence.

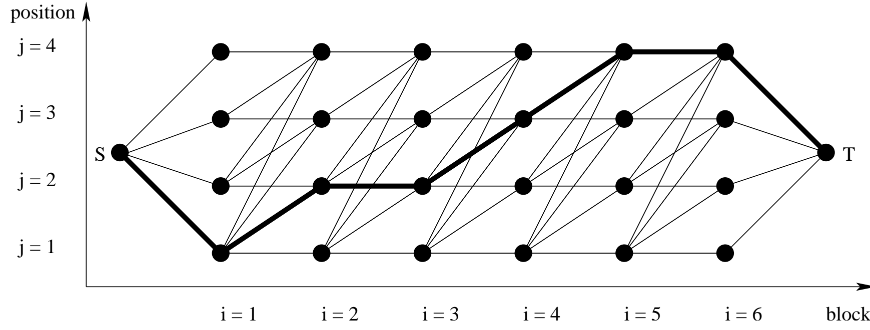


Figure 1.3: Example of the graph (V, E_x). The path in thick lines corresponds to the threading $(1, 2, 2, 3, 4, 4)$.

To each arc $e = ((i, k), (j, l)) \in E$ we associate a cost denoted by c_{ikjl}, or simply c_e. The costs of the arcs are related to the score coefficients introduced in the previous section. The cost of each x-arc is sum of three components: (1) the head of the arc (segment-to-position score), (2) the score of the loop between the adjacent segments (if any), and (3) the score of the interaction between the adjacent segments (if any). If the leading/trailing gaps are scored then their scores are associated to the outgoing/incoming arcs from/to the vertex S/T. The costs of z-arcs correspond to the pairwise segment interaction scores. In some of the models, the cost for placing the segment i on position k will be associated to the vertex (i, k) and denoted by c_{ik}.

An S-T path is said to activate the z-arcs that have both ends on this path. Each S-T path activates exactly $|R|$ z-arcs, one for each pair of segments in R. The subgraph induced by the x-arcs of an S-T path and the activated z-arcs is called an augmented path. Thus, solving PTP is equivalent to finding the shortest augmented path in G (as usual, the length of an augmented path is defined as the sum of the costs of its arcs). Fig. 1.4 provides an example of augmented path.

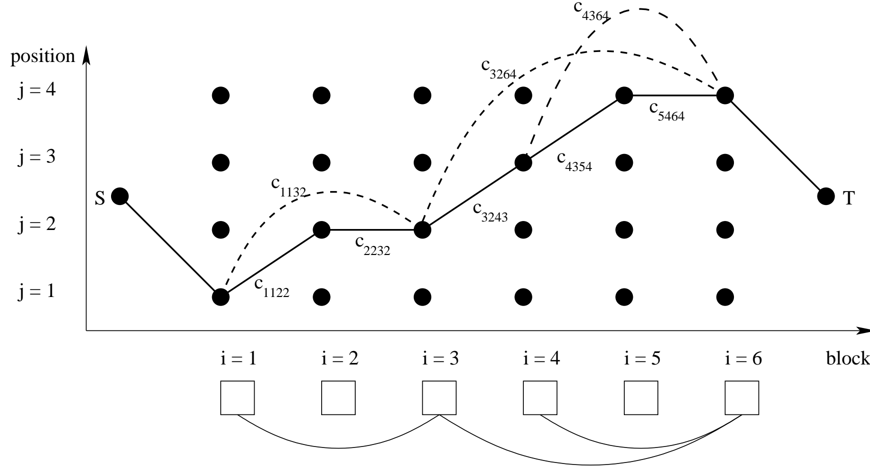


Figure 1.4: Example of augmented path. The generalized contact map graph is given in the bottom. The x arcs of the S-T path are in solid lines. The activated z-arcs are in dashed lines. The length of the augmented path is equal to the score of the threading $(1, 2, 2, 3, 4, 4)$.

Let us introduce the variables x_e, $e \in E_x$, z_e, $e \in E_z$, and y_v, $v \in V$. These variables will sometimes be denoted by $x_{i,k,i+1,l}$, z_{ikjl}, and y_{ik}. By interpreting x_e as a flow on the arc e, the problem of finding an S-T path in G becomes a problem of sending unit flow from S to T. In other words, there is one-to-one correspondence between the S-T paths and the vertices of the network-flow polytope X, defined by the constraints

$$\sum_{e \in \Gamma(S)} x_e = 1 \tag{1.1}$$

$$\sum_{e \in \Gamma^{-1}(T)} x_e = 1 \tag{1.2}$$

$$\sum_{e \in \Gamma(v)} x_e - \sum_{e \in \Gamma^{-1}(v)} x_e = 0 \qquad v \in V \tag{1.3}$$

$$x_e \geq 0 \qquad e \in E_x \tag{1.4}$$

where $\Gamma(v)$ (respectively $\Gamma^{-1}(v)$) denotes the set of the x-arcs with tail (respectively head) v. Constraint (1.1) (respectively (1.2)) corresponds to unit flow from S (respectively to T), and constraints (1.3) refer to flow conservation for each vertex. The well known properties of the network polytope X make it possible to replace the integrality requirements $x_e \in \{0, 1\}$ by $x_e \geq 0$.

The set of feasible threadings can also be expressed in the space of y-variables as a set Y,

defined by the constraints

$$\sum_{k=1}^{n} y_{ik} = 1 \qquad i = 1, \ldots, m \tag{1.5}$$

$$\sum_{l=1}^{k} y_{il} - \sum_{l=1}^{k} y_{i+1,l} \geq 0 \qquad i = 1, \ldots, m-1,\ k = 1, \ldots, n-1 \tag{1.6}$$

$$y_{ik} \in \{0, 1\} \qquad i = 1, \ldots, m,\ k = 1, \ldots, n \tag{1.7}$$

The binary variable y_{ik} is 1 if and only if segment i is placed on position k. Constraints (1.5) assign each segment to exactly one position and (1.6) ensure the order of segments (if segment i is placed after the kth position, then $i+1$ must also be placed after the kth position).

Starting from the S-T path defining sets X or Y, an augmented path defining set Z can be constructed by introducing the z-variables and adding appropriate connecting constraints. In this way, different MIP models for PTP can be obtained. Although equivalent, these models will either be easier, or more difficult, to solve using a given MIP solver, depending on their formulation. The strategy for deriving such models is: while keeping the vertices of the convex hull of the Z projection on X (or Y) invariant, either improve the LP bounds by tightening the linear relaxation $\overline{Z}$ of Z, or restate the model (maybe by chance) in a way that makes its LP relaxation easier for the chosen solver.

In the rest of this section we derive five MIP models for PTP. $F(\mathbf{M})$ and $\overline{F}(\mathbf{M})$ denote the feasible sets of a model **M** and its LP relaxation. $v(\mathbf{M})$ and $\overline{v}(\mathbf{M})$ refer respectively to the optimal objective values of a model and to its LP relaxation. The models from Sections 1.1.3-1.1.4 are first proposed in [41, 42], the model in Section 1.1.4 is from [4], and the one in Section 1.1.5 is introduced in [37]. Before describing the models, two easily verifiable observations that will be useful to remember when reading the model descriptions are:

Observation 1.1. Note that adding a constant to all arc costs does not change the set of optimal solutions. The same holds even if different constants d_{ij} are added to the costs c_{ikjl}, $1 \leq k \leq l \leq n$. This allows the objective function to be rotated in order to minimize the number of iterations of the simplex algorithm.

Observation 1.2. Let C be the set of segments participating in remote interactions, i.e. $C = \{i \mid (i,j) \in R \text{ or } (j,i) \in R \text{ for some } j\}$. It is easy to see that the number of S-T paths (which is also the number of feasible threadings) is $N_x = \binom{m+n-1}{m}$, and the number of the different z-components of all augmented paths is $N_z = \binom{|C|+n-1}{|C|}$.

For the example shown in Fig. 1.4, $C = \{1, 3, 4, 6\}$. The number of S-T paths (the number of possible fixations of the x-variables) is $N_x = \binom{6+4-1}{6} = 84$, and the number of possible fixations of the z-variables is $N_z = \binom{4+4-1}{4} = 35$. For fixed values of the z variables (in our case $z_{1132} = z_{3264} = z_{4364} = 1$, and all others equal to zero), the problem becomes simply a matter of finding the shortest S-T path passing through $(1,1)$, $(3,2)$, $(4,3)$ and $(6,4)$.

A nonlinear model using vertices

The most straightforward presentation of PTP as a mathematical programming problem is:

$$\min \left\{ \sum_{i=1}^{m} \sum_{k=1}^{n} c_{ik} y_{ik} + \sum_{(i,j)\in L} \sum_{k=1}^{n} \sum_{l=k}^{n} c_{ikjl} y_{ik} y_{jl} \mid y \in Y \right\}.$$

Despite the simplicity of this nonlinear 0-1 programming model, there are currently no algorithms or software able to solve efficiently such non-convex quadratic problems with thousands of binary variables. It is possible to linearize the model by introducing the z-variables. The products $y_{ik} y_{jl} = \min\{y_{ik}, y_{jl}\}$ can be replaced by z_{ikjl} in the objective function if the following tightest connecting constraints are added to the model [27]:

$$z_{ikjl} \le y_{ik},\ z_{ikjl} \le y_{jl},\ y_{ik} + y_{jl} - z_{ikjl} \le 1,\ 0 \le z_{ijkl} \le 1,\ (i,j) \in L,\ 1 \le k \le l \le n.$$

Note that the integrality of z is implied by the integrality of y. Although linear, the obtained model cannot be efficiently solved, mainly because of the weakness of its LP-bounds [41].

A model using x-arcs and z-arcs

The following model, later referred to as **MXZ**, is a restatement of the previous one in terms of network flow:

$$\text{Minimize} \quad \sum_{e \in E_x} c_e x_e + \sum_{e \in E_z} c_e z_e \tag{1.8}$$

$$\text{subject to:} \quad z_{ikjl} \le \sum_{e \in \Gamma(i,k)} x_e \qquad ((i,k),(j,l)) \in E_z \tag{1.9}$$

$$z_{ikjl} \le \sum_{e \in \Gamma^{-1}(j,l)} x_e \qquad ((i,k),(j,l)) \in E_z \tag{1.10}$$

$$\sum_{1 \le k \le l \le n} z_{ikjl} = 1 \qquad (i,j) \in R \tag{1.11}$$

$$x \in X \tag{1.12}$$

$$z_e \in \{0,1\} \qquad e \in E_z \tag{1.13}$$

The z-arcs are activated by a nonzero out-flow from their tail vertex (constraints (1.9)) and a nonzero in-flow into their head vertex (constraints (1.10)). The special ordered set (SOS) constraints (1.11) follow from the rest of the constraints, but even though they are redundant, they are included in the model primarily to control the branching strategy of the MIP solver.

For **MXZ** we can choose either x or z as the integer variables, as the integrality of x follows from the integrality of z, and vice versa. However, the better choice is z because, as mentioned in Observation 1.2, the space of z-variables is smaller than the space of x-variables.

1.1.4 A model using vertices, x-arcs, and z-arcs

In order to improve the LP-bounds and the branching strategy by imposing branching on SOS constraints (despite the expense of adding extra constraints), we modify the model **MXZ** by introducing the y-variables, associated to the vertices of the graph G. Thus we obtain the following model, denoted here by **MXYZ**:

$$\text{Minimize} \quad \sum_{e \in E_x} c_e x_e + \sum_{e \in E_z} c_e z_e \tag{1.14}$$

$$\text{subject to:} \quad y_{ik} = \sum_{l=k}^{n} z_{ikjl} \qquad (i,j) \in R,\ k = 1, \ldots, n \tag{1.15}$$

$$y_{jl} = \sum_{k=1}^{l} z_{ikjl} \qquad (i,j) \in R,\ l = 1, \ldots, n \tag{1.16}$$

$$y_{ik} = \sum_{e \in \Gamma(i,k)} x_e \qquad i \in C,\ k = 1, \ldots, n \tag{1.17}$$

$$\sum_{k=1}^{n} y_{ik} = 1 \qquad i \in C \tag{1.18}$$

$$y_{ik} \in \{0, 1\} \qquad i \in C,\ k = 1, \ldots, n \tag{1.19}$$

$$x \in X \tag{1.20}$$

$$z_e \geq 0 \qquad e \in E_z \tag{1.21}$$

The y-variables control the activation of z-arcs (constraints (1.15) and (1.16)), as well as the flow on x-arcs (constraints (1.17)). Constraints (1.18) and (1.19) correspond to (1.5) and (1.7) from the previous definition of the set Y. This model has no constraints corresponding to (1.6) in the set Y, because the order of the segments is imposed in X. From a computational point of view, even at the expense of adding new variables and constraints, **MXYZ** is preferable mainly for the following two reasons: (1) $\overline{F}$(**MXYZ**) $\subset$ $\overline{F}$(**MXZ**) (see Proposition 1), which means that $\overline{v}$(**MXYZ**) is a better bound on the optimal objective value than $\overline{v}$(**MXZ**); and (2) given that the y-variables are defined only for the segments in C, the size of the search space for this model is the same as in **MXZ** (see Observation 1.2), while the number of binary variables is $|C|n$, which is much less than $|R|\frac{n(n+1)}{2}$, the corresponding number in **MXZ**.

Proposition 1. $\overline{F}$(***MXYZ***) $\subset$ $\overline{F}$(***MXZ***).

Proof. It is easy to verify that constraints (1.15) and (1.17) imply (1.9); (1.16) and (1.17) imply (1.10); (1.15) and (1.18) imply (1.11). Therefore, $\overline{F}$(**MXYZ**) $\subseteq$ $\overline{F}$(**MXZ**). To prove that the inclusion is strict, consider the values of x and z shown in Fig. 1.5.

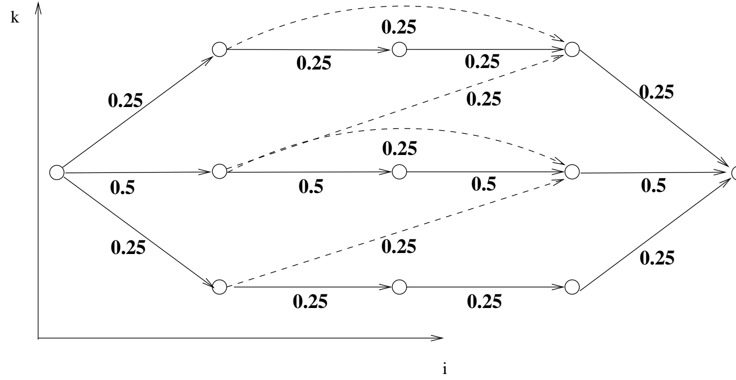


Figure 1.5: An instance of a network flow graph with $m = 3$, $n = 3$, and $L = \{(1, 3)\}$. The x-arcs are represented by solid lines and the z-arcs by dashed lines. Only the arcs with nonzero flow are given.The number associated to each arc is the value of the corresponding variable.

It is easy to see that these values satisfy (1.9)-(1.12). On the other hand, from (1.17) it follows that $y_{33} = 0.25$, while according to (1.16), $y_{33} = 0.5$. □

A model using vertices and z-arcs

The only combination left involves excluding the x-variables and testing the impact on the LP solver's efficiency of the possible reduction in the number of variables versus some increase in the number of constraints. Toward this end, the arcs in $E_x \setminus E_L$ are excluded from G and the arcs from $E_x \cap E_L$ are considered to be z-arcs. More precisely, in this model, $E_z = E_L$. The model **MYZ** obtained in this way is:

$$\text{Minimize} \quad \sum_{i=1}^{m}\sum_{k=1}^{n} c_{ik}y_{ik} + \sum_{e \in E_z} c_e z_e \tag{1.22}$$

$$\text{subject to:} \quad y_{ik} = \sum_{l=k}^{n} z_{ikjl} \qquad (i,j) \in L,\ k = 1,\dots,n \tag{1.23}$$

$$y_{jl} = \sum_{k=1}^{l} z_{ikjl} \qquad (i,j) \in L,\ l = 1,\dots,n \tag{1.24}$$

$$y \in Y \tag{1.25}$$

$$z_e \geq 0 \qquad e \in E_z \tag{1.26}$$

In this model, as in the previous one, the z-arcs are activated by the y-variables (constraints (1.23) and (1.24)). The gain with respect to **MXYZ** is about a 10% reduction in the number of variables, due to exclusion of non-duplicated x variables. Almost the same increase in the number of constraints due to (1.23) and (1.24) for $(i, j) \in A$ is observed for real-life instances. However, these modifications have a significant impact on the performance of the LP solver as we will see later.

1.1.5 The RAPTOR Model

Although derived differently, this model is very similar to the previous two. Let us make the following modifications in the model **MXYZ**: replace $i \in C$ by $i = 1, \ldots, m$ in constraints (1.17)-(1.19); replace the flow-tracing constraints (1.20) by the constraints

$$y_{jl} = \sum_{e \in \Gamma^{-1}(j,l)} x_e, \qquad j = 1, \ldots, m, \; l = 1, \ldots, n$$

connecting the y-variables and the tails of the x-arcs. In this way we obtain the **RAPTOR** model. It can also be derived from **MYZ** by replacing E_z with E, L with $L \cup \{(i, i+1) \mid i = 1, \ldots, m-1\}$, and removing constraints (1.6) (contained in (1.25)). Although seemingly minor, these modifications have a significant effect on the performance of the LP solver.

The LS algorithm and the self-threading case

Lathrop and Smith's B&B algorithm [18] uses lower bounds, which can be easily explained in terms of our network model. Consider a vertex $(i, k) \in V$, where $i \in C$. Let $P = S - (1, r_1) - \cdots - (i, k) - \cdots - (m, r_m) - T$ be an S-T path in G. Let

$$b(i, k, P) = \sum_{(i,j) \in R} c_{ikjr_j} + \sum_{(j,i) \in R} c_{jr_jik}$$

be the sum of the costs of the z-arcs with head or tail (i, k), activated by P. Let $b^*(i, k) = \min_P b(i, k, P)$, where the minimum is over all S-T paths passing through (i, k). If the costs of the x-arcs in $\Gamma(i, k)$ are updated by adding $\frac{1}{2}b^*(i, k)$, then the cost of the shortest S-T path in G is a lower bound on the optimal objective value ($b^*(i, k)$ is multiplied by $\frac{1}{2}$ because each arc is counted twice, once for its head and once for its tail). The $O(m^2 n^2)$ complexity of this procedure could be derived from the complexity of the shortest-path problem and the complexity of the algorithm for computing b^*. Lathrop and Smith provide a list of impressive computational results for a reach set of the so-called self-threading instances (the protein sequence is aligned to its own structure template). Andonov et al.[41, 42, 4] tested the MIP models on a large set of self-threading instances, and the results were always the same: the LP relaxation attains its minimum at a feasible (hence optimal) 0-1 vertex. The relaxation used in the **LS** algorithm is based on minimizing a function $\underline{f}(x)$, which is inferior to the objective function $f(x)$, over the set of feasible threadings. For such a relaxation, an optimal solution x^* to the relaxed problem is optimal for the original problem if $f(x^*) = \underline{f}(x^*)$. For the **LS** model, this is a kind of self-threading subclass defining property, and in most of the cases the optimal solution is found at the root of the B&B tree. This is the reason for the efficiency of the **LS** algorithm on instances in this subclass.

1.1.6 Experimental comparison of the MIP models

Extensive computational experiments in [41, 42] showed that **MXYZ** significantly outperforms the **LS** algorithm and the **MXZ** model. Here we present results from [4] which compare the models **MXYZ**, **MYZ** and **RAPTOR**. The models were solved using CPLEX 7.1 on a Pentium 2.40

Table 1.1: A set of instances where the LP solution is integer.

query length	space size	MXYZ iter.	MXYZ time	RAPTOR iter.	RAPTOR time	MYZ iter.	MYZ time
491	2.9e21	13624	25	13864	26	7907	13
491	2.5e25	22878	83	25747	118	10566	29
522	1.8e26	20627	111	15723	94	7920	22
491	1.1e27	41234	276	47082	347	16094	58
455	1.5e29	30828	390	36150	596	25046	241
522	1.8e29	18949	161	18598	169	12307	77
491	1.1e30	28968	365	40616	604	13870	68
491	1.4e30	58602	1303	66816	2083	29221	401
491	3.2e30	34074	572	41646	659	22516	186
522	5.3e31	26778	334	33395	468	13752	64
294	4.1e38	43694	619	52312	749	36539	314
583	1.3e39	124321	6084	147828	8019	57912	1120
757	9.9e45	121048	4761	166067	7902	92834	3117

Times are in seconds. For all instances, the **MYZ** model provides the best time.

GHz Linux PC with 4GB of RAM. The models are compared on real-life instances generated by FROST (Fold Recognition Oriented Search Tool) software [21].

The most important observation is that for almost all (about 95%) of the instances, the LP relaxation of all three models is integer-valued, thus providing optimal threading. This is true even for polytopes with more than 10^{46} vertices. Table 1.1 shows the number of simplex iterations and the time for each model on a sample from these instances. More precisely, of the 3600 instances solved by the **MYZ** model, the LP solution is non-integer in only 182 cases (about 5%). In all these cases, the solution contains mainly zeros and ones, and several 0.5 values. The largest number of nodes in the B&B tree for the 182 non-integer instances is 11; only 17 instances have a tree with six or more nodes (these instances are shown in Table 1.2). In most instances, only two nodes are sufficient for attaining optimality. The behavior of the other two models is similar. The number of nodes is slightly different from one model to another, and from CPLEX 7.1 to 8.0, but always stays in the same range of values. Table 1.2 shows that most of the solution time is usually spent in solving the LP relaxation. In addition, the gap between the LP score and the MIP score is so small that the LP relaxation could very well predict the closest template when comparing a query to multiple templates. There are, in fact, many similarities with the classic uncapacitated plant-location problem, in which most of the instances are easy to solve [10, 40]. The good behavior of the LP relaxation is definitely lost when using randomly generated score coefficients, but we believe that the behavior will stay good for each reasonable additive scoring scheme. In complexity-theory parlance, these observations can be summarized as: the subset of real-life instances of PTP is polynomially solvable.

These results show that all practical instances of PTP could be solved in affordable time using only a general-purpose LP solver and one of the MIP models described above. The problems with non-integer LP solutions could be driven to feasibility by using a simple ad-hoc procedure

Table 1.2: A set of instances with a non-integer LP Solution.

				MXYZ		RAPTOR		MYZ	
query length	space size	LP score	MIP score	LP time	MIP time	LP time	MIP time	LP time	MIP time
300	3.8e20	57.5	60.8	230	468	338	832	207	320
344	5.2e23	152.1	156.2	125	260	196	368	79	319
364	1.1e23	185.5	188.0	466	562	576	1642	262	635
416	4.1e26	314.8	318.8	219	523	357	552	91	168
394	1.9e28	214.2	217.1	425	599	692	1167	117	273
394	2.1e28	201.4	202.9	458	599	834	1140	208	421
427	1.6e29	256.3	261.6	1308	1692	2025	3109	304	653
508	1.3e30	316.2	317.5	195	281	339	447	72	126
491	1.8e30	97.4	98.1	245	262	427	545	70	87
511	1.9e31	415.7	420.1	1908	3893	3129	4053	1012	1773
508	3.9e32	180.3	181.9	841	1008	1389	1666	293	422
508	1.7e33	370.2	370.2	1292	1356	1706	2117	908	1182
491	1.2e33	90.2	90.8	542	927	737	827	202	218
522	6.6e34	-12.4	-11.8	1678	1723	1928	2119	258	293
583	1.3e36	-297.5	-296.6	1900	2533	4372	4648	773	910
508	1.23e38	347.8	354.5	4711	9349	6346	17903	1657	3949
508	1.66e40	201.1	210.4	8031	13449	10588	27055	2504	9631

Times are in seconds. The LP Relaxation value is the same for all models, and the gap between this value and the optimal solution is relatively small. **MYZ** is significantly faster than the other models.

instead of a MIP solver. As for the minimality of the polytope-describing constraints of the proposed MIP models, we must note that any attempt of aggregation (for example (1.15), (1.16), and (1.18) in **MXYZ**) spoils the integrality of the LP solution.

All numerical experiments show that the constraints (1.6) influence CPLEX solution times very favorably. Consequently, the **MYZ** model significantly outperforms **MXYZ** and **RAPTOR** for all large instances (see the corresponding columns in Tables 1.1 and 1.2).

1.1.7 Divide and conquer

The main drawback of the MIP models presented in the previous section is their huge number of variables and constraints. Although their LP relaxations yield optimal solutions for most real-life instances, solving these relaxations is still computationally expensive, and sometimes even impossible due to memory requirements. A possible approach to increasing efficiency is to split the problem into subproblems and to solve them separately. In this section we present such divide-and-conquer technique for the **MYZ** model.

Let $1 \leq L_1 \leq \cdots \leq L_m \leq n$, $1 \leq U_1 \leq \cdots \leq U_m \leq n$, $L_i \leq U_i$, $i = 1, \ldots, m$ be lower and upper bounds of the segment positions. These bounds can be imposed by setting to zero (removing) the variables y_{ik}, $i = 1, \ldots, m$, $k \notin \overline{L_i, U_i}$ and z_{ikjl}, $(i, j) \in R$, $k \notin \overline{L_i, U_i}$, $l \notin \overline{L_j, U_j}$

in the **MYZ** model. The resulting model is called $\mathbf{MYZ}(L,U)$ and its optimal objective value is denoted $v(L,U)$. Now if $\{(L^s,U^s) \mid s \in S\}$ is a set of bounds partitioning the space of feasible threadings, then $v(\mathbf{MYZ}) = \min\{v(L^s,U^s) \mid s \in S\}$. In other words, the solution to the original problem is the best of the subproblem solutions.

Splitting the problem into subproblems is useful for two reasons.First, each subproblem has a smaller number of variables and is thus easier to solve than the original problem. Second, and more important, information from the previously solved subproblems can be used when solving the next subproblem. Suppose that the first s subproblems have already been solved, and let v^* be the best of their objective values. Then v^* can be used as a cutoff value when solving subproblem $s+1$. If the LP relaxation of this subproblem is solved using a dual simplex method, the optimization can be stopped prematurely if the objective function exceeds v^*. The same cut can be applied not only at the root, but in any node of the B&B tree.

There are many ways to split the space of feasible threadings. Andonov et al. [4] have tested two natural, easy-to-implement, and efficient possibilities. Let us fix a segment i and partition the interval $\overline{1,n}$ of its possible positions into q subintervals $\overline{L_i^s,U_i^s}$, $s = 1,\dots,q$ of lengths approximately $\frac{n}{q}$ each. In this way, a partition of the feasible threadings space (L^s,U^s), $s = 1,\dots,q$, is obtained, where

$$\begin{aligned}
&L_i^s = 1 + (s-1)\left\lfloor \tfrac{n}{q} \right\rfloor + \min(s-1, n \bmod q),\ U_i^s = s\left\lfloor \tfrac{n}{q} \right\rfloor + \min(s, n \bmod q)\\
&L_j^s = 1,\ U_j^s = U_i^s,\ j = 1,\dots i-1\\
&L_j^s = L_i^s,\ U_j^s = n,\ j = i+1,\dots m
\end{aligned}$$

Intuitively, the feasible space of subproblem s is composed of the S-T paths in the network that pass through the vertices (i,k), $k \in \overline{L_i^s,U_i^s}$ (see Fig. 1.6(a)).

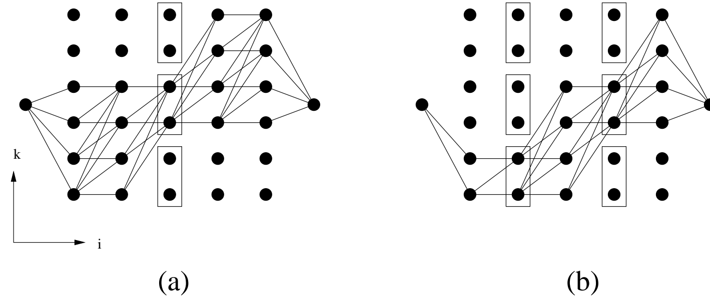


Figure 1.6: Instance with five segments and six free positions. (a) The problem is split on segment 3 into 3 subproblems. The easible set of the second subproblem is defined by $L^2 = (1,1,3,3,3)$ and $U^2 = (4,4,4,6,6)$. (b) The problem is split on segments 2 and 4 into 6 subproblems. The feasible set of the second subproblem is defined by $L^2 = (1,1,1,3,3)$ and $U^2 = (2,2,4,4,6)$.

Above, we defined a splitting for a fixed segment i and a fixed number of subproblems q. A natural question is how to choose good values for these parameters. For a fixed q, a good strategy is to choose the segment i in such a way that the most difficult of the resulting subproblems

become easier to solve. Formally, if the number of variables is considered as an approximate measure of subproblem's difficulty, a good choice would be:

$$i = \operatorname*{argmin}_{1\le j\le m} \left\{ \max_{1\le s\le q} \nu_{js} \right\},$$

where ν_{js} is the number of variables of subproblem s, split on the jth segment. The choice of the number of subproblems q is not obvious and will be discussed later.

The order in which the subproblems are solved is very important to the efficiency of this approach. A better record v^* allows earlier cuts in the subproblems that follow. Statistically, the chance of finding the global optimum in a subproblem is proportional to the size of its search space. It is not difficult to see that the number of S-T paths passing through a vertex (i, k) is $\binom{i+k-2}{i-1}\binom{m+n-i-k}{m-i}$. From this, it is easy to compute the search space size of each subproblem and to sort the subproblems in decreasing order according to their size.

This splitting technique can be generalized for two (or more) segments. Consider two segments i and j (see Fig. 1.6(b)). As before, the possible positions for each segment are partitioned into q subintervals, yielding $\frac{q(q+1)}{2}$ subproblems. Segments i and j can be chosen to minimize the maximal number of variables in the resulting subproblems, and the subproblems can be solved in decreasing order on their search space size. In the following paragraphs, SPLIT1 and SPLIT2 denote partitioning based on one and two segments. Finding the best splitting segment(s) in SPLIT1 requires considering each of the m segments, while in SPLIT2 $\frac{m(m-1)}{2}$ pairs of segments must be enumerated. In both cases, the time needed to choose the splitting segments is negligible with respect to the time needed to solve the resulting subproblems.

Andonov et al. [4] have tested these splitting techniques on a large set of threading instances. Table 1.3 presents the running times for SPLIT1 and SPLIT2 on a representative subset of these instances. The experiment shows that:

- Splitting reduces the running time by a factor of more than two for bigger instances when an appropriate number of subproblems is chosen.
- Running time decreases up to certain number of subproblems, and and then begins to increase again. The best number of subproblems is relatively small (no more than 15 for all solved instances). This phenomenon is due not only to the increased overhead for the initialization of the subproblems, but also to the smaller chance of finding the globally optimal solution in one of the first subproblems (see also Table 1.4).
- It is difficult to determine the optimal number of subproblems before solving them. This number is different for each instance and depends on the structure of the subproblems.
- SPLIT2 is more robust, in that its running time increases more slowly with the number of subproblems. Although SPLIT1 is clearly the winner for three subproblems, in the case of ten or more subproblems, SPLIT2 is the better choice. This observation is illustrated by rows "# instances where SPLIT1(SPLIT2) is better" in Table 1.3. Using SPLIT2 is preferable, because even when the number of subproblems is chosen randomly, there is a smaller chance of making a serious "mistake."

Table 1.3: Running times for SPLIT1 and SPLIT2 with different number of subproblems.

number of subproblems		1	3	6	10	15	21	28	36
space	split								
2.80e+19	1	13	**9**	11	13	19	24	31	32
	2	13	**10**	10	13	14	17	20	25
2.94e+21	1	33	**16**	17	22	27	35	45	55
	2	33	**21**	22	21	26	28	37	38
2.47e+25	1	80	**39**	40	57	76	102	123	154
	2	80	**40**	40	40	47	57	68	79
1.75e+26	1	99	**72**	101	142	191	257	335	421
	2	99	**77**	101	111	141	179	212	247
1.10e+27	1	308	95	**92**	115	153	189	251	292
	2	308	102	**92**	102	108	122	148	171
1.53e+29	1	1115	**329**	392	491	645	788	993	1163
	2	1115	392	604	**369**	441	635	662	744
1.78e+29	1	364	**144**	192	291	390	528	677	818
	2	364	**163**	175	195	243	312	381	478
1.09e+30	1	292	**134**	181	216	303	388	501	612
	2	292	167	**158**	225	276	273	342	419
1.44e+30	1	1117	482	**463**	512	676	840	1094	1314
	2	1117	511	**457**	464	534	660	768	800
3.20e+30	1	802	**314**	405	515	719	903	1216	1484
	2	802	322	366	**318**	396	525	665	763
5.34e+31	1	524	**277**	352	531	728	908	1020	1308
	2	524	**329**	409	405	475	496	561	701
# instances where SPLIT1 is better			11	3	1	0	0	0	0
# instances where SPLIT2 is better			0	5	9	11	11	11	11
average speedup SPLIT1			2.3	2.0	1.5	1.1	0.9	0.7	0.6
average speedup SPLIT2			2.0	1.9	1.9	1.6	1.3	1.1	1.0

The running times are in seconds. The best running time in each row is bold.

Table 1.4 presents the subproblem in which the global optimum was found. It shows that solving the subproblems in decreasing order on their search-space size yields good results. In many cases, the optimal solution is found when solving the first subproblem, and the rest of the subproblems are quickly abandoned using the obtained cutoff value. Even when the first subproblem does not contain the global optimum, it provides a good record that allows many of the following subproblems to be cut.

Table 1.4: Subproblem where the optimal solution is found

number of subproblems		3	6	10	15	21	28	36
space	split							
2.80e+19	1	1	2	3	2	2	5	5
	2	1	1	2	1	4	2	9
2.94e+21	1	1	1	1	2	1	2	1
	2	1	4	1	8	4	19	12
2.47e+25	1	1	1	1	2	2	3	3
	2	1	1	1	1	3	1	2
1.75e+26	1	1	2	2	3	5	6	7
	2	1	3	1	3	3	6	6
1.10e+27	1	1	1	1	2	2	5	4
	2	1	2	1	1	2	4	2
1.53e+29	1	1	2	3	4	5	7	9
	2	1	2	1	3	2	7	3
1.78e+29	1	1	1	2	3	4	6	5
	2	1	2	1	2	2	4	3
1.09e+30	1	1	2	1	2	3	4	5
	2	1	3	5	6	2	5	9
1.44e+30	1	1	1	1	1	1	1	1
	2	1	1	1	1	2	3	1
3.20e+30	1	1	2	2	4	6	7	11
	2	3	2	8	3	5	5	12
5.34e+31	1	3	4	6	8	12	15	18
	2	3	6	7	10	16	8	13

1.1.8 Parallelization

In the previous sections we have shown that an appropriate MIP model combined with divide-and-conquer strategy can significantly decrease the time for solving PTP. The application of these techniques allows to increase the size of practically solvable instances from $10^{18} - 10^{20}$ to $10^{35} - 10^{38}$. However, given that PTP is NP-hard, it is not surprising that solving instances of bigger size is still slow. A natural way to accelerate the algorithms for solving PTP is to use parallel computing. There are two possible approaches to parallelization.

The high-level parallelization considers each PTP as a single task. Recall that the query sequence must be aligned to each template (or a filtered subset of templates) from the database. These alignments are independent tasks and can be executed in parallel. Pley et al. [28] propose such parallelization. Since the tasks are irregular, dynamic load balance is used.

In this section we present a lower level parallelization proposed by Andonov et al. [42, 4]. This approach uses the divide-and-conquer strategy from the previous section. The subproblems

are considered as tasks to be executed on p processors. The best objective value known by a given processor is called local record, and the global record is the best among the local records. The efficiency of the parallel algorithm depends essentially on propagating the global record quickly among the processors in order to use it as a cut.

Parallel Algorithm

The parallel algorithm proposed by Andonov et al. [4] is based on centralized dynamic load balancing: tasks are dispatched from a centralized location (pool) in a dynamic way. The work pool is managed by a "master" who gives work on demand to idle "slaves." The master is also responsible for communicating new records to the slaves. Each slave solves the subproblem assigned to it, using the model **MYZ**. Note that dynamic load balancing is the only reasonable task allocation method when dealing with irregular tasks for which the amount of work is not known prior to execution.

To propagate the global record quickly, the CPLEX callback functions are used. These are user functions called at each simplex iteration (LP callback) and at each node of the B&B tree (MIP callback). The slaves use the MIP callback to communicate the local record to the master (if this record is improved at the current node). The LP callback is used to probe for new records coming from outside and to stop the optimization if the LP objective value becomes greater than the record. In the experiments, the local record was hardly ever updated—about once for every thousand simplex iterations. Furthermore, the only information exchanged between the master and a slave when a new task is transmitted is the number of this task. This information is sufficient for the slave to instantiate the CPLEX problem object directly in its own memory. This, together with the reduced number of record exchanges, makes this parallel implementation very efficient.

Computational Experiments

The numerical results presented in this section were obtained by using the CPLEX 7.1 callable library and the MPI communication library on a cluster of 16, 2 $\times$ Pentium III 1GHz 512MB Linux PCs connected by a Myrinet network. As in Section 1.1.6, the PTP instances are generated by FROST [21]. As discussed in Section 1.1.7, it is extremely difficult to predict the number of subproblems that will minimize running time. The problem becomes even more complicated when there are multiple processors. For these reasons, the algorithm was run on a large set of various instances and its behavior was observed, with both varying the problem granularity and the number of processors. Based on the resulting statistics, Andonov et al. [4] derive empirically a range of values for the observed parameters for which the program has close-to-optimal behavior.

Tables 1.5 and 1.6 summarize the execution times of the parallel code on a set of representative instances, using SPLIT1 and SPLIT2 respectively. The pth row gives the running times using p processors, $p = 1, \ldots, 12$. Columns 2-9 correspond to the different granularities (the number of subproblems is shown in the header). ∞ denotes cases in which the problem was not solved due to insufficient memory. This happened only for the original (non split) problem. The

Table 1.5: Running times for SPLIT1

p	1	3	6	10	15	21	28	36	avg	stddev	s_up	eff
$m = 33, n = 172, \|R\| = 84, \|\mathcal{T}\| = 1.23e38$												
1	17673	**8647**	**2517**	**2431**	2656	2309	2756	2809	4531	2910	1.0	1.0
2		2972	**1915**	**1341**	**1736**				1664	239	2.7	1.4
4			839	**864**	**1051**	**1145**			1020	116	4.4	1.1
6			810	**503**	**778**	**857**			712	151	6.4	1.1
8				482	**632**	**708**	**716**		685	37	6.6	0.8
10				481	**525**	**644**	**564**		577	49	7.8	0.8
12					523	**595**	**507**	**567**	556	36	8.1	0.7
$m = 31, n = 212, \|R\| = 81, \|\mathcal{T}\| = 1.29e39$												
1	3036	**2450**	**1126**	**1520**	868	1137	1381	1257	1698	555	1.0	1.0
2		824	**421**	**796**	**1141**				786	294	2.2	1.1
4			277	**351**	**500**	**469**			440	64	3.9	1.0
6			276	**279**	**338**	**346**			321	29	5.3	0.9
8				279	**310**	**232**	**325**		289	40	5.9	0.7
10				278	**309**	**203**	**285**		265	45	6.4	0.6
12					311	**189**	**188**	**237**	204	22	8.3	0.7
$m = 27, n = 225, \|R\| = 71, \|\mathcal{T}\| = 1.33e36$												
1	∞	**712**	**497**	**664**	472	557	613	704	624	92	1.0	1.0
2		654	**376**	**438**	**485**				433	44	1.4	0.7
4			296	**311**	**319**	**346**			325	15	1.9	0.5
6			294	**312**	**253**	**261**			275	26	2.3	0.4
8				312	**254**	**260**	**279**		264	10	2.4	0.3
10				310	**252**	**260**	**280**		264	11	2.4	0.2
12					252	**260**	**278**	**230**	256	19	2.4	0.2
$m = 30, n = 219, \|R\| = 90, \|\mathcal{T}\| = 4.13e+38$												
1	∞	**680**	**699**	**441**	523	693	1719	796	606	117	1.0	1.0
2		1245	**321**	**522**	**980**				607	275	1.0	0.5
4			135	**165**	**276**	**340**			260	72	2.3	0.6
6			92	**124**	**154**	**242**			173	50	3.5	0.6
8				114	**111**	**233**	**233**		192	57	3.2	0.4
10				78	**109**	**127**	**203**		146	40	4.1	0.4
12					109	**106**	**138**	**200**	148	39	4.1	0.3
$m = 32, n = 123, \|R\| = 86, \|\mathcal{T}\| = 1.19e33$												
1	443	**208**	**245**	**139**	179	240	287	290	197	43	1.0	1.0
2		150	**120**	**139**	**148**				135	11	1.5	0.7
4			82	**88**	**84**	**95**			89	4	2.2	0.6
6			82	**67**	**63**	**69**			66	2	3.0	0.5
8				63	**59**	**55**	**67**		60	5	3.3	0.4
10				63	**59**	**52**	**56**		55	2	3.5	0.4
12					59	**45**	**52**	**56**	51	4	3.9	0.3

Table 1.6: Running times for SPLIT2

p	1	3	6	10	15	21	28	36	avg	stddev	s_up	eff
$m = 33, n = 172, \|R\| = 84, \|\mathcal{T}\| = 1.23e38$												
1	17673	**8647**	**2517**	**2431**	2656	2309	2756	2809	4531	2910	1.0	1.0
2		3878	**1173**	**1400**	**1419**	1105	1562	1467	1330	111	3.4	1.7
4			703	**714**	**677**	**576**	700	719	655	58	6.9	1.7
6			657	**606**	**500**	**397**	510	515	501	85	9.0	1.5
8				614	**421**	**374**	**401**	431	398	19	11.4	1.4
10				374	**324**	**313**	**349**	394	328	15	13.8	1.4
12					243	**246**	**336**	**351**	311	46	14.6	1.2
$m = 31, n = 212, \|R\| = 81, \|\mathcal{T}\| = 1.29e39$												
1	3036	**2450**	**1126**	**1520**	868	1137	1381	1257	1698	555	1.0	1.0
2		983	**693**	**771**	**466**	592	663	656	643	129	2.6	1.3
4			342	**504**	**298**	**336**	354	329	379	89	4.5	1.1
6			343	**309**	**230**	**277**	185	222	272	32	6.2	1.0
8				259	**228**	**232**	**150**	179	203	37	8.4	1.0
10				258	**185**	**189**	**136**	154	170	24	10.0	1.0
12					185	**190**	**130**	**152**	157	24	10.8	0.9
$m = 27, n = 225, \|R\| = 71, \|\mathcal{T}\| = 1.33e36$												
1	∞	**712**	**497**	**664**	472	557	613	704	624	92	1.0	1.0
2		904	**383**	**393**	**316**	343	373		364	34	1.7	0.9
4			232	**159**	**173**	**140**	158	182	157	13	4.0	1.0
6			185	**123**	**113**	**106**	116	146	114	7	5.5	0.9
8				119	**109**	**120**	**90**	95	106	12	5.9	0.7
10				119	**109**	**109**	**89**	85	102	9	6.1	0.6
12					146	**112**	**68**	**70**	83	20	7.5	0.6
$m = 30, n = 219, \|R\| = 90, \|\mathcal{T}\| = 4.13e38$												
1	∞	**680**	**699**	**441**	523	693	1719	796	606	117	1.0	1.0
2		591	**187**	**261**	**318**	289	522	417	255	53	2.4	1.2
4			160	**146**	**153**	**166**	192	216	155	8	3.9	1.0
6			104	**115**	**126**	**115**	125	159	118	5	5.1	0.9
8				81	**107**	**89**	**99**	123	98	7	6.2	0.8
10				59	**107**	**93**	**89**	105	96	7	6.3	0.6
12					94	**67**	**78**	**83**	76	6	8.0	0.7
$m = 32, n = 123, \|R\| = 86, \|\mathcal{T}\| = 1.19e33$												
1	443	**208**	**245**	**139**	179	240	287	290	197	43	1.0	1.0
2		127	**103**	**80**	**102**	115	145	159	95	10	2.1	1.0
4			71	**51**	**63**	**62**	75	83	58	5	3.4	0.8
6			54	**39**	**46**	**48**	56	57	44	3	4.5	0.7
8				38	**40**	**39**	**46**	45	41	3	4.7	0.6
10				33	**36**	**35**	**39**	38	36	1	5.4	0.5
12					37	**29**	**34**	**36**	33	2	6.0	0.5

"avg" column contains the arithmetic mean of the values in bold from the corresponding row, while the "stddev" column gives the standard deviation of the same values. The column "s_up" contains the speedup, computed using the values from the column "avg" ($\text{avg}_1/\text{avg}_p$), and the last column shows the corresponding efficiency ($\text{s_up}_p/p$).

The last two columns characterize the average performance of the algorithm when the pair (processors, subproblems) belongs to the bold area in the corresponding row of the table. The bold area itself was determined by statistical analysis. For each row, the bold area consists of consecutive cells that optimize the average values of the last two columns over the set of instances considered in the experiment. Thus, when a pair (processors, granularity) is located in the bold area, the behavior of the algorithm is expected to be close to the data from the last two columns. The performed analysis makes an automatic choice of granularity possible, and transparent to the user.

The data presented in Tables 1.5 and 1.6 show that the SPLIT2 strategy is significantly better than SPLIT1. For SPLIT2, the mean running time decreases much faster than for SPLIT1, which naturally yields more attractive values in the last two columns. SPLIT2 is also more robust, as demonstrated by the standard deviations, which decrease faster with an increasing number of processors. In both cases, parallelism gives robustness to the algorithms: increasing the number of processors lessens the impact of granularity on execution time.

Table 1.7: Running times for instance with $m = 41$, $n = 194$, $|R| = 112$, $|\mathcal{T}| = 9.89e45$

p	1	3	6	10	15	21	28	36	45	55	66	avg	stddev	s_up	eff
1	∞	**4412**	**4726**	**3385**	2903	3638	3595	3931	3958			4174	572	1.0	1.0
2		3039	**1841**	**1755**	**1441**	1838	1870	2017	1980			1679	171	2.5	1.2
4			990	**1239**	**858**	**1010**	943	1019	1010			1035	156	4.0	1.0
6			955	**998**	**614**	**710**	673	680	692			774	163	5.4	0.9
8				686	**543**	**599**	**536**	519	535			559	28	7.5	0.9
10				681	**416**	**478**	**476**	425	440			456	28	9.1	0.9
12					415	**449**	**440**	**367**	387			418	36	10.0	0.8
16						464	**387**	**356**	**333**	352		358	22	11.6	0.7
18						383	**351**	**372**	**326**	313	359	349	18	11.9	0.7
24							343	**308**	**294**	**282**	307	294	10	14.2	0.6
26							373	**320**	**334**	**299**	296	317	14	13.1	0.5

Table 1.7 contains the running time for the biggest instance that was solved. The SPLIT2 strategy was used in this example. Up to 26 processors were used in order to observe the algorithm's efficiency. Although efficiency decreases below 0.8 when more than 12 processors are used, it remains above 0.5 even with 26 processors. The parallel algorithm is reasonably efficient up to about 10 processors. But this is not a serious drawback, because the query sequence is usually aligned to multiple templates. If there are more processors available, they can be used efficiently by separating them into groups and assigning a different query-to-template threading to each group.

1.1.9 Future research directions

In this section we discuss several possible extensions, improvements and open questions related to the algorithms presented in this chapter.

Properties of the PTP polytope It is well known that the network flow polytope defined by constraints (1.1)-(1.4) has only integral vertices. Unfortunately, it is not the case for the polytopes of the LP relaxations of the MIP models for PTP. It can be shown that even a single pairwise interaction between non-adjacent segments introduces non-integral vertices. However, as we have seen in Section 1.1.6, the optimum of the LP relaxations is attained in integral vertices for 95% of the real-life instances. Surprisingly, even small perturbations of the score coefficients move the optimum to non-integral vertex. This phenomenon is still unexplained. It would be interesting to study the structure of the PTP polytope and the relations between the properties of the score coefficients and the integrality of the LP solution. A study in this direction can also help to formulate tighter LP models.

Solving the MIP models by special-purpose methods. The advantage of MIP models presented in this chapter is that their LP relaxations give the optimal solution for most of the real-life instances. Their drawback is their huge size (both number of variables and number of constraints) which makes even solving the LP relaxation slow. The MIP models have very sparse and structured constraint matrix. Instead of solving them by general-purpose branch-and-bound algorithms using LP relaxation, one can try to design more efficient special-purpose algorithms. The first encouraging results in this direction are reported by Balev [5]. He proposes a MIP model similar to the ones discussed in Section 1.1.3 and solves it by branch-and-bound algorithm using Lagrangian relaxation. This algorithm is much faster than the general purpose methods. Another, still not investigated possibility is to take advantage of the network flow constraints and to design a decomposition network-simplex-like algorithm for solving the LP relaxations of the MIP models.

Variable segment lengths The definition of PTP given in Section 1.1.2 assumes that the length of each segment is fixed. This assumption is common for many threading methods but is somehow restrictive, since two proteins belonging to the same structural family may have a couple of amino acids attached to or detached from the endpoints of the corresponding secondary structure element. Taking into account this particularity could improve the quality of the threading method. It should be not very difficult to design MIP models with variable segment lengths, but this question is still open.

Semi-global threading The 3D structure of more complex proteins is formed of several compact structural units, called domains. While each database template describes a single domain, the query sequence can be a multi-domain protein. In this case the template must be aligned only to a (unknown) part of the query. We call this kind of alignment semi-global threading. The MIP models described in this chapter are flexible enough and can be easily adapted to semi-global

threading. This could be done by introducing extra constraints restricting either the length of each loop, or the sum of all loop lengths. The main difficulty will be to avoid the increase of the size and the complexity of the obtained models.

1.1.10 Conclusion

The protein threading problem is a challenging combinatorial optimization problem with very important practical applications and impressive computational complexity. Mathematical programming techniques, still not very popular in computational biology, can be a valuable tool for attacking problems that arise in this domain. The MIP models confirm that real-life instances are much easier to solve than artificial ones. The complexity of PTP is such that only an appropriate combination of different techniques can yield an efficient solution. By combining a careful choice of MIP formulation, a divide-and-conquer technique, and a parallel implementation, one can solve real-life instances of tremendous size in a reasonable amount of time.

The algorithms described in this chapter are already integrated in FROST [21] fold recognition software tool. They are general and flexible enough and can be readily plugged in (or easily adapted to) other threading-based fold recognition tools in order to improve their performance.

1.2 Optimal protein threading by cost-splitting

1.2.1 Motivation

Fold recognition methods based on threading are complex and time consuming computational techniques due to the still non-efficiently solved problems:

1. finding the best (with respect to the score function) possible sequence-3D structure alignment;

2. a statistical analysis of the raw scores allowing the detection of the significant sequence-structure alignments.

The first point above is related to the problem of finding the optimal sequence-to-structure alignment and is referred to as protein threading problem (PTP). From a computer scientist's viewpoint this is the most challenging part of the threading methods. Until recently, it was the main obstacle to the development of efficient and reliable fold recognition methods. In the general case, when variable-length alignment gaps are allowed and pairwise amino acid interactions are considered in the score function, PTP is NP-hard [48]. Moreover, it is MAX-SNP-hard [1], which means that there is no arbitrary close polynomial approximation algorithm, unless P = NP. In this context the progress done by the computational biology community in solving PTP during the last few years is really remarkable [18, 54, 43, 37, 38, 44]. The empirical results clearly illustrate that PTP is easier in practice than in theory and that it is possible to solve real-life (biological) instances in a reasonable amount of time. These results also show that one of the most promising approaches in solving this problem is using advanced mathematical programming (Mixed Integer Programming, MIP) models for PTP [42, 54, 43, 37]. The most amazing observation is that for almost all (more than 95%) of the instances, the LP relaxation of the MIP models is integer-valued, thus providing optimal threading. This is true even for polytopes with more than 10^{46} vertices. Moreover, when the LP relaxation is not integer, its value is a relatively good approximation of the integer solution. However, to the best of our knowledge, this observation has not been practically used before the current paper. Other successful Integer Programming approaches for solving combinatorial optimization problems originated in molecular biology are discussed in the recent survey [50].

The main drawback of mathematical programming approaches is that the corresponding models are often very large (over 10^6 variables). Even the most advanced MIP solvers need prohibitively large running time for solving such instances. For example, the authors in [37] find out 30 templates for which it takes about 15 hours to thread one target onto them on a Silicon Graphics Origin 3800 system, which has 40400 MHz MIPS R12000 CPUs and 20 GB of RAM. Different divide-and-conquer methods and parallel algorithms can be used to overcome this drawback [38, 42, 54].

A further step in solving the huge MIP models is the development of special-purpose algorithms based on advanced combinatorial optimization techniques like Lagrangian relaxation. Such an algorithm has been recently designed by S. Balev in [44] and computationally compared with the B&B algorithm from [48] and a heuristic used in [49]. The computational results are

very impressive and clearly show that the Lagrangian relaxation (LR) significantly outperforms both other algorithms. However, comparisons with MIP solver are not provided in [44].

In this section we continue the same direction of research and propose a new dedicated algorithm for solving protein threading MIP models. It is as well based on Lagrangian relaxation. But, both our Lagrangian dual formulation and the optimization technique that we use for solving it (the so-called cost-splitting [23]), differentiate from those described in [44]. Extensive computational results prove that: (i) our algorithm is in most cases faster than the one in [44]; (ii) both Lagrangian relaxation algorithms significantly outperform solving MIP models by LP relaxation. To the best of our knowledge the only other impressive application of LR to an alignment problem is discussed in [46].

Another contribution concerns the 2nd point above. When aligning a given query sequence to a set of 3D structures it is not possible to directly use the raw scores to rank the 3D structures. The reason is that these scores strongly depend on the query and template lengths and also, in a complicated way, on the particular features of the 3D structures. In addition, the query sequence may correspond to none of the existing folds. Therefore one must have means to evaluate the significance of an alignment score. This can be done as a preprocessing stage, by empirically calculating a distribution of scores for each template, using a set of sequences not related to it[2]. The underlying score normalization procedure involves threading a large set of queries against each template and requires solving millions of PTP. For example the package FROST (Fold Recognition-Oriented Search Tool) [49], uses a database of about 1200 known 3D structures, each one associated with empirically determined score distributions. Computing these distributions is extremely time consuming: it requires solving about 1200000 sequence-to-structure alignments and takes about 40 days on a 2.4 GHz computer and about 3 days on a cluster of 12 PCs [51]. Accelerating computations involved in this component is crucial for the development of efficient fold recognition methods.

Based on extensive comparisons we observe that the approximated solutions obtained by any one of the three algorithms considered here can be successfully used when computing scores distributions. Since these approximated solutions are obtained by polynomial algorithms, we experimentally prove that this heavy stage can be polynomially computed.

1.2.2 Protein threading problem revisited

For the sake of brevity, here we stick to the network optimization problem formulation proposed in [42, 43].

Towards this end we recall the definition of the set of feasible threadings Y, defined by the

[2]More justifications for this phase the interested reader can find in [49].

following constraints:

$$\sum_{k=1}^{n} y_{ik} = 1 \qquad i = 1, \dots, m \tag{1.27}$$

$$\sum_{l=1}^{k} y_{il} - \sum_{l=1}^{k} y_{i+1,l} \geq 0 \qquad i = 1, \dots, m-1,\ k = 1, \dots, n-1 \tag{1.28}$$

$$y_{ik} \in \{0, 1\} \qquad i = 1, \dots, m,\ k = 1, \dots, n \tag{1.29}$$

These constraints describe the set of feasible paths in a particular digraph (see the previous section), with vertex set $V = \{(i, j) \mid i = 1, \dots, m,\ j = 1, \dots, n\}$. The vertices $(i, j), j = 1, \dots, n$ will be referred to as ith layer. Each layer corresponds to a structural element, and each vertex in a layer corresponds to a positioning of this element on a query protein. Let $L \subseteq \{(i, k) \mid 1 \leq i < k \leq m\}$ be a given set of inter-layers links. This is the so-called *contact graph*: a link between layers i and k means that the corresponding structural elements are in contact in the 3D structure.

Let A_{ik} be the $2n \times \frac{n(n+1)}{2}$ node-arc incidence matrix for the subgraph spanned by the layers i and k, $(i, k) \in L$. The submatrix A_i, the first n rows of A_{ik}, (resp. A_k, the last n rows) corresponds to the layer i (resp. k). To avoid added notation we will use vector notation for the variables $y_i = (y_{i1}, ...y_{in}) \in B^n$ where B^n is the set of n-dimensional binary vectors, with assigned costs $c_i = (c_{i1}, ...c_{in}) \in R^n$ and $z_{ik} = (z_{i1k1}, \dots, z_{i1kn}, z_{i2k1}, \dots, z_{inkn}) \in B^{\frac{n(n+1)}{2}}$ for $(i, k) \in L$ with assigned costs $d_{ik} = (d_{i1k1}, \dots, d_{i1kn}, d_{i2k1}, \dots, d_{inkn}) \in R^{\frac{n(n+1)}{2}}$. In the sections below the vector d_{ik} will be considered as a $n \times n$ upper triangular matrix, having arbitrarily large coefficient below the diagonal. This slight deviation from the standard definition of an upper triangular matrix is used only for formal definition of some matrix operations.

Now the protein threading problem $PTP(L)$ is defined as:

$$z_{ip}^{L} = v(PTP(L)) = \min\{\sum_{i=1}^{m} c_i y_i + \sum_{(i,k) \in L} d_{ik} z_{ik}\} \tag{1.30}$$

$$\text{subject to: } y = (y_1, \dots, y_m) \in Y, \tag{1.31}$$

$$y_i = A_i z_{ik},\ y_k = A_k z_{ik} \qquad (i, k) \in L \tag{1.32}$$

$$z_{ik} \in B^{\frac{n(n+1)}{2}} \qquad (i, k) \in L \tag{1.33}$$

The shortcut notation $v(.)$ will be used for the optimal objective function value of a subproblem obtained from $PTP(L)$ with some z variables fixed.

1.2.3 Special cases

Throughout this section, vertex costs c_i are assumed to be zero. We study three sorts of contact graph that make PTP polynomially solvable.

Contact graph contains no crossing edges

Two links (i_1, k_1) and (i_2, k_2) such that $i_1 < i_2$ are said to be crossing when k_1 is in the open interval (i_2, k_2). The case when the contact graph L contains no crossing edges has been mentioned to be polynomially solvable for the first time in [1]. Here we present a different sketch for $O(n^3)$ complexity of PTP in this case.

If L contains no crossing edges, then $PTP(L)$ can be recursively divided into independent subproblems. Each of them consists in computing all shortest paths between the vertices of two layers i and k, discarding links that are not included in (i, k). Thus the result of this computation is a distance matrix D_{ik} such that $D_{ik}(j, l)$ is the optimal length between vertices (i, j) and (k, l). Note that for $j > l$ as there is no path in the graph, $D_{ik}(j, l)$ is an arbitrarily large coefficient. Finally, the solution of $PTP(L)$ is the smallest entry of D_{1m}.

We say that an edge $(i, k), i < k$ is included in the interval $[a, b]$ when $[i, k] \subseteq [a, b]$. Let us denote by $L_{(ik)}$ the set of edges of L included in $[i, k]$. Then, an algorithm to compute D_{ik} can be sketched as follows:

1. if $L_{(ik)} = \{(i, k)\}$ then the distance matrix is given by

$$D_{ik} = \begin{cases} d_{ik} & \text{if } (i,k) \in L \\ \tilde{0} & \text{otherwise} \end{cases} \tag{1.34}$$

   where $\tilde{0}$ is an upper triangular matrix in the previously defined sense (arbitrary large coefficients below the main diagonal) and having only zeros in its upper part.

2. otherwise as $L_{(ik)}$ has no crossing edges, there exists some $s \in [i, k]$ such that any edge of $L_{(ik)}$ but (i, k) is included in $[i, s]$ or in $[s, k]$. Then

$$D_{ik} = \begin{cases} D_{is}.D_{sk} + d_{ik} & \text{if } (i,k) \in L \\ D_{is}.D_{sk} & \text{otherwise} \end{cases} \tag{1.35}$$

   where the matrix multiplication is computed by replacing $(+, \times)$ operations on reals by $(\min, +)$.

Remark 1. If the contact graph has m vertices, and contains no crossing edges, then the problem is decomposed into $O(m)$ subproblems. For each of them, the computation of the corresponding distance matrix is a $O(n^3)$ procedure (matrix multiplication with $(\min, +)$ operations). Overall complexity is thus $O(mn^3)$. Typically, n is one or two orders of magnitude greater than m, and in practice, this special case is already expensive to solve.

1.2.4 All edges have their left end tied to a common vertex

A set of edges $L = \{(i_1, k_1), \ldots, (i_r, k_r)\}, k_1 < k_2 < \ldots k_r$ is called a *sheaf* if it has at least two elements and $i_t = i_1, t \leq r$. The arc costs corresponding to the link (i, k_s) are given by the upper triangular matrix d_{ik_s}.

The following algebra is used to prove the $O(n^2)$ complexity of the corresponding PTP.

Definition 1. Let A, B be two matrices of size $n \times n$. $M = A \otimes B$ is defined by $M(i,j) = \min_{i \leq r \leq j} A(i,r) + B(i,j)$

In order to compute $A \otimes B$, we use the following recursion: let M' be the matrix defined by $M'(i,j) = \min_{i \leq r \leq j} A(i,r)$, then

$$M'(i,j) = \min\{M'(i,j-1), A(i,j)\}, \text{ for all } j \geq i$$

Finally $A \otimes B = M' + B$. From this it is clear that $\otimes$ multiplication for $n \times n$ matrices is of complexity $O(n^2)$.

Theorem 1. *Let $L = \{(i,k_1), \ldots, (i,k_r)\}$ be a sheaf. Then $D_{ik_r} = (\ldots(d_{ik_1} \otimes d_{ik_2}) \otimes \ldots) \otimes d_{ik_r}$*

Proof. The proof follows the basic dynamic programming recursion for this particular case: for the sheaf $L = \{(i,k_1), \ldots, (i,k_r)\} = L' \bigcup \{(i,k_r)\}$, we have $v(L : z_{ijk_rl} = 1) = d_{ijk_rl} + \min_{j \leq s \leq l} v(L' : z_{ijk_{r-1}s} = 1)$. □

Sequence of independent subproblems

Given a contact graph $L = \{(i_1,k_1), \ldots, (i_r,k_r)\}$, $PTP(L)$ can be decomposed into two independent subproblems when there exists an integer $e \in (1,m)$ such that any edge of L is included either in $[1,e]$, either in $[e,m]$. Let $I = \{i_1, \ldots, i_s\}$ be an ordered set of indices, such that any element of I allows for a decomposition of $PTP(L)$ into two independent subproblems. Suppose additionally that for all $t \leq s-1$, one is able to compute $D_{i_t i_{t+1}}$. Then we have the following theorem:

Theorem 2. *Let $p = (p_1, p_2, \ldots, p_n)$ be obtained by the following matrix-vector multiplication $p = D_{i_1 i_2} D_{i_2 i_3} \ldots D_{i_{s-1} i_s} \overline{p}$, where $\overline{p} = (0, 0, \ldots, 0)$ and the scalar product in the matrix-vector multiplication is defined by changing "+" with "min" and "." with "+". Then for all i, $p_i = v(PTP(L : y_{1i} = 1)$, and $v(PTP(L)) = \min\{p_i\}$.*

Proof. Each multiplication by $D_{i_k i_{k+1}}$ in the definition of p is an algebraic restatement of the main step of the algorithm for solving the shortest path problem in a graph without circuits. □

Remark 2. With the notations introduced above, the complexity of $PTP(L)$ for a sequence of such subproblems is $O(sn^2)$ plus the cost of computing matrices $D_{i_t i_{t+1}}$.

From the last two special cases, it can be seen that if the contact graph can be decomposed into independent subsets, and if these subsets are single edges or sheaves, then there is a $O(srn^2)$ algorithm, where s is the cardinality of the decomposition, and r the maximal cardinality of each subset, that solves the corresponding PTP.

1.2.5 Relaxation through decomposition

In order to apply the results from the previous section, we need to find a suitable partition of L into $L^1 \bigcup L^2 ... \bigcup L^t$ where each L^s induces an easy solvable $PTP(L^s)$, and to use the s.c. cost-splitting variant of the Lagrangian duality. Now we can restate (1.30)–(1.33) equivalently as:

$$v_{ip}^L = \min \left\{ \sum_{s=1}^{t} (\sum_{i=1}^{m} c_i^s y_i^s + \sum_{(i,k) \in L^s} d_{ik} z_{ik}) \right\} \tag{1.36}$$

$$\begin{aligned} \text{subject to: } & y_i^1 = y_i^s, & & s = 2, t & & (1.37) \\ & y^s = (y_1^s, ..y_m^s) \in Y, & & s = 1, \dots, t & & (1.38) \\ & y_i^s = A_i z_{ik},\ y_k^s = A_k z_{ik} & & s = 1, \dots, t & (i,k) \in L^s \quad & (1.39) \\ & z_{ik} \in B^{\frac{n(n+1)}{2}} & & s = 1, \dots, t & (i,k) \in L^s \quad & (1.40) \end{aligned}$$

Taking (1.37) as the complicating constraints, we obtain the Lagrangian dual of $PTP(L)$:

$$v_{csd} = \max_{\lambda} \min_{y} \sum_{s=1}^{t} (\sum_{i=1}^{m} c_i^s(\lambda) y_i^s + \sum_{(i,k) \in L^s} d_{ik} z_{ik}) = \max_{\lambda} \sum_{s=1}^{t} v_{ip}^{L^s}(\lambda) \tag{1.41}$$

subject to (1.38), (1.39) and (1.40).

The Lagrangian multipliers λ^s are associated with the equations (1.37) and $c_i^1(\lambda) = c_i^1 + \sum_{s=2}^{t} \lambda^s$, $c_i^s(\lambda) = c_i^s - \lambda^s$, $s = 2, \dots, t$. The coefficients c_i^s are arbitrary (but fixed) decomposition (cost-split) of the coefficients c_i, i.e. given by $c_i^s = p_s c_i$ with $\sum p_s = 1$. From the Lagrangian duality theory follows $v_{lp} \leq v_{csd} \leq v_{ip}$. This means that for each PTP instance s.t. $v_{lp} = v_{ip}$ holds $v_{csd} = v_{ip}$. By applying the subgradient optimization technique ([23]) in order to obtain v_{csd}, one need to solve t problems $v_{ip}^{L^s}(\lambda)$ (see the definition of $v_{ip}^{L^s}$) for each λ generated during the subgradient iterations. As usual, the most time consuming step is $PTP(L^s)$ solving, but we have demonstrated its $O(n^2)$ complexity in the case when L^s is a union of independent sheaves and single links. More details concerning the actual implementation are given below.

Implementation issues

In order to apply the subgradient optimization technique to solve $PTP(L)$ one need i) to find a suitable partitioning of L into subgraphs with a special structure, and ii) to tune appropriately the parameters of the algorithm, used for finding v_{csd}.

Problem decomposition

The aim of cost-splitting techniques is to decompose a problem into subproblems of reasonable complexity. In our case, each subproblem should be a set of independent sheaves and single links. Several such decompositions exist, but the choice of a particular decomposition may impact a lot on performance: with a lot of subproblems, each one is simpler to solve, but it takes more subgradient iterations to reach a complete agreement between subproblems. Experimentally, the following decomposition appears to be suitable and is easy to obtain.

First all $(i, i+1)$ links ($i \in [1, m]$) are put in L^1 subproblem (if one of them, doesn't exist, it is added with a corresponding null cost matrix). All vertex costs c_i, $i \in [1, m]$ are affected to L^1 subproblem. Thus, all other subproblems have null vertex cost.

Then, given K a set of links we note by K^j a subset of K with right end of links in $[1, j]$. We select the subset of K which includes all the compatible independent sheaves built from right to left. That is, if $K^j \neq \emptyset$, let $s = max\{i|\exists(r, i) \in K^j\}$, then let $s' = max\{r|(r, s) \in K^j\}$ and $K^j_{rms} = \{(s', r) \in K^j\}$. $K^j_{(s',s)}$ is the right most sheaf subset (see Figure (1.7)). We can now

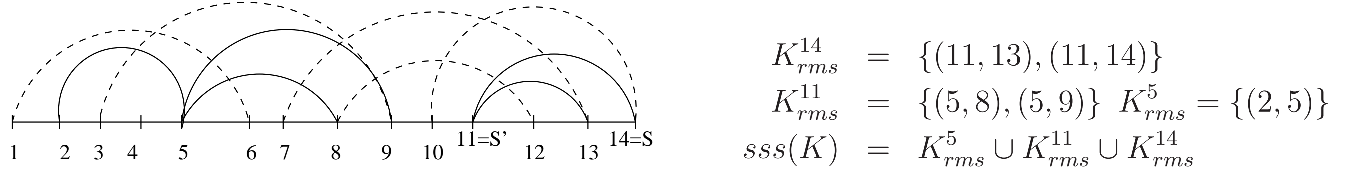


Figure 1.7: The right most sheaf of K

define formally $sss(K)$ the subset of sheaves: $sss(K) = \emptyset$ if $K = \emptyset$ else $sss(K^{s'}) \cup K^j_{(s',s)}$ with $j = max\{k|\exists(i, k) \in K\}$. The partition P is built applying iteratively sss. While $P = (L^1 \cup \ldots L^i) \neq L$, define $L^{i+1} = sss(L \backslash (L^1 \cup \ldots L^i))$ and add L^{i+1} in $\mathcal{P}$.

Subgradient algorithm

The subgradient ascend is an iterative search procedure that is used to maximize a concave function (for a comprehensive introduction, see [23]). In our case this is the function

$$v_{csd}(\lambda) = \min_y\{\sum_{s=1}^{t}(\sum_{i=1}^{m} c_i^s(\lambda) y_i^s + \sum_{(i,k)\in L^s} d_{ik} z_{ik})\} \tag{1.42}$$

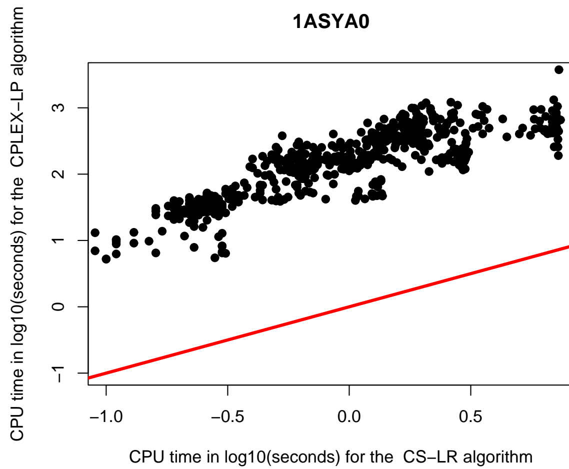
$$= \min\{\sum_{i=1}^{m} c_i y_i + \sum_{(i,k)\in L} d_{ik} z_{ik} + \sum_{s=2}^{t} \sum_{i\in I^{1s}} \lambda^{is}(y_i^1 - y_i^s)\} \tag{1.43}$$

subject to maximize in (1.41). This function is obtained from (1.36) by relaxation of the complicated constraints (1.37). The sets I^{1s} are derived from the actual realization of the partition of L and by elimination from (1.37) all i not common for subproblems L^1 and L^s. If $I^s, s = 2, \ldots, t$ be the set of layers covered by L^s then $I^{1s} = I^1 \bigcap I^s$. Instead (1.37) to hold for each $i \in \{1, \ldots, m\}$ one can achieve the goals by taking $I^{1s}, s = 2, \ldots, t$ as inclusive sets for the respective i. Thus (1.41) is function of $\lambda = (\lambda^2, \ldots, \lambda^t)$ with $\lambda^s \in R^{n|I^{1s}|}$. From Lagrangian duality theory the added term $\sum_{s=2}^{t} \sum_{i\in I^{1s}} \lambda^{is}(y_i^1 - y_i^s)$ is used for approaching the optimal λ^* from the current one by taking a small step along the subgradient. In this case $(\bar{y}_i^1 - \bar{y}_i^s), i \in L^{1s}$(for each i this is a n-vector with only two non-zero coefficients equal to 1 and -1 resp.), with $\bar{y}^s$ being an optimal solution to $PTP(c^s(\lambda), L^s)$ is the part of the subgradient, corresponding to λ^s. At each iteration, $\bar{y}^1$ provides a feasible solution that is used to compute an upper bound u_b for the optimal value; a lower bound l_b is given by the highest found value of the Lagrangian. In our experiments the step length: θ_i in iteration i is controlled by $\theta_i = 1.4(u_b^i - l_b^i)\rho^i \rho_0/\text{nbm}$ where $\rho^{500} = 0.001$, ρ_0 is an initial guess for step length, and nbm is the number of violated relaxed constraints (the length of the subgradient).

1.2.6 Experimental results

The numerical results presented in this section were obtained on an Intel(R) Xeon(TM) CPU 2.4 GHz, 2 GB RAM, RedHat 9 Linux. The behavior of the algorithm was tested by computing the same distributions as given in [44] (for the purpose of comparison), plus few extra-large instances based on real-life data generated by FROST (Fold Recognition Oriented Search Tool) software [49]. The MIP models were solved using CPLEX 7.1 solver [47].

In our first computational experiment we focus on computing score distributions phase and we study the quality of the approximated solutions given by three PTP algorithms. Five distributions are associated to any 3D template in the FROST database. They are computed by threading the template with sets of non related protein sequences having length respectively equal to: -30%, -15%, 0%, +15%, +30% of the template length. Any of these sets contains approximately 200 sequences.



Plot of time in seconds with CS-LR algorithm on the x-axis and the LP algorithm from [43] on the y-axis. Both algorithms compute approximated solutions for 962 threading instances associated to the template 1ASYA0 from the FROST database. The linear curve in the plot is the line $y = x$. What is observed is a significant performance gap between the algorithms. For example in a point $(x, y) = (0.5, 3)$ CS-LR is $10^{2.5}$ times faster than LP relaxation.

Figure 1.8: Cost-Splitting Lagrangian Relaxation versus LP Relaxation

Hence, computing a score distribution in the FROST database requires solving approximately 1000 sequence-to-template alignments. Only two values will be finally used: these are the score values obtained at the 1st and at the 3rd quartiles of the distribution (denoted respectively by q_{25} and q_{75}). FROST uses the following scheme: the raw score (RS) (i.e. the score obtained when a given query is aligned with the template) is normalized according to the formula $NS = \frac{q_{75}-RS}{q_{75}-q_{25}}$. Only the value NS (called normalized score) is used to evaluate the relevance of the computed raw score to the considered distribution.

We conducted the following experiment. For the purpose of this section we chose a set of 12 non-trivial templates. 60 distributions are associated to them. We first computed these distributions using an exact algorithm for solving the underlying PTP problem. The same distributions have been afterwords computed using the approximated solutions obtained by any of the three algorithms here considered. By approximated solution we mean respectively the following: i) for a MIP model this is the solution given by the LP relaxation; ii) for SB-LR (Stefan Balev's Lagrangian Relaxation) algorithm this is the solution obtained for 500 iterations (the upper bound

used in [44]). Any exit with less than 500 iterations is a sign that the exact value has been found; iii) for the Cost-Splitting Lagrangian Relaxation algorithm (CS-LR) this is the solution obtained either for 300 iterations or when the relative error between upper and lower bound is less than 0.001.

We use the MYZ integer programming model introduced in [43]. It has been proved faster than the MIP model used in the package RAPTOR [37] which was well ranked among all non-meta servers in CAFASP3 (Third Critical Assessment of Fully Automated Structure Prediction) and in CASP6 (Sixth Critical Assessment of Structure Prediction). Because of time limit we present here the results from 10 distributions only. Concerning the 1st quartile the relative error between the exact and approximated solution is 3×10^{-3} in two cases over all 2000 instances and less than 10^{-6} for all other cases. Concerning the 3rd quartile, the relative error is 10^{-3} in two cases and less than 10^{-6} for all other cases.

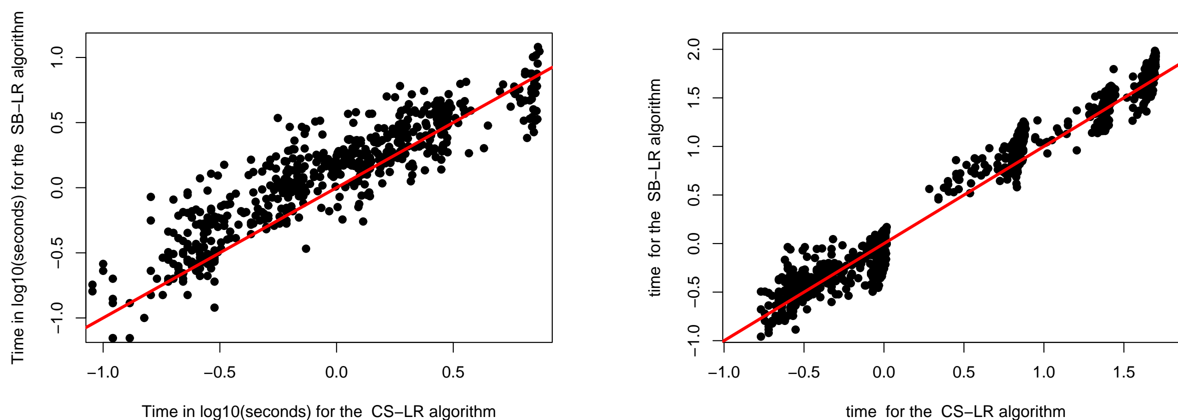


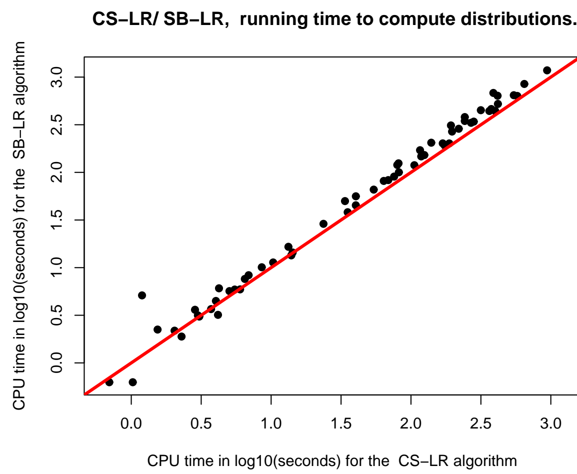
Figure 1.9: Plot of time in seconds with CS-LR (Cost-Splitting Lagrangian Relaxation) algorithm on the x-axis versus SB-LR (Stefan Balev's Lagrangian Relaxation) algorithm [44] on the y-axis concerning score distributions of two templates. Both the x-axis and y-axis are in logarithmic scales. The linear curve in the plot is the line $y = x$. **Left:** The template 1ASYA (the one referenced in [44]) has been threaded with 962 sequences. **Right:** 1ALO_0 is one of the templates yielding the biggest problem instances when aligned with the 704 sequences associated to it in the database. We observe that although CS-LR is often faster than SB-LR, in general the performance of both algorithms is very close.

All 12125 alignments for the set of 60 templates have been computed by the other two algorithms. Concerning the 1st quartile, the exact and approximated solution are equal for all cases for both (SB-LR and CS-LR) algorithms. Concerning the 3rd quartile and in case of SB-LR algorithm the exact solution equals the approximated one in all but two cases in which the relative error is respectively 10^{-3} and 10^{-5}. In the same quartile and in case of CS-LR algorithm the exact solution equals the approximated one in 12119 instances and the relative error is 7×10^{-4} in only 6 cases.

Obviously, this loss of precision (due to computing the distribution by not always taking the

optimal solution) is negligible and does not degrade the quality of the prediction. We therefore conclude that the approximated solutions given by any of above mentioned algorithm can be successfully used in the score distributions phase.

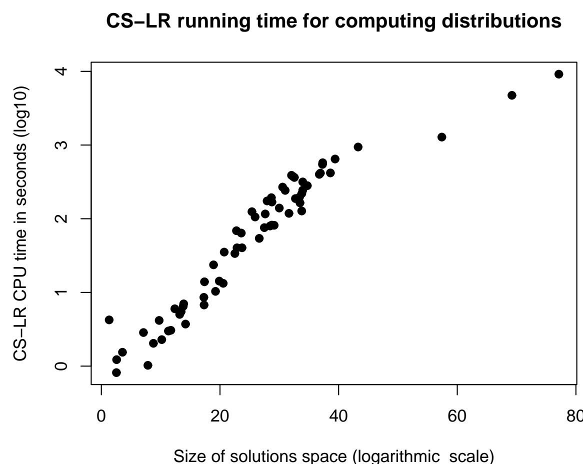
Our second numerical experiment concerns running time comparisons for computing approximated solutions by LP, SB-LR and CS-LR algorithms. The obtained results are summarized on figures 1.8, 1.9 and 1.10. Figure 1.8 clearly shows that CS-LR algorithm significantly outperforms the LP relaxation. Figures 1.9 and 1.10 illustrate that CS-LR is mostly faster than SB-LR algorithm. Time sensitivity with respect to the size of the problem is given in Fig. 1.10.



Plot of time in seconds with CS-LR algorithm on the x-axis and the SB-LR algorithm on the y-axis. Each point corresponds to the total time needed to compute one distribution determined by approximately 200 alignments of the same size. 61 distributions have been computed which needed solving totally 12125 alignments. Both the x-axis and y-axis are in logarithmic scales. The linear curve in the plot is the line $y = x$. CS-LR is consistently faster than SB-LR algorithm.

Figure 1.10: CS-LR versus SB-LR: recapitulation plot concerning 12125 alignments.

Additional experimental results



Each point in this plot corresponds to the total time required by CS-LR algorithm to compute one distribution determined by approximately 200 alignments of the same size. About 60 distributions have been computed which needed solving about 12000 alignments totally. The size of the biggest instance is $O(10^{77})$.

Figure 1.11: Evolution in time as a function of the solutions space size.

1.2.7 Conclusion

The results in this section confirm once more, that integer programming approach is well suited to solve protein threading problem. Here, we proposed a cost-splitting approach, and derived a new Lagrangian dual formulation for this problem. This approach compares favorably with the Lagrangian relaxation proposed in [44]. It allows to solve huge instances[3], with solution space of size up to $O(10^{77})$, within a few minutes.

The results lead us to think that even better performance could be obtained by relaxing additional constraints, relying on the quality of LP bounds. In this manner, the relaxed problem will be easier to solve. This is the subject of our current work.

[3]Solution space size of $O(10^{40})$ corresponds to a MIP model with 4×10^4 constraints and 2×10^6 variables [54].

1.3 Lagrangian approaches for a class of matching problems in computational biology

1.3.1 Preliminaries

Matching is important class of combinatorial optimization problems with many real-life applications. Matching problems involve choosing a subset of edges of a graph subject to degree constraints on the vertices. Many alignment problems arising in computational biology are special cases of matching in bipartite graphs. In these problems the vertices of the graph can be nucleotides of a DNA sequence, aminoacids of a protein sequence or secondary structure elements of a protein structure. Unlike classical matching problems, alignment problems have intrinsic order on the graph vertices and this implies extra constraints on the edges. As an example, Fig. 1.12 shows an alignment of two sequences as a matching in bipartite graph. We can see that the feasible alignments are 1-matchings without crossing edges.

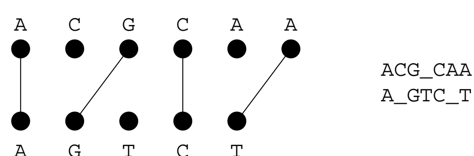
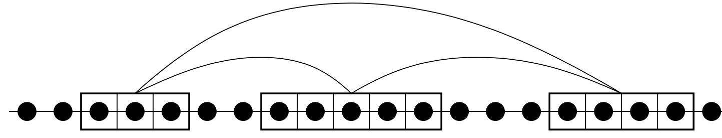


Figure 1.12: Matching interpretation of sequence alignment problem

Here, we reconsider protein threading problem (PTP) as a matching problem in a bipartite graph, recalling that the set of feasible threadings was defined as

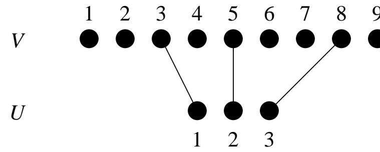
$$\mathcal{T} = \{(r_1, \ldots, r_m) \mid 1 \leq r_1 \leq \cdots \leq r_m \leq n\}.$$



(a)

abs. position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
rel. position block 1	1	2	3	4	5	6	7	8	9											
rel. position block 2				1	2	3	4	5	6	7	8	9								
rel. position block 3									1	2	3	4	5	6	7	8	9			

(b)



(c)

Figure 1.13: (a) Example of alignment of query sequence of length 20 and template containing 3 segments of lengths 3, 5 and 4. (b) Correspondence between absolute and relative block positions. (c) A matching corresponding to the alignment of (a).

The Protein threading problem, now, is a matching problem in a bipartite graph $(U \cup V, U \times V)$, where $U = \{u_1, \ldots, u_m\}$ is the ordered set of blocks and $V = \{v_1, \ldots, v_n\}$ is the ordered set of relative positions. The threading feasibility conditions can be restated in terms of matching in the following way. A matching $M \subseteq U \times V$ is feasible if:

(i) $d(u) = 1$, $u \in U$ (where $d(x)$ is the degree of x). This means that each block is assigned to exactly one position). By the way this implies that the cardinality of each feasible matching is m.

(ii) There are no crossing edges, or more precisely, if $(u_i, v_j) \in M$, $(u_k, v_l) \in M$ and $i < k$, then $j \leq l$. This means that the blocks preserve their order and do not overlap. The last inequality is not strict because of using relative positions.

Note that while (i) is a classical matching constraint, (ii) is specific for the alignment problems and makes them more difficult. Fig. 1.13(c) shows a matching corresponding to a feasible threading.

Proposition 2. *The number of feasible threadings is* $|\mathcal{T}| = \binom{m+n-1}{m}$.

Proof. We can define the relative positions as $r_i = j - \sum_{k=1}^{i-1} l_k + i - 1$. In this case the relative positions of the feasible threadings are related by

$$1 \leq r_1 < \cdots < r_m \leq m + n - 1$$

and a threading is determined by choosing m out of $m + n - 1$ positions. □

One of the possible ways to deal with alignment problems is to try to adapt the existing matching techniques to the new edge constraints of type (ii). Instead of doing this we propose a new graph model and we develop efficient matching algorithms based on this model.

We introduce an *alignment graph* $G = (U \times V, E)$. Each vertex of this graph corresponds to an edge of the matching graph. For simplicity we will denote the vertices by v_{ij}, $i = 1, \ldots, m$, $j = 1, \ldots, n$ and draw them as an $n \times m$ grid (see Fig. 1.14). The vertices v_{ij}, $j = 1, \ldots, n$ were called called ith layer. A layer corresponds to a block and each vertex in a layer corresponds to positioning of this block in the query sequence.

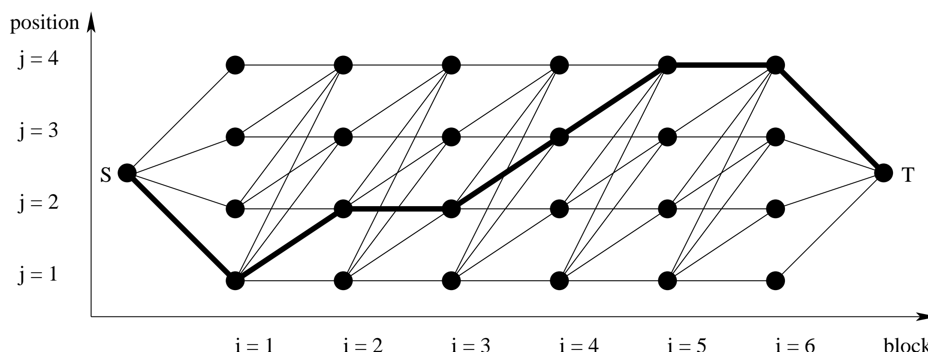


Figure 1.14: Example of alignment graph. The path in thick lines corresponds to the threading in which the positions of the blocks are 1,2,2,3,4,4.

One can connect by edges the pairs of vertices of G which correspond to pairs of noncrossing edges in the matching graph. In this case a feasible threading is an m-clique in G. A similar approach is used in [55, 56]. We introduce only a subset of the above edges, namely the ones that connect vertices from adjacent columns and have the following regular pattern: $E = \{(v_{ij}, v_{i+1,l}) \mid i = 1, \ldots, m-1, 1 \leq j \leq l \leq n\}$. We add two more vertices S and T and edges connecting S to all vertices from the first column and T to all vertices from the last column. Now it is easy to see the one-to-one correspondence between the set of feasible threadings (or matchings) and the set of S-T paths in G. Fig. 1.14 illustrates this correspondence (to help the reader, we redraw this figure once more).

Till now we gave several alternative ways to describe the feasible alignments. Alignment problems in computational biology involve choosing the best of them based on some score function. The simplest score functions associate weights to the edges of the matching graph. For example, this is the case of sequence alignment problems. By introducing alignment graphs similar to the above, classical sequence alignment algorithms, such as Smith-Waterman or Needleman-Wunch, can be viewed as finding shortest S-T paths. When the score functions use structural

information, the problems are more difficult and the shortest path model cannot incorporate this information.

The score functions in PTP evaluate the degree of compatibility between the sequence amino acids and their positions in the template blocks. The interactions (or links) between the template blocks are described by the so-called generalized contact map graph, whose vertices are the blocks and whose edges connect pairs of interacting blocks. Let L be the set of these edges:

$$L = \{(i, k) \mid i < k \text{ and blocks } i \text{ and } k \text{ interact}\}$$

Sometimes we need to distinguish the links between adjacent blocks and the other links. Let $R = \{(i, k) \mid (i, k) \in L,\ k - i > 1\}$ be the set of remote (or non-local) links. The links from $L \setminus R$ are called local links. Without loss of generality we can suppose that all pairs of adjacent blocks interact.

The links between the blocks generate scores which depend on the block positions. In this way a score function of PTP can be presented by the following sets of coefficients

- c_{ij}, $i = 1, \ldots, m$, $j = 1, \ldots, n$, the score of putting block i on position j

- d_{ijkl}, $(i, k) \in L$, $1 \le j \le l \le n$, the score generated by the interaction between blocks i and k when block i is on position j and block k is on position l.

The coefficients c_{ij} are some function (usually sum) of the preferences of each query amino acid placed in block i for occupying its assigned position, as well as the scores of pairwise interactions between amino acids belonging to block i. The coefficients d_{ijkl} include the scores of interactions between pairs of amino acids belonging to blocks i and j. Loops (sequences between adjacent blocks) may also have sequence specific scores, included in the coefficients $d_{i,j,i+1,l}$.

The score of a threading is the sum of the corresponding score coefficients and PTP is the optimization problem of finding the threading of minimum score. If there are no remote links (if $R = \emptyset$) we can put the score coefficients on the vertices and the edges of the alignment graph and PTP is equivalent to the problem of finding the shortest S-T path. In order to take the remote links into account, we add to the alignment graph the edges

$$\{(v_{ij}, v_{kl}) \mid (i, k) \in R,\ 1 \le j \le l \le n\}$$

which we will refer as z-edges.

An S-T path is said to activate the z-edges that have both ends on this path. Each S-T path activates exactly $|R|$ z-edges, one for each link in R. The subgraph induced by the edges of an S-T path and the activated z-edges is called augmented path. Thus PTP is equivalent to finding the shortest augmented path in the alignment graph (see Fig. 1.15).

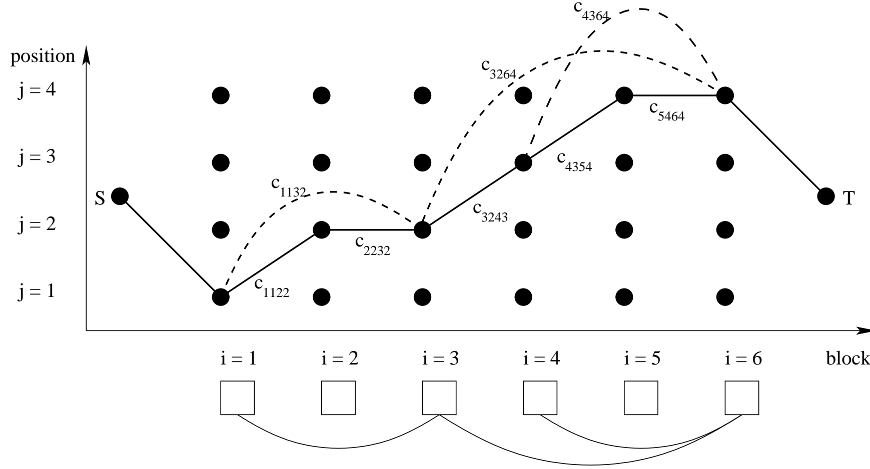


Figure 1.15: Example of augmented path. The generalized contact map graph is given in the bottom. The x arcs of the S-T path are in solid lines. The activated z-arcs are in dashed lines. The length of the augmented path is equal to the score of the threading $(1, 2, 2, 3, 4, 4)$.

As we will see later, the main advantage of this graph is that some simple alignment problems reduce to finding the shortest S-T path in it with some prices associated to the edges and/or vertices. The last problem can be easily solved by a trivial dynamic programming algorithm of complexity $O(mn^2)$. In order to address the general case we need to represent this graph optimisation problem as an integer programming problem.

1.3.2 Integer programming formulation

In order to relate the PTP to a class of matching problems,we restate the definition of the set Y as the polytope defined by the following constraints:

$$\sum_{j=1}^{n} y_{ij} = 1 \qquad i = 1, \ldots, m \tag{1.44}$$

$$\sum_{l=1}^{j} y_{il} - \sum_{l=1}^{j} y_{i+1,l} \geq 0 \qquad i = 1, \ldots, m-1,\ j = 1, \ldots, n-1 \tag{1.45}$$

$$y_{ij} \geq 0 \qquad i = 1, \ldots, m,\ j = 1, \ldots, n \tag{1.46}$$

Constraints (1.44) ensure the feasibility condition (i) and (1.45) are responsible for (ii). That is why $Y \cap B^{mn}$ is exactly the set of feasible threadings.

In order to take into account the interaction costs, we introduce a second set of binary variables z_{ijkl}, $(i, k) \in L$, $1 \leq j \leq l \leq n$. To avoid added notation we will use vector notation for the variables $y_i = (y_{i1}, \ldots y_{in}) \in B^n$ with assigned costs $c_i = (c_{i1}, \ldots c_{in}) \in R^n$ and $z_{ik} = (z_{i1k1}, \ldots, z_{i1kn}, z_{i2k2}, \ldots, z_{i2kn}, \ldots, z_{inkn}) \in B^{\frac{n(n+1)}{2}}$ for $(i, k) \in L$ with assigned costs $d_{ik} = (d_{i1k1}, \ldots, d_{i1kn}, d_{i2k2}, \ldots, d_{i2kn}, \ldots, d_{inkn}) \in R^{\frac{n(n+1)}{2}}$.

Consider the $2n \times \frac{n(n+1)}{2}$ node-edge incidence matrix of the subgraph spanned by two interacting layers i and k. The submatrix A' containing the first n rows (resp. A'' containg the last n rows) corresponds to the layer i (resp. layer k).

Now the protein threading problem can be defined as

$$z_{IP}^{L} = v(PTP(L)) = \min\{\sum_{i=1}^{m} c_i y_i + \sum_{(i,k)\in L} d_{ik} z_{ik}\} \tag{1.47}$$

$$\begin{aligned} \text{subject to: } & y = (y_1, \ldots, y_m) \in Y, & & (1.48) \\ & y_i = A' z_{ik} & (i,k) \in L \quad & (1.49) \\ & y_k = A'' z_{ik} & (i,k) \in L \quad & (1.50) \\ & z_{ik} \in B^{\frac{n(n+1)}{2}} & (i,k) \in L \quad & (1.51) \end{aligned}$$

The shortcut notation $v(.)$ will be used for the optimal objective function value of a subproblem obtained from $PTP(L)$ with some z variables fixed.

1.3.3 Complexity results

In this section we study the structure of the polytope defined by (1.48)-(1.50) and $z_{ik} \in R_{+}^{\frac{n(n+1)}{2}}$, as well as the impact of the set L on the complexity of the algorithms for solving the PTP problem. Throughout this section, vertex costs c_i are assumed to be zero. This assumption is not restrictive because the costs c_{ij} can be added to $d_{i,j,i+1,l}$, $l = j, \ldots, n$. We will consider the costs d_{ik} as $n \times n$ matrices containing the coefficients d_{ijkl} above the main diagonal and arbitrary large numbers below the main diagonal. In order to simplify the descriptions of the algorithms given in this section we introduce the following matrix operations.

Definition 2. Let A and B be two matrices of compatible size. $A \cdot B$ is the matrix product of A and B where the addition operation is replaced by "min" and the multiplication operation is replaced by "+".

Definition 3. Let A and B be two matrices of size $n \times n$. $M = A \otimes B$ is defined by $M(i,j) = \min_{i \le r \le j} A(i,r) + B(i,j)$

PTP is known to be NP-complete in the general case [16]. Below we present four kinds of contact graphs that make PTP polynomially solvable.

Contact graph contains only local edges

As mentioned above, in this case PTP reduces to finding the shortest S-T path in the alignment graph which can be done by $O(mn^2)$ dynamic programming algorithm. An important property of an alignment graph containing only local edges is that it has a tight LP description.

Theorem 3. *The polytope Y is integral, i.e. it has only integer-valued vertices.*

Proof. Let A be the matrix of the coefficients in (1.44)-(1.45) with columns numbered by the indices of the variables. One can prove that A is totaly unimodular (TU) by performing the following sequence of TU preserving transformations.

for $i = 1, \dots, n$
  delete column (i, n) (these are unit columns)
for $i = 1, \dots, m$
  for $j = n-1, \dots, 1$
    pivot on a_{ij} (A is TU iff the matrix obtained by a pivot operation on A is TU
    delete column (i, j) (now this is unit column)

The final matrix is an unit column that is TU. Since all the transformations are TU preserving, A is TU and Y is integral.

One could prove the same assertion by showing that an arbitrary feasible solution to (1.44)-(1.46) is a convex combination of some integer-valued vertices of Y. The best such vertex (in the sense of an objective function) might be a good approximate solution to a problem whose feasible set is an intersection of Y with additional constraints.

Let y is an arbitrary non-integer solution to (1.44)–(1.46). Because of (1.44), (1.45) an unit flow[4] $f = (f_{sj},\ f_{(i,k)(i+1,j)})$, $i = \overline{1, m-1}$, $j = \overline{1, n}$, in G exist s.t.

$$\sum_{k \le j} f_{(i,k)(i+1,j)} = y_{ij},\ i = \overline{1, m-1}, \qquad f_{sj} = y_{1j},\ j = \overline{1, n}.$$

By the well known properties of the network flow polytope, the flow f can be expressed as a convex combination of integer-valued unit flows (paths in G). But each such flow corresponds to an integer-valued y, i.e. $y_{ij} = f_{(i-1,k)(ij)} = 1$. Thus, the convex combination of the paths that gives f is equivalent to a convex combination of the respective vertices of Y that gives y.

The details for efficiently finding of the set of the vertices participating in the convex combination could be easily stressed by this sketch of the prove. □

Contact graph contains no crossing edges

Two links (i_1, k_1) and (i_2, k_2) such that $i_1 < i_2$ are said to be crossing when k_1 is in the open interval (i_2, k_2). The case when the contact graph L contains no crossing edges has been mentioned to be polynomially solvable for the first time in [1]. Here we present a different sketch for $O(mn^3)$ complexity of PTP in this case.

If L contains no crossing edges, then $PTP(L)$ can be recursively divided into independent subproblems. Each of them consists in computing all shortest paths between the vertices of two layers i and k, discarding links that are not included in (i, k). The result of this computation is a distance matrix D_{ik} such that $D_{ik}(j, l)$ is the optimal length between vertices (i, j) and (k, l). Note that for $j > l$, as there is no path in the graph, $D_{ik}(j, l)$ is an arbitrarily large coefficient. Finally, the solution of $PTP(L)$ is the smallest entry of D_{1m}.

[4]The 4 indices i, k, p, j used for arcs labeling follows the convention: tail at vertex (i, k) head at vertex (p, j). Sometimes the brackets will be dropped.

We say that a link $(i, k), i < k$ is included in the interval $[a, b]$ when $[i, k] \subseteq [a, b]$. Let us denote by $L_{(ik)}$ the set of links of L included in $[i, k]$. Then, an algorithm to compute D_{ik} can be sketched as follows:

1. If $L_{(ik)} = \{(i, k)\}$ then the distance matrix is given by

$$D_{ik} = \begin{cases} d_{ik} & \text{if}(i, k) \in L \\ \tilde{0} & \text{otherwise} \end{cases} \tag{1.52}$$

   where $\tilde{0}$ is an upper triangular matrix in the previously defined sense (arbitrary large coefficients below the main diagonal) and having only zeros in its upper part.

2. Otherwise, as $L_{(ik)}$ has no crossing edges, there exists some $s \in [i, k]$ such that any edge of $L_{(ik)}$ except (i, k) is included either in $[i, s]$ or in $[s, k]$. Then

$$D_{ik} = \begin{cases} D_{is} \cdot D_{sk} + d_{ik} & \text{if}(i, k) \in L \\ D_{is} \cdot D_{sk} & \text{otherwise} \end{cases} \tag{1.53}$$

If the contact graph has m vertices, and contains no crossing edges, then the problem is decomposed into $O(m)$ subproblems. For each of them, the computation of the corresponding distance matrix is a $O(n^3)$ procedure (matrix multiplication with $(\min, +)$ operations). Overall complexity is thus $O(mn^3)$. Typically, n is one or two orders of magnitude greater than m, and in practice, this special case is already expensive to solve.

Contact graph is a single star

A set of edges $L_{(i)} = \{(i, k_1), \ldots, (i, k_r)\}, k_1 < k_2 < \ldots k_r$ is called a *star*[5].

Theorem 4. *Let $L_{(i)} = \{(i, k_1), \ldots, (i, k_r)\}$ be a star. Then $D_{ik_r} = (\ldots(d_{ik_1} \otimes d_{ik_2}) \otimes \ldots) \otimes d_{ik_r}$.*

Proof. The proof follows the basic dynamic programming recursion for this particular case: for the star $L = \{(i, k_1), \ldots, (i, k_r)\} = L' \bigcup \{(i, k_r)\}$, we have $v(L : z_{ijk_rl} = 1) = d_{ijk_rl} + \min_{j \leq s \leq l} v(L' : z_{ijk_{r-1}s} = 1)$. □

In order to compute $A \otimes B$, we use the following recursion: let M' be the matrix defined by $M'(i, j) = \min_{i \leq r \leq j} A(i, r)$, then

$$M'(i, j) = \min\{M'(i, j - 1), A(i, j)\}, \text{ for all } j \geq i.$$

Finally $A \otimes B = M' + B$. From this it is clear that $\otimes$ multiplication for $n \times n$ matrices is of complexity $O(n^2)$ and hence the complexity of PTP in this case is $O(rn^2)$.

[5] This definition corresponds to the case when all edges have their left end tied to a common vertex. Star can be symmetrically defined: i.e. all edges have their right end tied to a common vertex. All proofs require minor modification to fit this case.

Contact graph is decomposable

Given a contact graph $L = \{(i_1, k_1), \ldots, (i_r, k_r)\}$, $PTP(L)$ can be decomposed into two independent subproblems when there exists an integer $e \in (1, m)$ such that any edge of L is included either in $[1, e]$, either in $[e, m]$. Let $I = \{i_1, \ldots, i_s\}$ be an ordered set of indices, such that any element of I allows for a decomposition of $PTP(L)$ into two independent subproblems. Suppose additionally that for all $t \leq s - 1$, one is able to compute $D_{i_t i_{t+1}}$. Then we have the following theorem:

Theorem 5. *Let* $p = (p_1, p_2, \ldots, p_n) = D_{i_1 i_2} \cdot D_{i_2 i_3} \cdot \ldots \cdot D_{i_{s-1} i_s} \cdot \overline{p}$*, where* $\overline{p} = (0, 0, \ldots, 0)$*. Then for all* i*,* $p_i = v(PTP(L : y_{1i} = 1))$*, and* $v(PTP(L)) = \min_{1 \leq i \leq n}\{p_i\}$*.*

Proof. Each multiplication by $D_{i_k i_{k+1}}$ in the definition of p is an algebraic restatement of the main step of the algorithm for solving the shortest path problem in a graph without circuits. □

With the notations introduced above, the complexity of $PTP(L)$ for a sequence of such subproblems is $O(sn^2)$ plus the cost of computing matrices $D_{i_t i_{t+1}}$.

From the last two special cases, it can be seen that if the contact graph can be decomposed into independent subsets, and if these subsets are single edges or stars, then there is a $O(srn^2)$ algorithm, where s is the cardinality of the decomposition, and r the maximal cardinality of each subset, that solves the corresponding PTP.

Remark 3. As a corollary from theorem 3 we can easily derive that when L is cross free and does not contain stars, the polytope defined by (1.49)–(1.50) and $z_{ik} \in R_+^{\frac{n(n+1)}{2}}$ is integer.

The threading polytope

Let P_{yz} be the polytope defined by (1.48)–(1.50) and $z_{ik} \in R_+^{\frac{n(n+1)}{2}}$ and let P_{yz}^I be the convex hull of the feasible points of (1.48)-(1.51). We will call P_{yz}^I a threading polytope.

All of the preceeding polynomiality results were derived without any refering to the LP relaxation of (1.47)–(1.51). The reason is that even for a rather simple version of the graph L the polytope P_{yz} is non-integral. We have already seen (indirectly) that if L contains only local links then $P_{yz} = P_{yz}^I$. Recall the one-to-one correspondence between the threadings, defined as points in Y and the paths in graph G. If $L = \{(i, i+1), i = 1, m-1\}$ then P_{yz} is a linear description of a unit flow in G that is an integral polytope. Unfortunately, this happenens to be a necessary condition also.

Theorem 6. *Let* $n \geq 3$ *and* L *contains all local links. Then* $P_{yz}^I = P_{yz}$ *if and only if* $R = \emptyset$*.*

Proof. ($\Rightarrow$) Without loss of generality we can take $R = (1, 3)$, $m = 3$ and $n = 3$. Then the point $A = (y_{11} = y_{12} = y_{21} = y_{22} = 0.5, y_{32} = 0.75,\ y_{33} = 0.25,\ z_{1121} = z_{2132} = z_{1222} = z_{1232} = 0.5, z_{2232} = z_{2233} = z_{1132} = z_{1133} = 0.25) \ \in P_{yz}$ and the only eligible (whose convex hull could possibly contain A) integer-valued vertices of P_{yz} are $B = (y_{11} = y_{21} = y_{32} = z_{1132} = 1)$ and $C = (y_{12} = y_{22} = y_{32} = z_{1232} = 1)$ but A is not in the segment $[B, C]$. The generalization of this proof for arbitrary $m, n \geq 3$ and R is almost straightforward.

($\Leftarrow$) Follows directly from Theorem 3. □

This is a kind of negative result setting a limit to relying on LP solution.

1.3.4 Lagrangian approaches

Consider an integer program

$$z_{IP} = \min\{cx : x \in S\}, \text{where } S = \{x \in Z^n_+ : Ax \leq b\} \tag{1.54}$$

Relaxation and duality are the two main ways of determining z_{IP} and upper bounds for z_{IP}. The linear programming relaxation is obtained by changing the constraint $x \in Z^n_+$ in the definition of S by $x \geq 0$. The Lagrangian relaxation is very convenient for problems where the constraints can be partitioned into a set of "simple" ones and a set of "complicated" ones. Let us assume for example that the complicated constraints are given by $A^1x \leq b^1$, where A^1 is $m \times n$ matrix, while the nice constraints are given by $A^2x \leq b^2$. Then for any $\lambda \in R^m_+$ the problem

$$z_{LR}(\lambda) = \min_{x \in Q}\{cx + \lambda(b^1 - A^1x)\}$$

where $Q = \{x \in Z^n_+ : A^2x \leq b^2\}$ is Lagrangian relaxation of (1.54), i.e. $z_{LR}(\lambda) \leq z_{IP}$ for each $\lambda \geq 0$. The best bound can be obtained by solving the Lagrangian dual $z_{LD} = \max_{\lambda \geq 0} z_{LR}(\lambda)$. It is well known that relations $z_{IP} \geq z_{LD} \geq z_{LP}$ hold.

An even better relaxation, called *cost-splitting*, can be obtained by applying Lagrangian duality to the reformulation of (1.54) given by

$$z_{IP} = \min cx^1 \tag{1.55}$$

$$\text{subject to: } A^1x^1 \leq b^1, \;\; A^2x^2 \leq b^2, \tag{1.56}$$

$$x^1 - x^2 = 0 \tag{1.57}$$

$$x^1 \in Z^n_+, \;\; x^2 \in Z^n_+, \tag{1.58}$$

Taking $x^1 - x^2 = 0$ as the complicated constraint, we obtain the Lagrangian dual of (1.55)-(1.58)

$$z_{CS} = \max_u\{\min c^1x^1 + \min c^2x^2\} \tag{1.59}$$

$$\text{subject to: } A^1x^1 \leq b^1, \;\; A^2x^2 \leq b^2, \tag{1.60}$$

$$x^1 \in Z^n_+, \;\; x^2 \in Z^n_+, \tag{1.61}$$

where $u = c^2, c^1 = c - u$.

The following well known polyhedral characterization of the cost splitting dual will be used later:

Theorem 7 (see [23])**.**

$$z_{CS} = \max\left\{cx : \; \text{conv}\{x \in Z^n_+ : \; A^1x \leq b^1\} \cap \text{conv}\{x \in Z^n_+ : \; A^2x \leq b^2\}\right\}$$

where $\text{conv}\{A\}$ *denotes the convex hull of* A*.*

In both relaxations in order to find z_{LD} or z_{CS} one has to look for the maximum of a concave piecewise linear function. This appeals for using the so called subgradient optimization technique. For the function $z_{LR}(\lambda)$, the vector $s^t = b^1 - A^1x^t$, where x^t is an optimal solution to $\min_Q\{cx + \lambda^t(b^1 - A^1x)\}$, is a subgradient at λ^t. The following subgradient algorithm is an analog of the steepest ascent method of maximizing a function:

- (Initialization): Choose a starting point λ^0, Θ_0 and ρ. Set $t = 0$ and find a subgradient s^t.
- While $s^t \neq 0$ and $t < t_{\max}$ do { $\lambda^{t+1} = \lambda^t + \Theta_t s^t$; $t \leftarrow t + 1$; find s^t}

This algorithm stops either when $s^t = 0$, (in which case λ^t is an optimal solution) or after a fixed number of iterations. We experimented two schemes for selecting $\{\Theta_t\}$:

$$\Theta_t = \Theta_0\rho^t \tag{1.62}$$

$$\Theta_t = \Theta_0\frac{\kappa_t(U_t - L_t)\rho^t}{||s^t||_1} \tag{1.63}$$

where

$0 < \rho < 1$
$\{\kappa_t\}$ is a random sequence whose terms are uniformly chosen in [1, 1.4]
L_t is the best value of $z_{LR}(\lambda)$ up to iteration t
U_t is the best value of any feasible solution found up to iteration t
$||s^t||_1$ is the 1-norm of the subgradient

1.3.5 Lagrangian relaxation

Relying on complexity results from section 1.3.3, we show now how to apply Lagrangian relaxation taking as complicating constraints (1.50). Recall that these constraints insure that the y-variables and the z-variables select the same position of block k. Associating Lagrangian multipliers λ_{ik} to the relaxed constraints we obtain

$$z_{LR}(\lambda) = \min_{y,z}\left\{\sum_{i=1}^{m} c_i(\lambda)y_i + \sum_{(i,k)\in L} d_{ik}(\lambda)z_{ik}\right\}$$

where

$$c_i(\lambda) = c_i + \sum_{(k,i)\in L}\lambda_{ki}, \quad d_{ik}(\lambda) = \sum_{(i,k)\in L}(d_{ik} - \lambda_{ik}A'').$$

Consider this relaxation for a fixed λ. Suppose that a block i is on position j in the optimal solution. Then the optimal values of the variables z_{ijkl} can be found using the method desctibed in section 1.3.3. In this way the relaxed problem decomposes to a set of independent subproblems. Each subproblem has a star as a contact graph. After solving all the subproblems, we can update

the costs $c_i(\lambda)$ with the contribution of the star with root i and find the shortest S-T path in the alignment graph.

Note that for each λ the solution defined by the y-variables is feasible to the original problem. In this way at each iteration of the subgradient optimisation we have an heuristic solution. At the end of the optimization we have both lower and upper bounds on the optimal objective value.

Symmetrically, we can relax the left end of each link or even relax the left end of one part of the links and the right end of the rest. The last is the approach used in [5]. The same paper describes a branch-and-bound algorithm using this Lagrangian relaxation instead of the LP relaxation.

1.3.6 Cost splitting

In order to apply the results from the previous sections, we need to find a suitable partition of L into $L^1 \bigcup L^2 ... \bigcup L^t$ where each L^s induces an easy solvable $PTP(L^s)$, and to use the cost-splitting variant of the Lagrangian duality. Now we can restate (1.47)-(1.51) equivalently as:

$$z_{IP}^L = \min \left\{ \sum_{s=1}^{t} (\sum_{i=1}^{m} c_i^s y_i^s + \sum_{(i,k) \in L^s} d_{ik} z_{ik}) \right\} \tag{1.64}$$

$$\begin{aligned} \text{subject to: } & y_i^1 = y_i^s, && s = 2, t && && (1.65) \\ & y^s = (y_1^s, ..y_m^s) \in Y, && s = 1, \dots, t && && (1.66) \\ & y_i^s = A_i z_{ik}, \;\; y_k^s = A_k z_{ik} && s = 1, \dots, t && (i,k) \in L^s && (1.67) \\ & z_{ik} \in B^{\frac{n(n+1)}{2}} && s = 1, \dots, t && (i,k) \in L^s && (1.68) \end{aligned}$$

Taking (1.65) as the complicating constraints, we obtain the Lagrangian dual of $PTP(L)$:

$$z_{CS} = \max_{\lambda} \min_{y} \sum_{s=1}^{t} (\sum_{i=1}^{m} c_i^s(\lambda) y_i^s + \sum_{(i,k) \in L^s} d_{ik} z_{ik}) = \max_{\lambda} \sum_{s=1}^{t} z_{IP}^{L^s}(\lambda) \tag{1.69}$$

subject to (1.66), (1.67) and (1.68).

The Lagrangian multipliers λ^s are associated with the equations (1.65) and $c_i^1(\lambda) = c_i^1 + \sum_{s=2}^{t} \lambda^s$, $c_i^s(\lambda) = c_i^s - \lambda^s$, $s = 2, \dots, t$. The coefficients c_i^s are arbitrary (but fixed) decomposition (cost-split) of the coefficients c_i, i.e. given by $c_i^s = p_s c_i$ with $\sum p_s = 1$.

From the Lagrangian duality theory it follows that $z_{LP} \leq z_{CS} \leq z_{IP}$. However choosing the decomposition remains a delicate issue. A tradeoff has to be found between tightness of the bound and complexity of the dual. At one extreme, when decomposing the interaction graph into cross-free sets, the dual problem is of $O(mn^3)$ complexity. This makes this approach hopeless for practical situations. At the other extreme, each set in the decomposition could contain a single edge. This is a very favorable situation for complexity matters, but it turns out that in this case, the cost-splitting dual boils down to LP bound:

Theorem 8. *If $t = |L|$ then $z_{CS} = z_{LP}$.*

Proof. From Th. 7, we have

$$z_{CS} = \max \left\{ cy + dz : \bigcap_{(i,k)\in L} \text{conv}\{y, z \in Z_+^n : y_i = A_i^k z_{ik} \wedge y_k = A_k^i z_{ik}\} \right\}$$

However, as underlined in Rem. 3, the set

$$\{y, z \in R_+^n : y_i = A_i^k z_{ik} \wedge y_k = A_k^i z_{ik})\}$$

only has integer extremal points, which amounts to say that

$$\{y, z \in R_+^n : y_i = A_i^k z_{ik}\} = \text{conv}\{y, z \in Z_+^n : y_i = A_i^k z_{ik} \wedge y_k = A_k^i z_{ik}\}$$

The result follows:

$$z_{CS} = \max \left\{ cy + dz : \bigcap_{(i,k)\in L} \{y, z \in R_+^n : y_i = A_i^k z_{ik} \wedge y_k = A_k^i z_{ik}\} \right\} = z_{LP}$$

□

By applying the subgradient optimization technique ([23]) in order to obtain z_{CS}, one need to solve t problems $v_{IP}^{L^s}(\lambda)$ for each λ generated during the subgradient iterations. As usual, the most time consuming step is $PTP(L^s)$ solving, but we have demonstrated its $O(n^2)$ complexity in the case when L^s is a union of independent stars.

1.3.7 Experimental results

In this section we present three kinds of experiments. First, in subsection 1.3.7, we show that the branch-and-bound algorithm based on the Lagrangian relaxation from section 1.3.5 (BB_LR) can be successfully used for solving exactly huge PTP instances. In subsection 1.3.7, we study the impact of the approximated solutions given by different PTP solvers on the quality of the prediction. Lastly, in subsection 1.3.7 we experimentally compare the two relaxations proposed in this paper and show that they have similar performances.

In order to evaluate the performance of our algorithm and to test it on real problems, we integrated it in the structure prediction tool FROST [21, 22]. FROST (Fold Recognition-Oriented Search Tool) is intended to assess the reliability of fold assignments to a given protein sequence. In our experiments we used its the structure database, containing about 1200 structure templates, as well as its score function. FROST uses a specific procedure to normalize the alignment score and to evaluate its significance. As the scores are highly dependent on sequence lengths, for each template of the database this procedure selects 5 groups of non homologous sequences corresponding to -30%, -15%, 0%, +15% and +30% of the template length. Each group contains about 200 sequences of equal length. Each of the about 1000 sequences is aligned to the template. This procedure involves about 1,200,000 alignments and is extremely computationally expensive [57]. The values of the score distribution function F in the points 0.25 and 0.75 are approximated by this empirical data. When a "real" query is threaded to this template, the raw alignment score S is replaced by the *normalized distance* $NS = \frac{F(.75)-S}{F(.75)-F(.25)}$. Only the value NS is used to evaluate the relevance of the computed raw score to the considered distribution.

Solving PTP exactly

To test the efficiency of our algorithm we used the data from 9,136 threadings made in order to compute the distributions of 10 templates. Figure 1.16 presents the running times for these alignments. The optimal threading was found in less than one minute for all but 34 instances. For 32 of them the optimum was found in less than 4 minutes and only for two instances the optimum was not found in one hour. However, for these two instances the algorithm produced in one minute a suboptimal solution with a proved objective gap less than 0.1%.

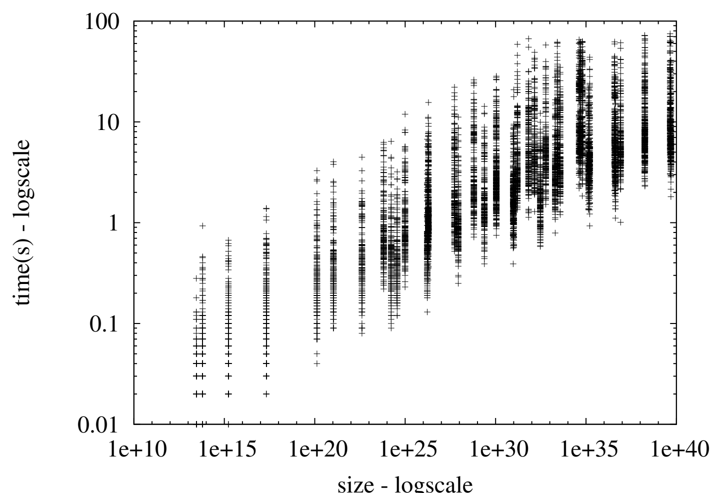


Figure 1.16: Running times of 9,136 threading instances as a function of the search space size. The experiment is made on 1.8 GHz Pentium PC with 512 MB RAM

It is interesting to note that for 79% of the instances the optimal solution was found in the root of the branch-and-bound tree. This means that the Lagrangian relaxation produces a solution which is feasible for the original problem. The same phenomenon was observed in [37, 4] where integer programming models are solved by linear relaxation. However, the dedicated algorithm based of the Lagrangian relaxation from section 1.3.5 is much faster than a general purpose solver using the linear relaxation. For comparison, solving instances of size of order 10^{38} by CPLEX of ILOG solver reported in [4] takes more than one hour on a faster than our computer, while instances of that size were solved by LR algorithm in about 15 seconds.

The use of BB_LR made possible to compute the exact score distributions of all templates from the FROST database for the first time [57]. An experiment on about 200 query proteins of known structure shows that using the new algorithm improves not only the running time of the method, but also its quality. When using the exact distributions, the sensitivity of FROST (measured as the percentage of correctly classified queries) is increased by 7%. Moreover, the quality of the alignments produced by our algorithm (measured as the difference with the VAST alignments) is also about 5% better compared to the quality of the alignments produced by the heuristic algorithm.

Impact of the approximated solution on the quality of the prediction

We compared BB_LR to two other algorithms used by FROST – a steepest-descent heuristic (H) and an implementation of the branch-and-bound algorithm from [18] (B). The comparison was made over 952 instances (the sequences threaded to the template 1ASYA when computing its score distribution). Each of the three algorithms was executed with a timeout of 1 minute per instance. We compare the best solutions produced during this period. The results of this comparison are summarized in Table 1.8. For the smallest instances (the first line of the table) the performance of the three algorithms is similar, but for instances of greater size our algorithm clearly outperforms the other two. It was timed out only for two instances, while B was timed out for all instances. L finds the optimal solution for all but 2 instances, while B finds it for no instance. The algorithm B cannot find the optimal solution for any instance from the fourth and fifth lines of the table even when the timeout is set to 2 hours. The percentage of the optima found by H degenerates when the size of the problem increases. Note however that H is a heuristic algorithm which produces solutions without proof of optimality. Table 1.9 shows the distributions computed by the three algorithms. The distributions produced by H and especially by B are shifted to the right with respect to the real distribution computed by L. This means that for example a query of length 638AA and score 110 will be considered as significantly similar to the template according to the results provided by B, while in fact this score is in the middle of the score distribution.

Table 1.8: Comparison between three algorithms: branch-and-bound using Lagrangian relaxation (L), heuristic steepest-descent algorithm (H), and branch-and-bound of Lathrop and Smith (B). The results in each row are average of about 200 instances.

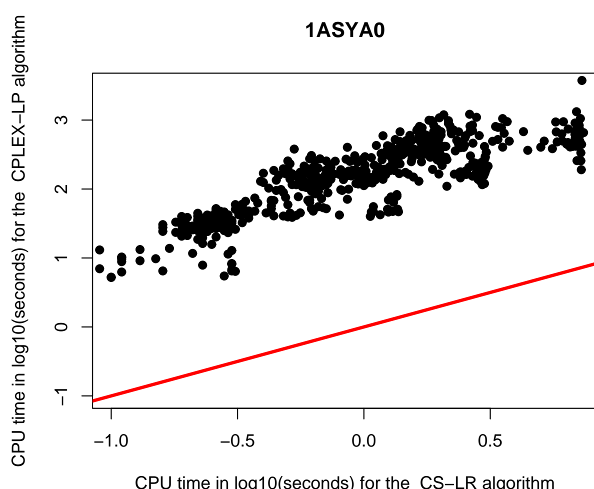
query length	m	n	$\|T\|$	average time(s)			opt(%)		
				L	H	B	L	H	B
342	26	4	3.65e03	0.0	0.1	0.0	100	99	100
416	26	78	1.69e24	0.6	43.6	60.0	100	63	0
490	26	152	1.01e31	2.6	53.8	60.0	100	45	0
564	26	226	1.60e35	6.4	56.6	60.0	100	40	0
638	26	300	1.81e38	12.7	59.0	60.0	99	31	0

We conducted the following experiment. For the purpose of this section we chose a set of 12 non-trivial templates. 60 distributions are associated to them. We first computed these distributions using an exact algorithm for solving the underlying PTP problem. The same distributions have been afterwords computed using the approximated solutions obtained by any of the three algorithms here considered. By approximated solution we mean respectively the following: i) for a MIP model this is the solution given by the LP relaxation; ii) for the Lagrangian Relaxation (LR) algorithm this is the solution obtained for 500 iterations (the upper bound used in [5]). Any exit with less than 500 iterations is a sign that the exact value has been found; iii) for the Cost-Splitting algorithm (CS) this is the solution obtained either for 300 iterations or when the relative

Table 1.9: Distributions produced by the three algorithms.

query	distribution (L)			distribution (H)			distribution (B)		
length	$F(.25)$	$F(.50)$	$F(.75)$	$F(.25)$	$F(.50)$	$F(.75)$	$F(.25)$	$F(.50)$	$F(.75)$
342	790.5	832.5	877.6	790.5	832.6	877.6	790.5	832.5	877.6
416	296.4	343.3	389.5	299.2	345.4	391.7	355.2	405.5	457.7
490	180.6	215.2	260.4	184.5	219.7	263.4	237.5	290.4	333.0
564	122.6	150.5	181.5	126.3	157.5	187.9	183.3	239.3	283.4
638	77.1	109.1	142.7	87.6	118.5	150.0	154.5	197.0	244.6

error between upper and lower bound is less than 0.001.



Plot of time in seconds with CS algorithm on the x-axis and the LP algorithm from [4] on the y-axis. Both algorithms compute approximated solutions for 962 threading instances associated to the template 1ASYA0 from the FROST database. The linear curve in the plot is the line $y = x$. What is observed is a significant performance gap between the algorithms. For example in a point $(x, y) = (0.5, 3)$ CS is $10^{2.5}$ times faster than LP relaxation. These results were obtained on an Intel(R) Xeon(TM) CPU 2.4 GHz, 2 GB RAM, RedHat 9 Linux. The MIP models were solved using CPLEX 7.1 solver.

Figure 1.17: Cost-Splitting Relaxation versus LP Relaxation

We use the MYZ integer programming model introduced in [4]. It has been proved faster than the MIP model used in the package RAPTOR [37] which was well ranked among all non-meta servers in CAFASP3 (Third Critical Assessment of Fully Automated Structure Prediction) and in CASP6 (Sixth Critical Assessment of Structure Prediction). Because of time limit we present here the results from 10 distributions only[6]. Concerning the 1st quartile the relative error between the exact and approximated solution is $3 \times 1_1 30^{-3}$ in two cases over all 2000 instances and less than 10^{-6} for all other cases. Concerning the 3rd quartile, the relative error is 10^{-3} in two cases and less than 10^{-6} for all other cases.

All 12125 alignments for the set of 60 templates have been computed by the other two algorithms. Concerning the 1st quartile, the exact and approximated solution are equal for all cases

[6]More data will be solved and provided for the final version.

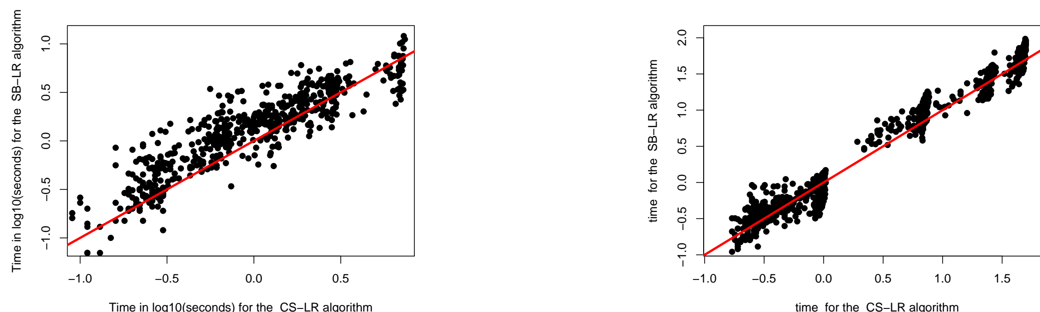


Figure 1.18: Plot of time in seconds with CS (Cost-Splitting Relaxation) algorithm on the x-axis versus LR (Lagrangian Relaxation) algorithm [5] on the y-axis concerning score distributions of two templates. Both the x-axis and y-axis are in logarithmic scales. The linear curve in the plot is the line $y = x$. **Left:** The template 1ASYA (the one referenced in [5]) has been threaded with 962 sequences. **Right:** 1ALO_0 is one of the templates yielding the biggest problem instances when aligned with the 704 sequences associated to it in the database. We observe that although CS is often faster than LR, in general the performance of both algorithms is very close.

for both (LR and CS) algorithms. Concerning the 3rd quartile and in case of LR algorithm the exact solution equals the approximated one in all but two cases in which the relative error is respectively 10^{-3} and 10^{-5}. In the same quartile and in case of CS algorithm the exact solution equals the approximated one in 12119 instances and the relative error is 7×10^{-4} in only 6 cases.

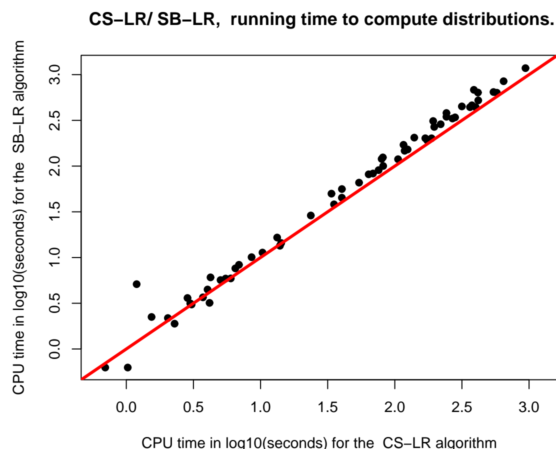
Obviously, this loss of precision (due to computing the distribution by not always taking the optimal solution) is negligible and does not degrade the quality of the prediction. We therefore conclude that the approximated solutions given by any of above mentioned algorithms can be successfully used in the score distributions phase.

Cost splitting versus Linear Programming and Lagrangian relaxations

Our third numerical experiment concerns running time comparisons for computing approximated solutions by LP, LR and CS algorithms. The obtained results are summarized on figures 1.17, 1.18 and 1.19. Figure 1.17 clearly shows that CS algorithm significantly outperforms the LP relaxation. Figures 1.18 and 1.19 compare CS with LR algorithm and illustrate that they give close running times (CS being slightly faster than LR). Time sensitivity with respect to the size of the problem is given in Fig. 1.19.

1.3.8 Conclusion

The results presented in this paper confirm once more that integer programming approach is well suited to solve the protein threading problem. Even if the possibilities of general purpose solvers using linear programming relaxation are limited to instances of relatively small size, one can use the specific properties of the problem and develop efficient special purpose solvers. After



Plot of time in seconds with CS algorithm on the x-axis and the LR algorithm on the y-axis. Each point corresponds to the total time needed to compute one distribution determined by approximately 200 alignments of the same size. 61 distributions have been computed which needed solving totally 12125 alignments. Both the x-axis and y-axis are in logarithmic scales. The linear curve in the plot is the line $y = x$. CS is consistently faster than the LR algorithm.

Figure 1.19: CS versus LR: recapitulation plot concerning 12125 alignments.

studying these properties we propose two Lagrangian approaches, Lagrangian relaxation and cost splitting. These approaches are more powerful than the general integer programming and allow to solve huge instances[7], with solution space of size up to 10^{77}, within a few minutes.

The results lead us to think that even better performance could be obtained by relaxing additional constraints, relying on the quality of LP bounds. In this manner, the relaxed problem will be easier to solve. This is the subject of our current work.

This paper deals with the problem of global alignment of protein sequence and structure template. But the methods presented here can be adapted to other classes of matching problems arising in computational biology. Examples of such classes are semi-global alignment, where the structure is aligned to a part of the sequence (the case of multi-domain proteins), or local alignment, where a part of the structure is aligned to a part of the sequence. Problems of structure-structure comparison, for example contact map overlap, are also matching problems that can be treated with similar techniques. Solving these problems by Lagrangian approaches is the subject of the next section.

[7]Solution space size of 10^{40} corresponds to a MIP model with 4×10^4 constraints and 2×10^6 variables [54].

1.4 A Novel Algorithm for Finding Maximum Common Ordered Subgraph

1.4.1 Introduction

It is a fundamental axiom of biology that the 3-dimensional structure of a protein has a crucial influence on its function- two proteins that are similar in their 3-dimensional structure will likely have similar functions. Comparing two protein structures for similarity is therefor a crucial task and has been extensively investigated [67].
Since it is not clear what quantitative measure to use for comparing protein structures, a multitude of measures have been proposed. Each measure aims in capturing the intuitive notion of similarity. We study the *contact-map-overlap* (CMO) measure, first proposed in [68]. This measure has been found to be very useful for measuring protein similarity - it is robust, takes partial matching into account, translation invariant and captures the intuitive notion of similarity very well for details. Thus the problem of designing efficient algorithms that guarantee the CMO quality is an important one that has eluded researchers so far.

Here, we present an algorithm for exact solving the CMO problem (the formal definition is given below). The CMO is just one of the scoring schemes used for comparison of protein structures. The protein's primary sequence is usually though-of as composed of residues. Under specific physiological conditions, the linear arrangement of residues will fold and adopt a complex three dimensional shape, called native state (or tertiary structure) of the protein. In its native state, residues that are far away along the linear arrangement may come into proximity in three dimensional space. The proximity relation between residues in a protein is captured by a mathematical construct called a contact map. Formally, a map is specified by a $0-1$ symmetric $n \times n$ matrix C whose 1-elements correspond to pairs of amino acids on 3D contact, i.e. $c_{ij} = 1$ if the Euclidean distance of two heavy atoms(or the minimum distance between any two atoms belonging to those residues) from the i-th and the j-th amino acid of a protein is smaller than a given threshold in the protein native fold. In the pairwise comparison one tries to evaluate the similarity in the 3D- folds of two proteins by determining the maximum overlap (also called alignment) of contacts map (See Fig 1.20, where the two contact maps are shown in red and the matching in blue).

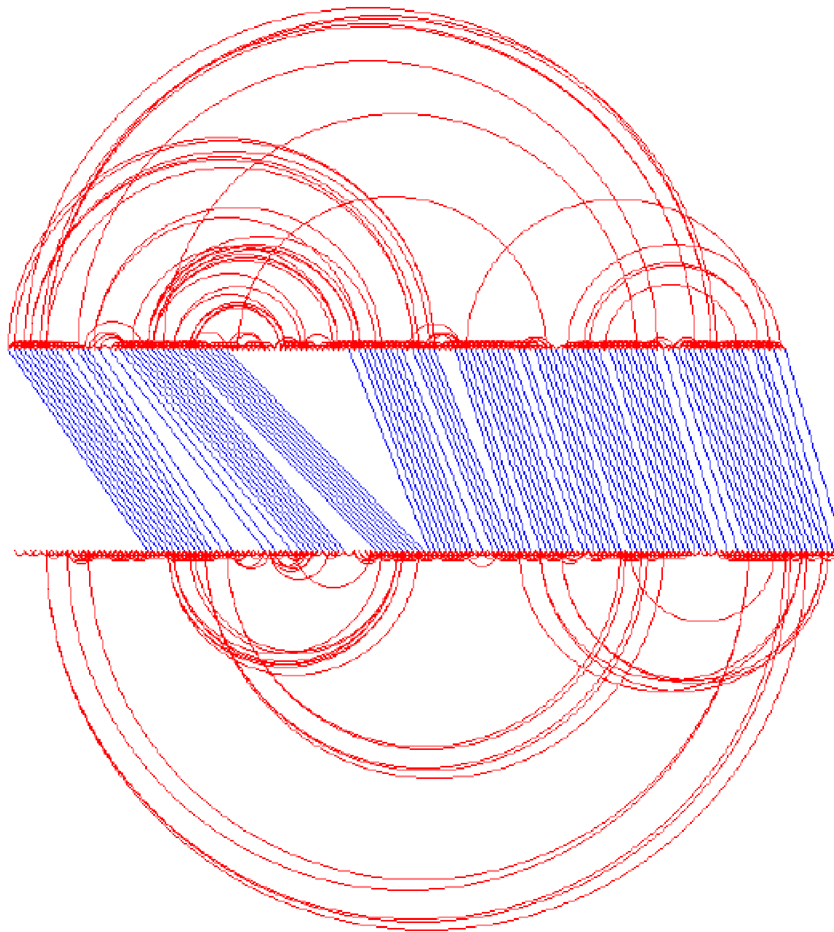


Figure 1.20: Contact map overlap instance

Our interest in the comparison of the $3D$ structures of protein molecules based on such maximal common sub-graph detection is provoked by the apparent similarity of the resulting optimization problem with the one derived by another challenging problem- the protein folding, approached by the protein threading technique. For the later in [4] we have presented a methodology, based on s.c. non-crossing matching in bipartite graphs, culminated in highly efficient algorithms for solving the PTP by using the Lagrangian duality [5, 12, 58]. In the same time (independently) in [46] a Lagrangian approach have been reported successful for the CMO problem. Please refer to this paper for the history of this problem, the various techniques for solving it, and at the end for the triumph of the algorithm proposed there. So the challenge is : could one create a competitive algorithm based on the above-mentioned PTP platform ? Below, we concentrate on the description of such an algorithm and answer affirmatively to this question. The counterpart of the CMO problem in the graph theory is the well known maximum common subgraph problem

(MCS) [69]. The bad news for the later is its APX-hardness (see *A compendium of NP optimization problems* available at `http://www.nada.kth.se/~viggo/problemlist/`). The only difference between the above defined CMO and MCS is that the izomorphism used for the MCS is not restricted to the non-crossing matchings only. Nevertheless the CMO is also known [65] to be NP-hard. Some authors [66] use the adjectives sequential, if in doing protein structure alignment the sequential order (the vertices of the graphs are ordered by a linear sequence and the bijection is order-preserving) is enforced, and non-sequential (equivalent to MCS) in the other case.

1.4.2 The mathematical model

We are going to present the CMO problem as a matching problem in a bipartite graph, which in turn will be posed as a longest augmented path problem in a structured graph. Toward this end we need to introduce few notations as follows. The contacts maps of two proteins P1 and P2 are given by graphs $G_m = (V_m, E_m)$ with $V_m = \{1, 2, \ldots, n_m\}$ for $m = 1, 2$. The vertices V_m are better seen as ordered points on a line and correspond to the residues of the proteins. The arcs (i, j) correspond to the contacts. The right and left neighborings of node i are elements of the sets $\delta_m^+(i) = \{j | j > i, (i, j) \in E_m\}$, $\delta_m^-(i) = \{j | j < i, (j, i) \in E_m\}$. Let $i \in V_1$ be matched with $k \in V_2$ and $j \in V_1$ be matched with $l \in V_2$. We will call a matching *non-crossing*, if $i < j$ implies $k < l$. A feasible alignment of two proteins P_1 and P_2 is given by a non-crossing matching in the complete bipartite graph B with a vertex set $V_1 \cup V_2$.

Let the weight w_{ikjl} of the matching couple $(i, k)(j, l)$ be set as follows

$$w_{ikjl} = \begin{cases} 1 & \text{if } (i, j) \in E_1 \text{ and } (k, l) \in E_2 \\ 0 & \text{otherwise} \end{cases} \tag{1.70}$$

For a given non-crossing matching M in B we define its weight $w(M)$ as a sum over all couples of edges in M. The CMO problem consists then in maximizing $w(M)$, where M belongs to the set of all non-crossing matchings in B.

In [4, 5, 12, 58] we've already dealt with non-crossing matching and proposed a network flow presentation of similar one-to-one mappings (in fact the mapping there was many-to-one). The adaptation of this approach to CMO is as follows: The edges of the bipartite graph B are mapped to the points of $n_1 \times n_2$ rectangular grid $B' = (V', E')$ according to: point - $(i, k) \in V'$ $\longleftrightarrow$ edge - (i, k) in B.

Definition. The **feasible path** is an arbitrary sequence $(i_1, k_1), (i_2, k_2), \ldots, (i_t, k_t)$ of points in B' such that $i_j < i_{j+1}$ and $k_j < k_{j+1}$ for $j = 1, 2, \ldots, t - 1$.

The correspondence feasible path $\longleftrightarrow$ non-crossing matching is obvious. This way the problems on non-crossing matchings are converted to problems on feasible paths. We also add arcs $(i, k) \rightarrow (j, l) \in E'$ iff $w_{ikjl} = 1$. In B', solving CMO corresponds to finding the densest (in terms of arcs) subgraph of B' whose node set is a feasible path (see for illustration Fig. 1.21).

To each node $(i, k) \in V'$ we associate now a $0/1$ variable x_{ik}, and to each arc $(i, k) \rightarrow (j, l) \in E'$, a $0/1$ variable y_{ikjl}. Denote by X the set of feasible paths. The problem can now be stated as

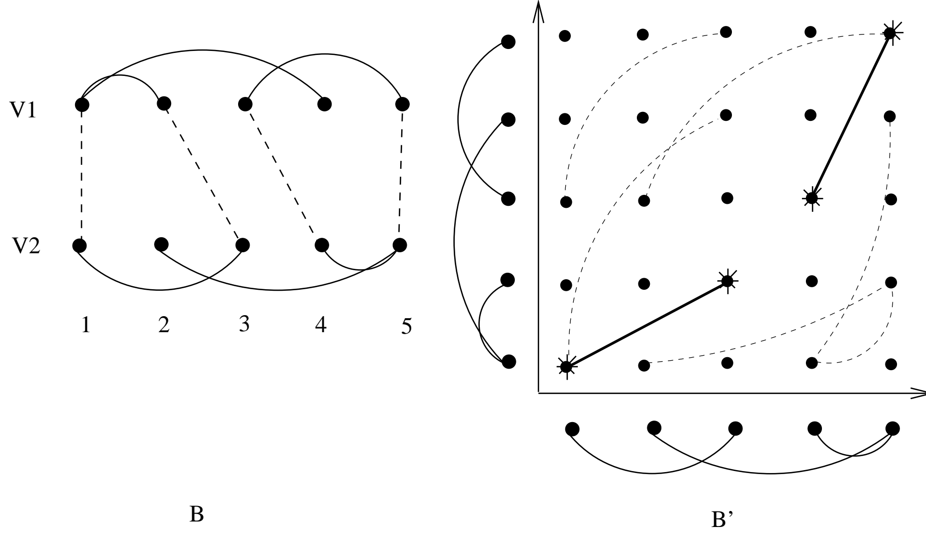


Figure 1.21: Left: Vertex 1 from V1 is matched with vertex 1 from V2 and 2 is matched with 3: matching couple $(1, 1)(2, 3)$. Other matching couples are $(3, 4)(5, 5)$. This defines a feasible matching $M = \{(1, 1)(2, 3), (3, 4)(5, 5)\}$ with weight $w(M) = 2$. Right: The same matching visualized in graph B'.

$$v(CMO) = \max \sum_{(ik)(jl)\in E'} y_{ikjl} \tag{1.71}$$

subject to

$$x_{ik} \geq \sum_{l\in\delta_2^+(k)} y_{ikjl}, \quad j \in \delta_1^+(i) \qquad \begin{matrix} i = 1, 2, \ldots, n1 - 1, \\ k = 1, 2, \ldots, n2 - 1 \end{matrix} \tag{1.72}$$

$$x_{ik} \geq \sum_{l\in\delta_2^-(k)} y_{jlik}, \quad j \in \delta_1^-(i) \qquad \begin{matrix} i = 2, 3, \ldots, n1, \\ k = 2, 3, \ldots, n2 \end{matrix} \tag{1.73}$$

$$x_{ik} \geq \sum_{j\in\delta_1^+(i)} y_{ikjl}, \quad l \in \delta_2^+(k) \qquad \begin{matrix} i = 1, 2, \ldots, n1 - 1, \\ k = 1, 2, \ldots, n2 - 1 \end{matrix} \tag{1.74}$$

$$x_{ik} \geq \sum_{j\in\delta_1^-(i)} y_{jlik}, \quad l \in \delta_2^-(k) \qquad \begin{matrix} i = 2, 3, \ldots, n1, \\ k = 2, 3, \ldots, n2. \end{matrix} \tag{1.75}$$

$$x \in X \tag{1.76}$$

Actually, we know how to represent X with linear constraints. Recalling the definition of feasible path, (1.76) is equivalent to

$$\sum_{l=1}^{k} x_{il} + \sum_{j=1}^{i-1} x_{jk} \leq 1, \quad i = 1, 2, \ldots, n1, \quad k = 1, 2, \ldots, n2. \tag{1.77}$$

We recall that from the definition of the feasible paths in B' (non-crossing matching in B) the j-th residue from $P1$ could be matched with at most one residue from $P2$ and vice-versa. This explains the sums into right hand side of (1.72) and (1.74) – for arcs having their tails at vertex (i,k); and (1.73) and (1.75)– for arcs heading to (i,k). Any $(i,k)(j,l)$ arc can be activated ($y_{ikjl} = 1$)iff $x_{ik} = 1$ and $x_{jl} = 1$ and in this case the respective constraints are active because of the objective function.

A tighter description of the polytop defined by (1.72)–(1.75) and $0 \le x_{ik} \le 1$, $0 \le y_{ikjl}$ could be obtained by lifting the constraints (1.73) and (1.75) as it is shown in Fig. 1.22. The points shown are just the predecessors of (i,k) in graph B' and they form a grid of $\delta_1^-(i)$ rows and $\delta_2^-(k)$ columns.Let $i_1, i_2, \ldots, i_s$ be all the vertices in $\delta_1^-(i)$ ordered according the numbering of the vertices in V_1 and likewise $k_1, k_2, \ldots, k_t$ in $\delta_2^-(k)$. Then the vertices in the l-th column $(i_1,k_l),(i_2,k_l),\ldots(i_s,k_l)$ correspond to pairwise crossing matchings and at most one of them could be chosen in any feasible solution $x \in X$ (see (1.75)). This "all crossing" property will stay even if we add to this set the following two sets: $(i_1,k_1),(i_1,k_2),\ldots,(i_1,k_{l-1})$ and $(i_s,k_{l+1}),(i_s,k_{l+2},\ldots,(i_s,k_t)$. Denote by $col_{ik}(l)$ the union of these three sets and analogously by $row_{ik}(j)$ the corresponding union for the j-th row of the grid. When the grid is one column (row) only the set $row_{ik}(j)(col_{ik}(l))$is empty.

Now a tighter LP relaxation of (1.72)–(1.75) is obtained by changing (1.73) with

$$x_{ik} \ge \sum_{(r,s)\in row_{ik}(j)} y_{rsik}, \quad j \in \delta_1^-(i) \qquad \begin{array}{l} i = 2,3,\ldots,n1, \\ k = 2,3,\ldots,n2 \end{array} \tag{1.78}$$

and (1.75) with

$$x_{ik} \ge \sum_{(r,s)\in col_{ik}(l)} y_{rsik}, \quad l \in \delta_2^-(k) \qquad \begin{array}{l} i = 2,3,\ldots,n1, \\ k = 2,3,\ldots,n2. \end{array} \tag{1.79}$$

Remark:Since we are going to apply the Lagrangian technique there is no need neither for an explicit description of the set X neither for lifting the constraints (1.72) (1.74).

1.4.3 Lagrangian relaxation approach

Here, we show how the Lagrangian relaxation of constraints (1.78) and (1.79) leads to an efficiently solvable problem, yielding upper and lower bounds that are generally better than those found by the best known exact algorithm [46].

Let $\lambda_{ikj}^h \ge 0$ (respectively $\lambda_{ikj}^v \ge 0$) be a Lagrangian multiplier assigned to each constraint (1.78) (respectively (1.79). By adding the slacks of these constraints to the objective function with weights λ, we obtain the Lagrangian relaxation of the CMO problem

$$\begin{aligned} LR(\lambda) = \quad & \max \sum_{i,k,j\in\delta_1^-(i)} \lambda_{ikj}^h (x_{ik} - \sum_{(r,s)\in row_{ik}(j)} y_{rsik}) \\ & + \sum_{i,k,l\in\delta_2^-(k)} \lambda_{ikl}^v (x_{ik} - \sum_{(r,s)\in col_{ik}(l)} y_{rsik}) + \sum_{(ik)(jl)\in E_{B'}} y_{ikjl} \end{aligned} \tag{1.80}$$

subject to $x \in X$, (1.72), (1.74) and $y \ge 0$.

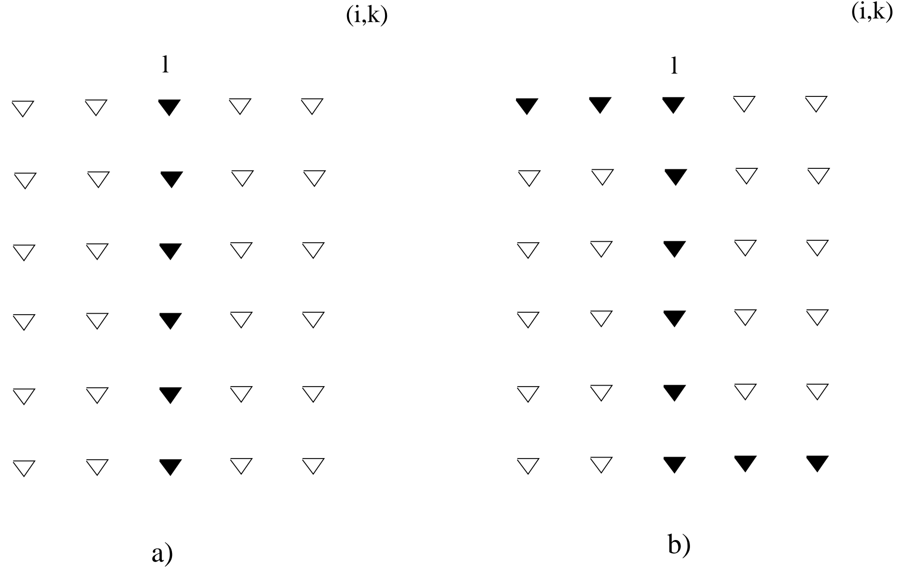


Figure 1.22: $\triangledown$: set of vertices in V' which are tails for the arcs heading to (i, k). In a): $\blacktriangledown$ correspond to the indices of y_{jlik} in (1.75) for l fixed. In b): $\blacktriangledown$ correspond to the indices of y_{jlik} in (1.79) for l fixed.

Proposition 3. *$LR(\lambda)$ can be solved in $O(|V'| + |E'|)$ time.*

Proof:

For each $(i, k) \in V'$, if $x_{ik} = 1$ then the optimal choice y_{ikjl} amounts to solving the following : The heads of all arcs in E' outgoing from (i, k) form a $|\delta^+(i)| \times |\delta^+(k)|$ table. To each point (j, l) in this table, we assign the profit $\max\{0, c_{ikjl}(\lambda)\}$, where $c_{ikjl}(\lambda)$ is the coefficient of y_{ikjl} in (1.80). Each vertex in this table is a head of an arc outgoing from (i, k).Then the subproblem we need to solve consists in finding a subset of these arcs having a maximal sum $c_{ik}(\lambda)$ of profits(the arcs of negative weight are excluded as a candidates for the optimal solution) and such that their heads lay on a feasible path. This could be done by a dynamic programming approach in $O(|\delta^+(i)||\delta^+(k)|)$ time. Once profits $c_{ik}(\lambda)$ have been computed for all (i, k) we can find the optimal solution to $LR(\lambda)$ by using the same DP algorithm but this time on the table of $n1 \times n2$ points with profits for (i, k)-th one given by

$$c_{ik}(\lambda) + \sum_{j \in \delta_1^-(i)} \lambda^h_{ikj} + \sum_{l \in \delta_2^-(k)} \lambda^v_{ikl}. \tag{1.81}$$

where the last two terms are the coefficients of x_{ik} in (1.80).

Remark:The inclusion $x \in X$ is explicitly incorporated in the DP algorithm.

The algorithm

In order to find the tightest upper bound on $v(CMO)$ (or eventually to solve the problem) , we need to solve in the dual space of the Lagrangian multipliers $LD = \min_{\lambda \geq 0} LR(\lambda)$, whereas

$LR(\lambda)$ is a problem in x, y. A number of methods have been proposed to solve Lagrangian duals: subgradient method, dual ascent methods,constraint generation method, column generation, bundel methods,augmented lagrangian methods, etc. Here, we choose the subgradient method. It is an iterative method in which at iteration t, given the current multiplier vector λ^t, a step is taken along a subgradient of $LR(\lambda)$, then if necessary, the resulting point is projected onto the nonnegative orthant. It is well known that practical convergence of the subgradient method is unpredictable. For some problems, convergence is quick and fairly reliable, while other problems tend to produce erratic behavior of the multiplier sequence, or the Lagrangian value, or both.In a "good" case, one usually observe a saw-tooth pattern in the Lagrangian value for the first iterations, followed by a roughly monotonic improvement and asymptotic convergence to a value that is hopefully the optimal lagrangian bound. The computational runs on a reach set of real-life instances confirm a "good" case belonging of our approach at some expense in the speed of the convergence.

In our realization, the update scheme for λ_{ikj} is $\lambda_{ikj}^{t+1} = \max\{0, \lambda_{ikj}^t - \Theta^t g_{ikj}^t\}$ [8], where $g_{ikj}^t = \bar{x}_{ik} - \sum \bar{y}_{jlik}$ is the sub-gradient component($0, 1$,or -1), calculated on the optimal solution $\bar{x}$, $\bar{y}$ of $LR(\lambda^t)$. The step size Θ^t is $\Theta^t = \frac{\alpha(LR(\lambda^t) - Z_{lb})}{\sum(g_{ikj}^t)^2 + \sum(g_{ikl}^t)^2}$ where Z_{lb} is a known lower bound for the CMO problem and α is an input parameter. Into this approach the x-components of $LR(\lambda^t)$ solution provides a feasible solution to CMO and thus a lower bound also . The best one (incumbent) so far obtained is used for fathoming the nodes whose upper bound falls below the incumbent and also in section 1.4.4 for reporting the final gap.If $LD \leq v(CMO)$ then the problem is solved. If $LD > v(CMO)$ holds, in order to obtain the optimal solution, one could pass to a branch&bound algorithm suitably tailored for such an upper bounds generator.

From among various possible nodes splitting rules, the one shown in Fig. 1.23 gives quite satisfactory results (see section 1.4.4). Formally, let the current node be a subproblem of CMO defined over the vertices of V' falling in the interval $[lc(i), uc(i)]$ for $i = 1, n_2$(in Fig. 1.23these are the points in-between two broken lines).Let $(rowbest, colbest)$ be the $\arg\max\min(S_u(i,k), S_d(i,k))$, where $S_d(i,k) = \sum_{j \leq k} \max(uc(j) - i, 0)$ and $S_u(i,k) = \sum_{j \geq k} \max(i - lc(j), 0)$. Now, the two descendants of the current node are obtained by discarding from its feasible set the vertices in $S_d(rowbest, colbest)$ and $S_u(rowbest, colbest)$ resp.The goal of this strategy is twofold:to create descendants that are balanced in sense of feasible set size and to reduce maximally the parent node's feasible set.

In addition, the following heuristics happened to be very effective during the traverse of the B&B tree nodes. Once the lower and the upper bound are found at the root node, an attempt to improve the lower bound is realized as follows.

Let $(i_{k_1}, k_1), (i_{k_2}, k_2), \ldots, (i_{k_s}, k_s)$ be an arbitrary feasible path which activates certain number of arcs (recall that each iteration in the sub-gradient optimization phase generates such path and lower bound as well).

Then for a given strip size sz (an input parameter set by default to 4), the matchings in the original CMO are restricted to fall in a neighborhood of this path, allowing x_{ik} to be non zero only for

$$\max\{1, i_j - sz\} \leq i \leq \min\{n1, i_j + sz\}, j = k_1, k_2, \ldots, k_s.$$

[8] analogously for λ_{ikl}

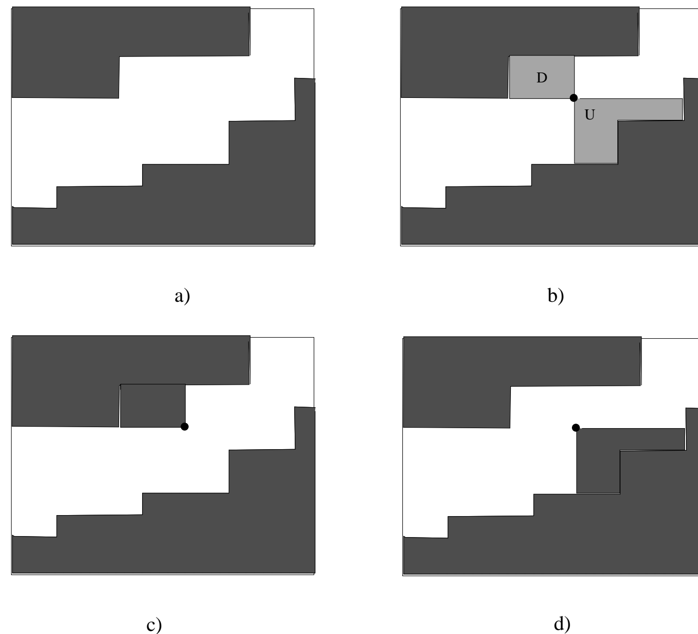


Figure 1.23:

The Lagrangian dual of this subproblem is solved and a better lower bound is possibly sought. If the bound improves the incumbent,the same procedure is repeated by changing the strip alongside the new feasible solution.

Finally, the main steps of the B&B algorithm are as follows:
Initialization: Set L={original CMO problem, i.e. no restrictions on the feasible paths}.
Problem selection and relaxation: Select and delete the problem P^i from L having the biggest upper bound. Solve the Lagrangian dual of P^i. (Here a repetitive call to a heuristics is included after each improvement on the lower bound).
Fathoming and Pruning: Follow the classical rules.
Partitioning : Create two descendants of P^i using $(rowbest, colbest)$. Add these descendants to L.
Termination : if $L = \emptyset$, the solution (x^*, y^*) which yielded the incumbent objective value is optimal.

1.4.4 Computational results

The numerical results presented in this section were obtained on a cluster of 12 AMD Opteron(TM) CPU 2.4 GHz, 4 Gb Ram, RedHat 9 Linux, connected by a 1 Gb Ethernet network. The algorithm was implemented in C. To test its performance we used a set of large proteins suggested by Jeffrey Skolnick that was used in various recent papers related to protein structure comparison [46, 70]. This set contains 33 proteins with a total of 40 domains classified by SCOP into five

families (see Table 1.10) [9]. Below we compare the performance of our approach with the previously known exact algorithm [46]. Note that both approaches use diverse (but Lagrangian type) relaxations. Our algorithm will be called a_purva[10] while the other Lagrangian algorithm is denoted here by (LR)[11].

	Fold	Family	Proteins
1	Flavodoxin-like	CheY-related	1b00, 1dbw, 1nat, 1ntr, 1qmp(A,B,C,D), 3chy, 4tmy(A,B)
2	Cupredoxin-like	Plastocyanin/ azurin-like	1baw, 1byo(A,B), 1kdi, 1nin, 1pla 2b3i, 2pcy, 2plt
3	TIM beta/alpha-barrel	Triosephosphate isomerase (TIM)	1amk, 1aw2, 1b9b, 1btm, 1hti 1tmh, 1tre, 1tri, 1ydv, 3ypi, 8tim
4	Ferritin-like	Ferritin	1b71, 1bcf, 1dps, 1hfa, 1ier, 1rcd
5	Microbial ribonucleases	Fungal ribonucleases	1rn1(A,B,C)

Table 1.10: The Skolnick set

The Skolnick set requires aligning 780 pairs of proteins. Those are medium size proteins, the number of their residues varies from 95 (2b3iA) to 252 (1aw2A). The maximum number of contacts is 593 (1btmA). We bounded the execution time to 1800 seconds for both algorithms. a_purva succeeded to solve 171 couples for the given period of time, while LR solved only 157 couples. Figure 1.24 illustrates LR/a_purva time ratio as a function of solved instances. It is easily seen that a_purva is significantly faster than LR (up to several hundred times in the majority of cases). Table 1.11 contains more details concerning the first 164 pairs of proteins. We observed that this set is a very interesting one. It is characterized by the following properties: a) in all but 5 instances the a_purva running time is less than 10 seconds; b) in all instances the relative gap[12] at the root of the B&B is smaller than 4, while in all other instances this gap is much larger : greater than 18 even for the couples we succeeded to solve for less than 1800 sec: c) this set contains all instances such that both proteins belong to the same family according SCOP classification. In other words, each pair such that both proteins belong to the same family is an easily solvable instance for a_purva and this feature can be successfully used as a discriminator (at least for the Skolnick set). In fact, by virtue of this relation (similar structure - less computational time and vice-versa) we were able to correctly classify this 40 items set in 2000 seconds overall running time on all 780 instances.

[9]Caprara et al. [46] mention only four families. This wrong classification is also accepted in other studies [70]. The families are in fact five as shown in Table 1.10. According to SCOP classification the protein 1arn1 does not belong to the first family as indiquated in [46]. Note that this corroborates the results obtained in [46] but the authors considered it as a mistake.

[10]Apurva (Sanskrit) = not having existed before, unknown, wonderful, ...

[11]The code of LR, as well as the contact map graphs for the Skolnick set, were kindly provided to us by Giuseppe Lancia.

[12]We define the relative gap as $100 \times \frac{UB-LB}{UB}$.

F	Proteins Name	CMO	Time LR	Time a_pr	Proteins Name	CMO	Time LR	Time a_pr
1	1b00A 1dbwA	149	192.00	1.2	1ntr_ 1qmpA	119	545.94	7.18
1	1b00A 1nat_	145	166.98	1.11	1ntr_ 1qmpB	115	454.01	4.23
1	1b00A 1ntr_	118	565.47	3.59	1ntr_ 1qmpC	116	610.93	6.56
1	1b00A 1qmpA	143	198.72	1.33	1ntr_ 1qmpD	118	522.53	4.44
1	1b00A 1qmpB	136	439.95	59.65	1ntr_ 3chy_	130	339.86	5.53
1	1b00A 1qmpC	139	263.81	1.68	1ntr_ 4tmyA	126	450.05	3.34
1	1b00A 1qmpD	137	181.23	1.89	1ntr_ 4tmyB	127	399.26	3.75
1	1b00A 3chy_	154	141.50	0.85	1qmpA 1qmpB	221	3.77	0.03
1	1b00A 4tmyA	155	143.92	0.9	1qmpA 1qmpC	232	0.35	0.02
1	1b00A 4tmyB	155	75.41	0.73	1qmpA 1qmpD	230	0.02	0.03
1	1dbwA 1nat_	157	226.42	1.51	1qmpA 3chy_	160	69.78	1.07
1	1dbwA 1ntr_	130	426.13	5.53	1qmpA 4tmyA	162	98.21	0.78
1	1dbwA 1qmpA	152	159.74	2.93	1qmpA 4tmyB	164	50.48	0.62
1	1dbwA 1qmpB	150	63.63	1.52	1qmpB 1qmpC	221	1.60	0.02
1	1dbwA 1qmpC	150	180.52	2.38	1qmpB 1qmpD	220	1.61	0.03
1	1dbwA 1qmpD	152	111.28	1.78	1qmpB 3chy_	156	68.17	0.84
1	1dbwA 3chy_	164	84.22	1.19	1qmpB 4tmyA	157	51.32	0.58
1	1dbwA 4tmyA	161	73.71	1.1	1qmpB 4tmyB	156	66.11	0.64
1	1dbwA 4tmyB	163	47.87	1.11	1qmpC 1qmpD	226	3.65	0.02
1	1nat_ 1ntr_	127	302.39	3.59	1qmpC 3chy_	157	75.14	1.23
1	1nat_ 1qmpA	157	66.03	1.04	1qmpC 4tmyA	162	55.46	1.26
1	1nat_ 1qmpB	149	69.00	0.99	1qmpC 4tmyB	162	78.52	0.58
1	1nat_ 1qmpC	152	73.53	1.07	1qmpD 3chy_	158	59.47	1.11
1	1nat_ 1qmpD	151	99.14	1.33	1qmpD 4tmyA	157	59.23	0.71
1	1nat_ 3chy_	163	76.95	0.86	1qmpD 4tmyB	159	53.27	0.59
1	1nat_ 4tmyA	175	15.58	0.28	3chy_ 4tmyA	171	54.33	0.55
1	1nat_ 4tmyB	172	19.06	0.37	3chy_ 4tmyB	174	41.43	0.5
1					4tmyA 4tmyB	230	0.02	0.02
2	1bawA 1byoA	152	11.59	0.25	1byoB 2b3iA	135	7.21	0.27
2	1bawA 1byoB	155	6.11	0.18	1byoB 2pcy_	175	2.28	0.05
2	1bawA 1kdi_	140	33.84	0.55	1byoB 2plt_	174	3.90	0.06
2	1bawA 1nin_	153	9.45	0.21	1kdi_ 1nin_	129	52.53	1.13
2	1bawA 1pla_	124	28.04	0.62	1kdi_ 1pla_	126	33.59	0.89
2	1bawA 2b3iA	130	15.57	0.38	1kdi_ 2b3iA	122	40.83	0.84
2	1bawA 2pcy_	148	6.91	0.16	1kdi_ 2pcy_	145	15.19	0.3
2	1bawA 2plt_	161	5.22	0.13	1kdi_ 2plt_	150	24.56	0.32
2	1byoA 1byoB	192	2.61	0.02	1nin_ 1pla_	130	22.76	0.69
2	1byoA 1kdi_	148	17.89	0.35	1nin_ 2b3iA	129	25.55	0.5
2	1byoA 1nin_	140	30.14	0.85	1nin_ 2pcy_	139	23.31	0.49
2	1byoA 1pla_	150	7.55	0.16	1nin_ 2plt_	146	18.85	0.52
2	1byoA 2b3iA	132	10.26	0.39	1pla_ 2b3iA	122	12.65	0.32
2	1byoA 2pcy_	176	2.18	0.04	1pla_ 2pcy_	143	4.75	0.14
2	1byoA 2plt_	172	3.77	0.07	1pla_ 2plt_	144	7.10	0.17
2	1byoB 1kdi_	152	11.89	0.21	2b3iA 2pcy_	127	11.79	0.35
2	1byoB 1nin_	141	21.05	0.6	2b3iA 2plt_	140	7.37	0.17
2	1byoB 1pla_	148	6.94	0.16	2pcy_ 2plt_	172	3.67	0.06
3	1amk_ 1aw2A	411	1272.28	1.48	1btmA 1tmhA	432	1801.97	2.81
3	1amk_ 1b9bA	400	1044.23	2.04	1btmA 1treA	433	1512.26	2.59

3	1amk_ 1btmA	427	1287.48	2.38	1btmA 1tri_	419	1455.08	3.26
3	1amk_ 1htiA	407	265.16	1.4	1btmA 1ydvA	385	692.72	1.52
3	1amk_ 1tmhA	424	638.26	1.29	1btmA 3ypiA	406	1425.09	2.43
3	1amk_ 1treA	411	716.51	1.52	1btmA 8timA	408	940.59	2
3	1amk_ 1tri_	445	447.54	0.97	1htiA 1tmhA	416	588.98	1.07
3	1amk_ 1ydvA	384	462.44	1.05	1htiA 1treA	426	395.23	0.81
3	1amk_ 3ypiA	412	427.66	0.97	1htiA 1tri_	412	779.84	1.55
3	1amk_ 8timA	410	386.73	0.94	1htiA 1ydvA	382	405.04	1.09
3	1aw2A 1b9bA	411	961.04	3.28	1htiA 3ypiA	422	148.75	0.56
3	1aw2A 1btmA	434	750.67	3.1	1htiA 8timA	463	112.65	0.52
3	1aw2A 1htiA	425	363.03	1.78	1tmhA 1treA	513	119.27	0.23
3	1aw2A 1tmhA	474	185.72	0.51	1tmhA 1tri_	413	630.57	2.19
3	1aw2A 1treA	492	157.79	0.37	1tmhA 1ydvA	384	785.56	1.5
3	1aw2A 1tri_	408	1313.53	3.51	1tmhA 3ypiA	417	766.79	2.11
3	1aw2A 1ydvA	386	650.55	1.62	1tmhA 8timA	421	516.44	1.47
3	1aw2A 3ypiA	401	895.17	2.28	1treA 1tri_	401	1169.41	2.68
3	1aw2A 8timA	423	276.06	1.76	1treA 1ydvA	389	1419.90	2.21
3	1b9bA 1btmA	441	653.29	2.08	1treA 3ypiA	407	522.65	1.34
3	1b9bA 1htiA	394	809.23	2.27	1treA 8timA	425	310.95	1.15
3	1b9bA 1tmhA	418	548.56	1.34	1tri_ 1ydvA	371	1040.31	1.92
3	1b9bA 1treA	410	613.99	1.25	1tri_ 3ypiA	412	607.52	1.75
3	1b9bA 1tri_	391	1804.98	3.32	1tri_ 8timA	412	830.38	1.45
3	1b9bA 1ydvA	362	1608.97	6.1	1ydvA 3ypiA	374	355.82	0.92
3	1b9bA 3ypiA	396	700.45	1.88	1ydvA 8timA	388	399.47	0.99
3	1b9bA 8timA	392	634.48	1.66	3ypiA 8timA	418	267.14	0.65
3	1btmA 1htiA	403	1566.88	3.51				
4	1b71A 1bcfA	211	1800.08	453.08	1bcfA 1rcd_	222	528.84	1.99
4	1b71A 1dpsA	174	1800.43	266.54	1dpsA 1fha_	180	1800.24	9.45
4	1b71A 1fha_	216	1802.46	303.02	1dpsA 1ier_	184	1800.31	8.42
4	1b71A 1ier_	214	1801.32	480.43	1dpsA 1rcd_	184	1490.02	5.7
4	1b71A 1rcd_	211	1802.48	319	1fha_ 1ier_	299	69.34	0.25
4	1bcfA 1dpsA	187	510.17	3.81	1fha_ 1rcd_	295	36.40	0.19
4	1bcfA 1fha_	218	1017.59	2.69	1ier_ 1rcd_	297	24.03	0.15
4	1bcfA 1ier_	226	556.33	3.28				
5	1rn1A 1rn1B	191	1.23	0.03	1rn1B 1rn1C	197	0.21	0.01
5	1rn1A 1rn1C	190	1.01	0.03				
6	1qmpD 1tri_	131	1801.09	1674.98	1byoB 1rn1C	66	1800.09	686.03
6	1kdi_ 1qmpD	73	1800.15	904.75	1dbwA 1treA	145	1802.01	1703.2
6	1tmhA 4tmyB	112	1802.80	1521.23	1dbwA 1tri_	149	1800.73	1173.5
6	1dpsA 4tmyB	89	1800.39	913.24				

Table 1.11: Column one contains the number of the families according to table 1.10. The sixth class contains the hardest solved Skolnick set intstances. Column two(six) contains the names of the couples, column three(seven) is the score, column four(height) gives the time in seconds taken by LR algoritm, and column five(nine) presents the corresponding time taken by a_purva.

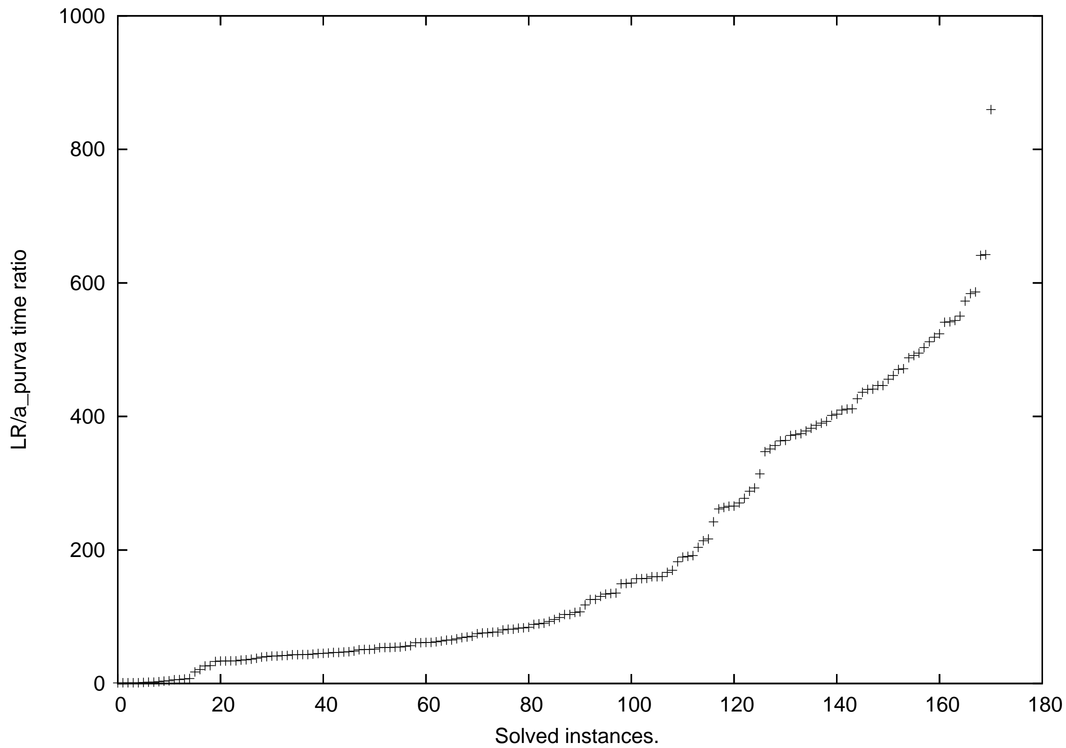


Figure 1.24: $\frac{\text{LR time}}{\text{a_purva time}}$ ratio as a function of solved instances

Our next observation (see Figures 1.25 and 1.26) concerns the quality of gaps obtained by both algorithms on the set of unsolved instances. Remember that when a Lagrangian algorithm stops because of time limit (1800 sec. in our case) it provides two bounds: one upper (UB), and one lower (LB). Providing these bounds is a real advantage of a B&B type algorithm compared to any metaheuristics. These values can be used as a measure for how far is the optimization process from finding the exact optimum. The value UB-LB is usually called absolute gap. Any one of the 609 points (x, y) in Figure 1.25) presents the absolute gap for `a_purva` (x coordinate) and for `LR` (y coordinate) algorithm. All points are above the $y = x$ line (i.e. the absolute gap for `a_purva` is always smaller than the absolute gap for `LR`). On the other hand the entire figure is very asymmetric in a profit of our algorithm since its maximal absolute gap is 33, while it is 183 for `LR`.

We afterwards similarly compared lower and upper bounds separately. This is illustrated in Figures 1.26. Any point denoted by $\circ$ has the lower bound computed by `a_purva`(`LR`) as x (y) coordinate, while any point denoted by $\times$ has the upper bound computed by `a_purva`(`LR`) as x (y) coordinate. We observe that in a large majority the points $\circ$ are below the $y = x$ line while the points $\times$ are above this line. This shows that usually the lowers bounds found by `a_purva` are higher, while its upper bounds are all smaller and it is clear that `a_purva` significantly outperforms `LR` on quality of its bounds.

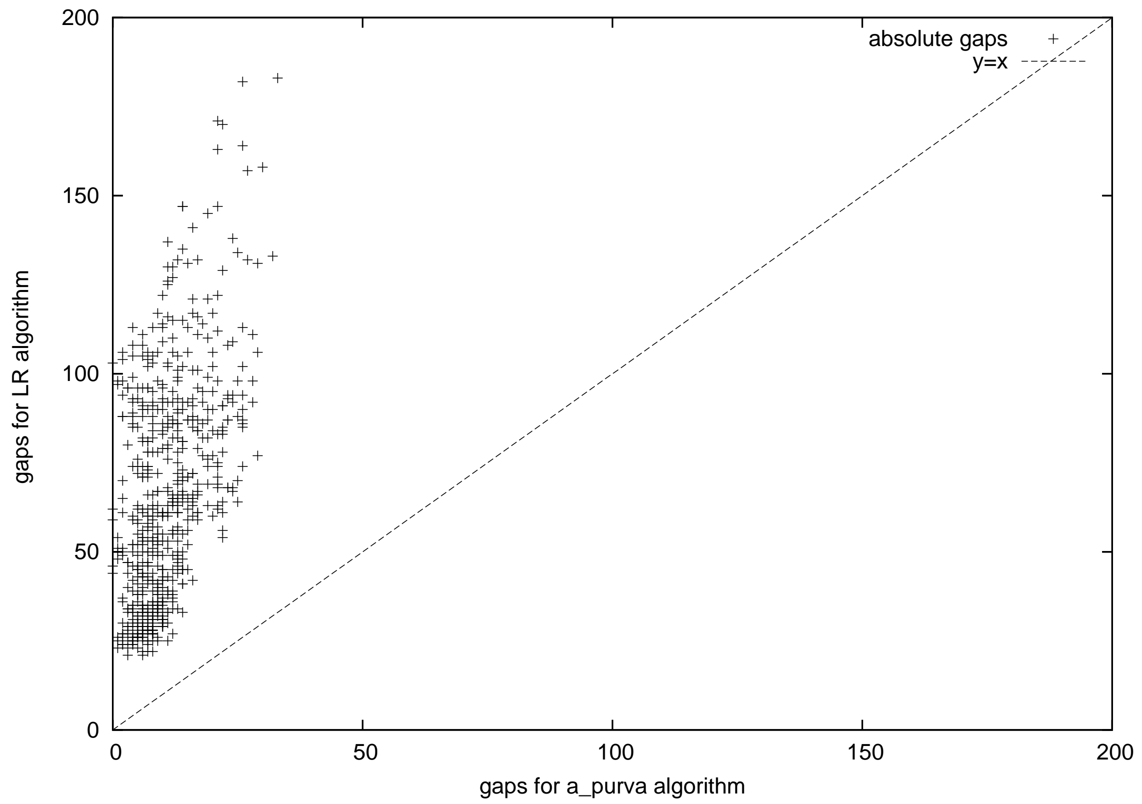


Figure 1.25: Comparing absolute gaps on the set of unsolved instances. The gaps computed by a_purva are significantly smaller.

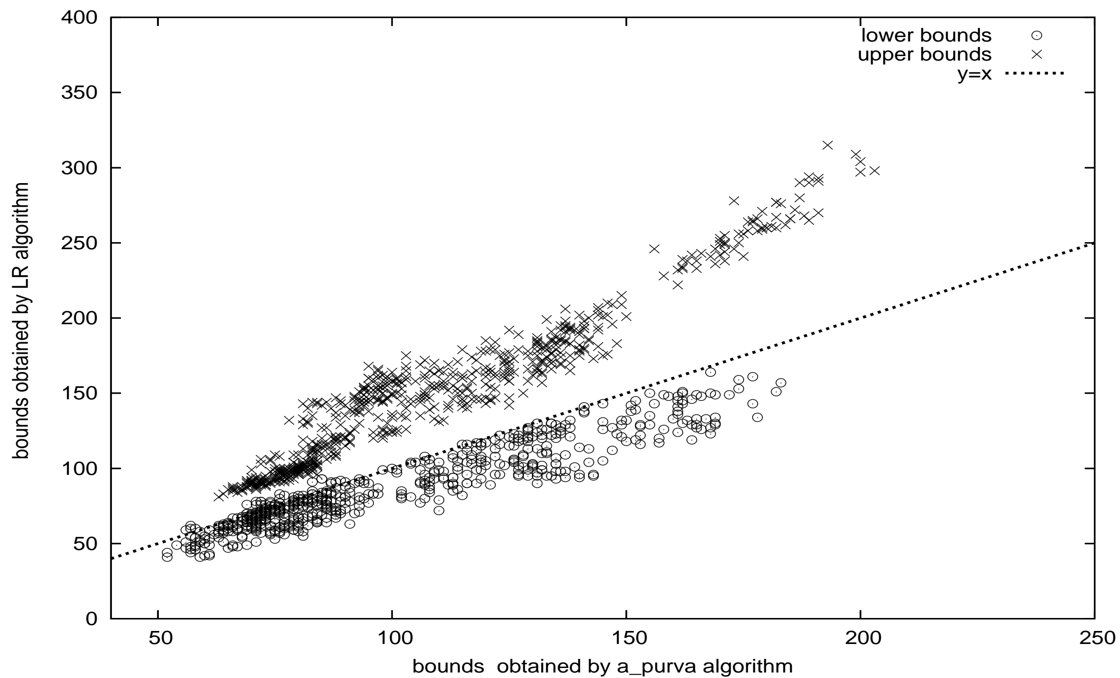


Figure 1.26: Comparing the quality of lower and upper bounds on the set of unsolved instances. a_purva clearly ourperforms LR on the quality of its bounds.

1.4.5 Conclusion

In this section, we give efficient exact B&B algorithm for contact map overlap problem . The bounds are found by using Lagrangian relaxation and the dual problem is solved by sub-gradient approach. The efficiency of the algorithm is demonstrated on a benchmark set of 780 instances and the dominance over the existing algorithms is total. When the algorithm is used for classification purposes (and this was the primary goal) the average time for correctly classifying two proteins of the same class is 0.6 seconds.

1.5 Optimal Segmentation of Bacterium Genomes

1.5.1 Introduction

A practical way to study the plasticity of bacterium genomes without systematically sequencing all the available strains is to exploit the LR-PCR (Long Range Polymerase Chain Reaction) technique. The genomes of the strains are split into a large number of short segments before performing a LR-PCR on each of them. Depending on the reorganization, the deletion or the insertion of certain genomic zones, it is expected that a few segments will not be amplified. Thus, a *profile* – or a signature – can be assigned to each strain. It represents the set of amplified and non amplified segments. The final step is to perform a global analysis of all the profiles. This strategy, recently tested by Ohnishi *et al.* [60] to study the genome diversity of *E. coli*, is explained on Fig. 1.27.

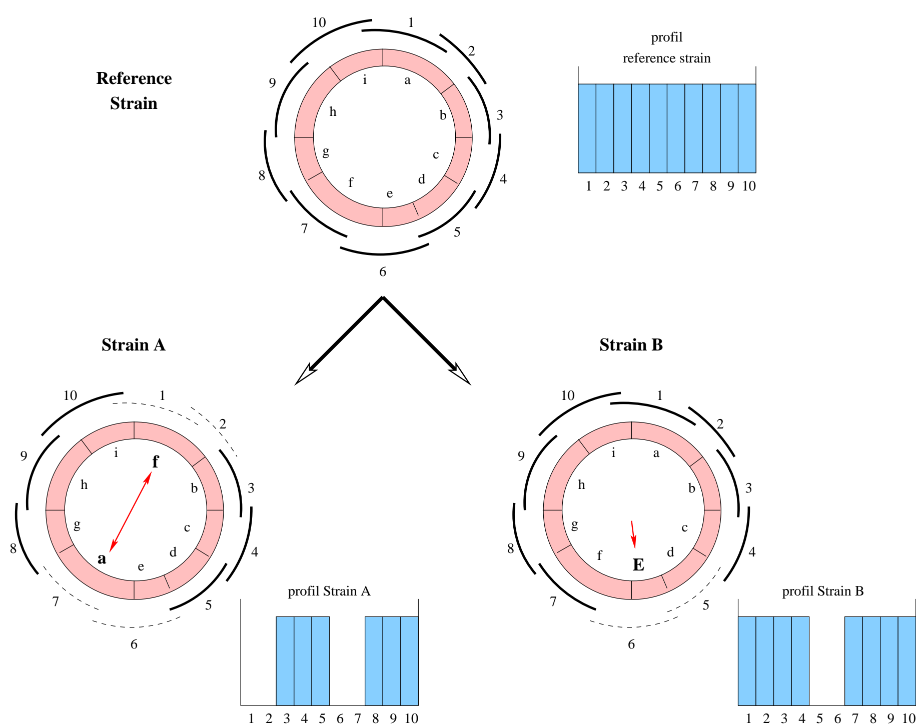


Figure 1.27: Strategy to study the plasticity of a circular bacterial genome: a reference strain is fully covered with overlapping segments. The two extremities of each segment are characterized by starting and ending primers. On the reference strain, the LR-PCR amplifies all the segments. On strain A the zones **a** and **f** have swapped, preventing the amplification of segments 1, 2, 6 and 7. On strain B, zone **e** is modified: segments 5 and 6 cannot be amplified.

This strategy first implies to determine the set of segments which will cover the genome. A strain whose genome is entirely sequenced is chosen as reference. Then, potential PCR primers are localized on the genome since they specify the position where the segments start and end.

Having these data, the goal is to cover the genome with overlapping segments of nearly identical size, knowing that the segments locations are constrained by the position of the primers.

Actually, the distribution of the primer sites along the bacterium genome is non-uniform. There may have large regions (a few Kbp) without primer sites or, on the contrary, very dense regions of primer sites. In addition, some regions are forbidden: they correspond to repeated zones, bacteriophage sequences, or mobile elements such as transposons. As some of these regions are longer than the expected length of the covering segments, the circular genome is cut into a few number of linear pieces, called domains (see Fig. 1.28).

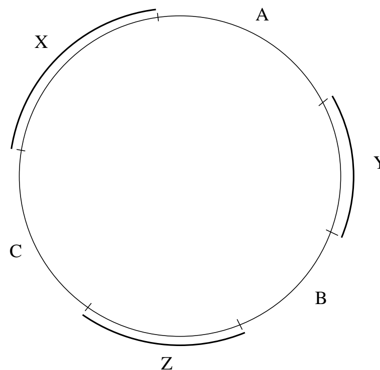


Figure 1.28: Regions X, Y and Z are forbidden. The length of each one of them is larger than the size of the covering segments. The problem of segmenting a full genome is therefore transformed in segmenting three linear pieces denoted here as A, B and C and called domains. Any domain is associated to the solution of an independent subproblem.

Thus, the problem of segmenting a complete bacterial genome is reduced to cover each domain with segments of nearly identical size. Along a domain, there are specific positions called *primer sites*. The overlapping segments can start and end *only* at these particular positions. If we assume, for the sake of simplicity, that a solution is made of a list S of N segments, and that each segment can take only P different positions, then the number of possibilities is equal to P^N. Finding the best one when N is large is clearly a combinatorial problem (in real application, $N > 100$).

More formally, the problem can be formulated as follows. Given a *domain*, i.e. a DNA sequence ranging from a few 100 Kpb to a few Mbp, together with all potential primer positions, we need to cover it with a sequence of overlapping segments of nearly identical size. Such a covering will be called a *segmentation* if the segments satisfy the following conditions:

- The length of any segment varies in the range $[\underline{L}, \overline{L}]$.
- The length of the overlap between any two consecutive segments varies in the range $[\underline{O}, \overline{O}]$.
- The distance from the beginning of the domain to the starting-primer of the first segment is no more than D_s. The distance from the ending-primer of the last segment to the end of the domain is no more than D_e.

Two cases of this problem have been considered. In the first one we search for a sequence S of overlapping segments, each one of size as close as possible to a *given* ideal length L. In the second case, the value of L is *unknown* and we look for a couple (L^*, S), where L^*, $\underline{L} \leq L^* \leq \overline{L}$, and such that the sequence S is of of minimal error with respect to L^*.

For each case we: (i) formulate a suitable combinatorial optimization model; (ii) program dedicated algorithm for solving these models; (iii) analyze the complexity of the proposed algorithms. We are not aware of other algorithms from the literature to have been used for this purpose. This paper focusses on the algorithmic aspects of the problem. The reader interested in the genomic aspects can find more details in the accompanying paper [63].

Organization of the paper is as follows. The formal statement of the problem and definitions are given in section 1.5.2. Section 1.5.3 considers the first case of the problem, while the second case is discussed in section 1.5.4. Numerical results and complexity analysis are provided in section 1.5.5.

1.5.2 Graph problem formulation

The formal statement of the problem is as follows. Let be given: i) a nucleotide sequence D containing D_L elements (called domain); ii) a set S^l of starting-primer sites; iii) a set S^r of ending-primer sites. We can then define the set F of *feasible segments*, i.e. couples of starting and ending primers, $f = [b, e], \quad b < e$, such that:

- $b \in S^l, \quad e \in S^r$.
- the length $l(f) = e - b$ satisfies $\underline{L} \leq l(f) \leq \overline{L}$.

Let us denote by F_s (resp. F_t) the set of segments which can begin (resp. end) a segmentation. This means that if $f = [b, e] \in F_s$ then $b \leq D_s$ and that if $f = [b, e] \in F_t$ then $D_L - e \leq D_e$.

Definition 4. The segment f' *is compatible with* the segment f (denoted as $f \prec f'$), iff f' starts to the left of the ending-primer site of f and the length of the overlap is in $[\underline{O}, \overline{O}]$.

Definition 5. A sequence $S = f_1, f_2, \ldots f_k$ of feasible segments will be referred to as a *covering sequence (segmentation)* if $f_1 \in F_s, f_k \in F_t$ and $f_i \prec f_{i+1}$.

Definition 6. A *covering graph* of the nucleotide sequence is a directed graph $G(V, A)$:

- the node set $V = F \cup \{s, t\}$, where s and t two additional vertices.
- the arc set

$$\begin{aligned} A = \quad & \{(f, f') \in F \times F \ : f \prec f'\} \\ \cup \quad & \{(s, f) \in \{s\} \times F_s\} \\ \cup \quad & \{(f, t) \in F_t \times \{t\}\} \end{aligned}$$

Remark 4. Note that the covering graph $G(V, A)$ is without circuits because of the binary relation "is compatible with". The non-directed version of this graph is a subgraph of the so called interval graph (see chapter 1.5.4 [62]) over the set of feasible intervals.

1.5.3 The case when the segment length is given

In this section we assume that an *ideal* length L is given and we define a cost function $C_L(f)$ on F as: $\forall f \in F \quad C_L(f) = |l(f) - L|$. The problem to solve can be considered as a $\boldsymbol{minmax}$ (bottleneck) variant of the classical Shortest Path Problem (**SPP**), if the length of a path $\boldsymbol{r} = \boldsymbol{s}, \boldsymbol{v_1}, \ldots \boldsymbol{v_k}, \boldsymbol{t}$ is determined by $\boldsymbol{C_L(r)} = \max_{\boldsymbol{v_i} \in \boldsymbol{r}} \boldsymbol{C_L(v_i)}$.

One can easily see an one-to-one correspondence between covering sequences and the directed paths from $\boldsymbol{s}$ to $\boldsymbol{t}$ in $\boldsymbol{G}$. In this context the length of a path can be viewed as the error of the segmentation associated to this path. If we denote by $\boldsymbol{R}$ the set of paths from $\boldsymbol{s}$ to $\boldsymbol{t}$, the problem to be solved is $\min_{\boldsymbol{r} \in \boldsymbol{R}} \boldsymbol{C_L(r)} = \boldsymbol{C^*_L}$. An instance of the problem is given on Fig. 1.29, while its corresponding covering graph is depicted on Fig. 1.30.

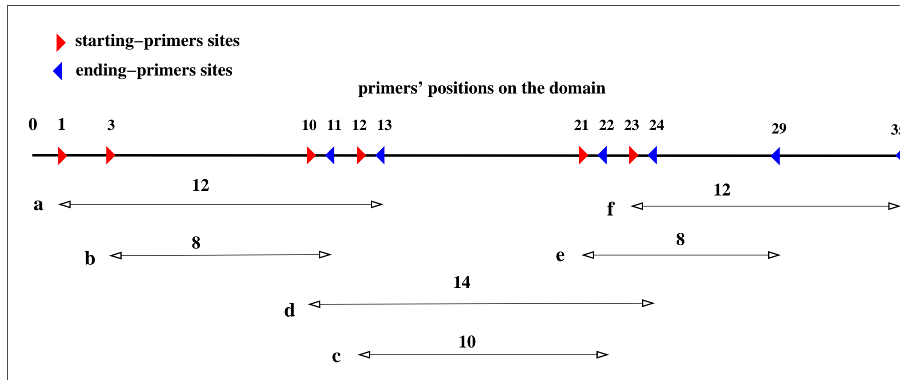


Figure 1.29: An instance of the problem where: $\boldsymbol{D_L} = 35$, $\boldsymbol{S^l} = \{1, 3, 10, 12, 21, 23\}$, $\boldsymbol{S^r} = \{11, 13, 22, 24, 29, 35\}$, $\boldsymbol{D_s} = 3$, $\boldsymbol{D_e} = 6$, $(\underline{\boldsymbol{L}}, \overline{\boldsymbol{L}}) = (6, 14)$, $\boldsymbol{L} = 10$, $(\underline{\boldsymbol{O}}, \overline{\boldsymbol{O}}) = (1, 3)$. For the sake of simplicity we do not consider the entire set $\boldsymbol{F}$, but a subset of it containing the feasible segments $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$, $\boldsymbol{d}$, $\boldsymbol{e}$, $\boldsymbol{f}$ with lengths respectively $(12, 8, 10, 14, 8, 12)$.

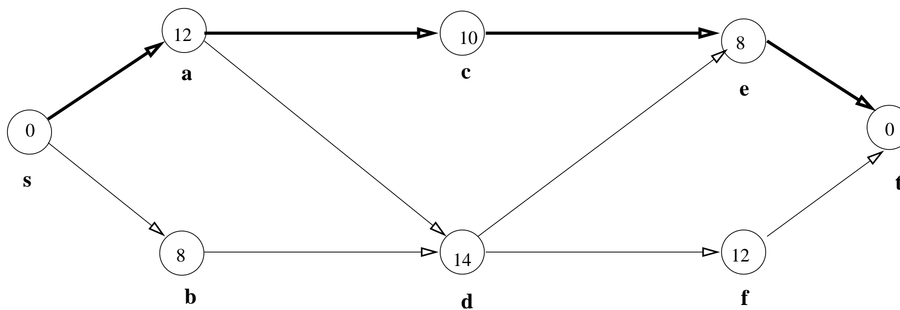


Figure 1.30: The covering graph corresponding to Fig. 1.29. The circles contain the segments lengths. When the algorithm **SPP** is applied to this graph, it finds an optimal path $(\boldsymbol{s} - \boldsymbol{a} - \boldsymbol{c} - \boldsymbol{e} - \boldsymbol{t})$ which contains segments with lengths 12, 10 and 8. The error with respect to 10 equals 2.

For a graph $\boldsymbol{G}$ without circuits, a dynamic programming recurrence gives an algorithm linear in $\boldsymbol{A}$. Let us denote by $\boldsymbol{d_i}$ the length (in sense of max instead of sum) of the shortest path from

s to v_i and let $\Gamma^{-1}(v)$ be the set of all predecessors of v. Then obviously we have :

$$d_i = \min_{v_j \in \Gamma^{-1}(v_i)} \max\{d_j, C_L(v_i)\} \tag{1.82}$$

which leads to the algorithm **SPP** given below.

Algorithm SPP(G)
{Search for the shortest $s - t$ path in **G** using a DP recurrence}
$d_i = \min_{v_j \in \Gamma^{-1}(v_i)} \max\{d_j, C_L(v_i)\} = \max\{d_k, C_L(v_i)\}$
    set $\pi[i] = k$;   (note that $k = \mathbf{arg} \min_{v_j \in \Gamma^{-1}(v_i)} \max\{d_j, C_L(v_i)\}$
  endfor
print : $c \leftarrow \pi[|V|]$;
  while $c > 0$ **do**
    print c;
    $c \leftarrow \pi[c]$;
  endwhile

The **SPP** algorithm takes as input a graph and generates a list of segments. The vertices of **G** are topologically sorted before processed. This means that vertices are arranged on a line in such a way that all arcs are from left to right.

Complexity Analysis

If the graph is represented by the predecessors of each vertex, then the algorithm **SPP** has complexity $O(|A|)$. This follows from the observation that $|A|$ equals the sum of in-degrees of the vertices and from the fact that the complexity of the topological sort is $O(|A|)$ (see [62] chapter 3.3.4).

Remark 5. For our problem the indices of the vertices are naturally induced by appearance of the starting-primers, i.e. the graph is already topologically sorted.

1.5.4 The case when the segment length is unknown

Up to now, the error of the segmentation was measured by the maximal deviation of the segments from a *given* ideal length L. Usually, this length is taken as the middle of the interval $[\underline{L}, \overline{L}]$ (i.e. $L = (\underline{L} + \overline{L})/2$) and is in fact a kind of simplification of the problem. Note for example that on Fig. 1.29 there is a feasible path (a, d, f). The deviation in the lengths of the corresponding segments, $(12, 14, 12)$, is very small. In fact the error with respect to $L = 13$ is one and this path is definitely a good candidate for the LR-PCR technique. However, it cannot be discovered in the framework of the above described model.

For these reasons, in this section, we make a step further toward a quite natural generalization of the problem by considering L as a parameter and looking for L^* such that the best segmentation with respect to it is of minimal error. This will change the original problem $\min_{r\in R} C_L(r) = C_L^*$ to the problem $\min_L \min_{r\in R} C_L(r) = C^*$. In order to put the later one in more tractable form we can exclude L from the model in the following way: For an arbitrary $(s-t)$-path $r = sv_0 \dots v_n t$ let $c^r_{min} = \min_{v_i \in r}\{l(v_i)\}$ and $c^r_{max} = \max_{v_i\in r}\{l(v_i)\}$. (Recall that $l(i)$ is the length of the ith segment). Then the following assertion is true:

Theorem 9. *The minimal error of the segmentation given by a path r is $0.5(c^r_{max} - c^r_{min})$ and it is attained at the length $L^*(r) = 0.5(c^r_{max} + c^r_{min})$.*

If we call $c^r_{max} - c^r_{min}$ *spread* of the path r then according to the theorem an equivalent reformulation of the above-mentioned problem is simply to **find the $(s-t)$-path in G of minimal spread**, which is to find $\triangle^* = c_{max} - c_{min} = \min_{r\in R}\{c^r_{max} - c^r_{min}\}$

Now, let us associate to any vertex $i \neq s$ of the covering graph a set $\mathcal{A}_i$ defined as follows:

$$\mathcal{A}_i = \{(c^r_{\text{min}}, c^r_{\text{max}}) \mid r \text{ being a path from } s \text{ to } i\} \tag{1.83}$$

In this way a list $\mathcal{A}_i$ contains diverse spreads corresponding to *all* possible $(s-i)$-paths. The solution is the minimal spread in the list $\mathcal{A}_t$. An intuitive construction of the lists $\mathcal{A}_i$ is illustrated on Fig. 1.31, while formally they are computed by the recurrences (1.84)

$$\mathcal{A}_i = \begin{cases} \{(\overline{L}, \underline{L})\} & \text{if } i = s \\ \bigcup_{j\in\Gamma^-(t)} \mathcal{A}_j & \text{if } i = t \\ \bigcup_{j\in\Gamma^-(i)} \{l(i) \triangleright I \mid I \in \mathcal{A}_j\} & \text{otherwise} \end{cases} . \tag{1.84}$$

where $e \triangleright (l, u)$ denotes the smallest interval containing e, l and u.

Remark 6. Note that the recurrence (1.84) is correct since the covering graph is without circuits.

Defined in this way, the set $\mathcal{A}_t$ contains the pair (c^r_{min}, c^r_{max}) for *any* r being a path from s to t. If the vertices of the graph are topologically sorted, the recurrence (1.84) can be computed by a single traverse of the graph. The rest of the algorithm is now straightforward: select from $\mathcal{A}_t$ the couple (l, u) with minimal spread, delete vertices with length not in the interval $[l, u]$. *Any* of the $(s-t)$-paths in the reduced graph is optimal.

Complexity Analysis

If the graph is represented by the predecessors of each vertex, then the algorithm **SPP** has complexity $O(|A|)$.

This algorithm is in fact a simple enumeration procedure and the size of the sets $\mathcal{A}_i$ could be very large. For these reasons we introduce an operation (say * operation) which leads to a

significant reduction in these sets size. The * operation retains only those couples which are eligible for continuation, i.e. mutually non inclusive and is more precisely defined as follows:

$$\mathcal{A}^* = \mathcal{A} \setminus \{(l, u) \in \mathcal{A} \mid \exists (l', u') \in \mathcal{A}, [l', u'] \subset [l, u]\} \tag{1.85}$$

The recurrence (1.84) is respectively modified:

$$\mathcal{A}_i^* = \begin{cases} \{(\overline{L}, \underline{L})\} & \text{if } i = s \\ \left(\bigcup_{j \in \Gamma^-(t)} \mathcal{A}_j^* \right)^* & \text{if } i = t \\ \left(\bigcup_{j \in \Gamma^-(i)} \{l(i) \triangleright I \mid I \in \mathcal{A}_j^*\} \right)^* & \text{otherwise} \end{cases} \tag{1.86}$$

The * operation removes from $\mathcal{A}_i$ only pairs (l, u) which are obviously non optimal, because of (1.85), and we therefore *do not* lose solution. The algorithm **SITA**, is described below.

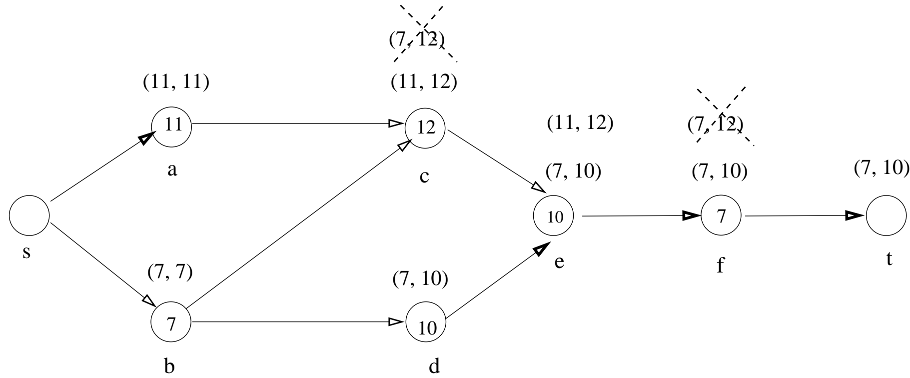


Figure 1.31: This graph illustrates the links between six segments (a,b,c,d,e,f) and the two artificial vertices s and t. The circles contain the lengths of the corresponding segments. Above any vertex i is given the set A_i as defined in (1.83). Note that no one of the elements of the list A_e can be eliminated. Although the element $(7, 10)$ appears less interesting than $(11, 12)$, its elimination leads to a loss of the solution. In contrast, at vertex c we can eliminate $(7, 12)$ since $[11, 12] \subset [7, 12]$ without loosing the solution. Respectively, at vertex f we can eliminate $(7, 12)$ since $[7, 10] \subset [7, 12]$. This elements reduction corresponds to the * operation defined in (1.85).

Algorithm SITA(G)
input: directed graph $G(V, A, C)$;
output: minimal spread (c_{min}, c_{max});
initialization: $A^*_s \leftarrow \{(\overline{L}, \underline{L})\}$;
topological_sort(**G**);
for i=2 **to** $|V|$ **do**
for all $v_j \in \Gamma^-(v_i)$ **do** $A^*_i \leftarrow \left(\bigcup A^*_j\right)^*$ **enddo** ;
enddo ;
$(c_{min}, c_{max}) \leftarrow \textbf{arg} \min_{(l,u)\in A^*_t} (u - l)$;

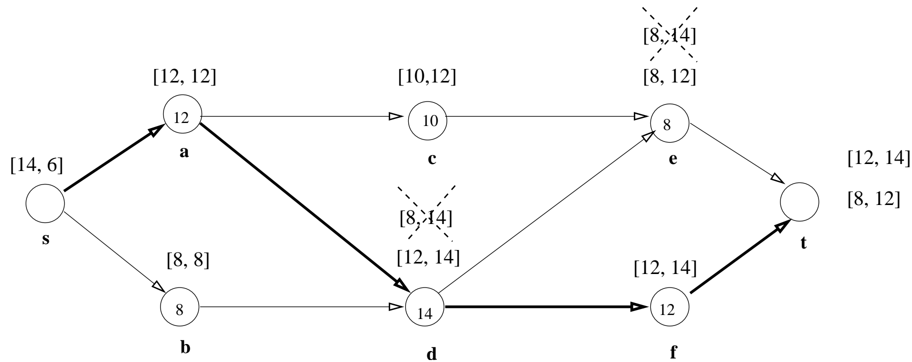


Figure 1.32: The graph illustrates the behavior of the algorithm **SITA** on the problem instance depicted on Fig. 1.29. It finds that an optimal spread of size 2 exists and that the associate length L^* equals 13. The obtained optimal path is $(s - a - d - f - t)$. It contains segments with lengths 12, 14 and 12. Note that run on the same graph, the **SPP** algorithm was misled by the value of L^* and returned a path of spread 4.

Complexity analysis

Let $\mathcal{A}^*$, $\mathcal{B}^*$ be two sets such that any $e \in \mathcal{A}^*$ (resp. any $e \in \mathcal{B}^*$) is a minimum in respect to the inclusion relation. Note that in this case we can define the following total order relation in $\mathcal{A}^*$ (resp. $\mathcal{B}^*$)

$$(l, u) \prec (l', u') \text{ iff } (l < l') \wedge (u < u'). \tag{1.87}$$

If we assume now that $\mathcal{A}^*$ and $\mathcal{B}^*$ are sorted according to (1.87), then applying sort-merge alike algorithm we can realize the operation $(\mathcal{A}^* \bigcup \mathcal{B}^*)^*$ in $O(\max(|\mathcal{A}^*|, |\mathcal{B}^*|))$ operations (interval comparisons). Also note that the result is directly sorted according to (1.87). Using this observation, we can easily prove that the complexity of the **SITA** algorithm is $O(C|A|)$, where C is the maximum number of eligible intervals and all of them are in the interval $[\underline{L}, \overline{L}]$. The inequality $C \leq (\overline{L} - \underline{L})/2$ can be easily verified.

1.5.5 Computational experiments

The **SPP** and **SITA** algorithms are general purpose in sense of underlying graphs, but the primary goals were to use them for the interval graphs discussed in the introduction. That's why all runs are done on graphs, corresponding to domains of varying lengths with uniformly distributed primers. Thus the lack of sufficient biological material is compensated by a randomly generated genomes and despite some mismatches with the reality they could serve well for measuring the computational analysis of their efficiency.

Recalling that the basic parameters are: the *length* D_L of the studied genome domain, the *number* n of primers in this domain, the allowing length from $\underline{L}$ to $\overline{L}$ for the segments and overlap from $\underline{O}$ to $\overline{O}$, it seems more convenient to express the computational complexity of the algorithms as a function of these parameters. Towards this end, the following mixture of probabilistic and deterministic arguments are used below.

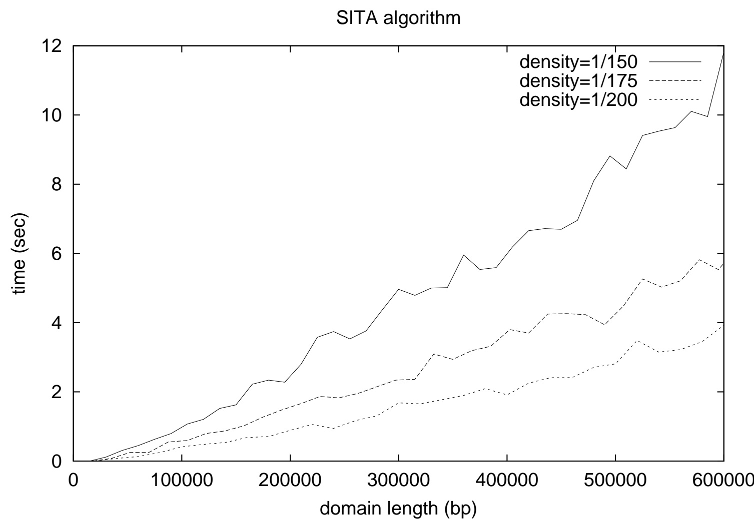


Figure 1.33: Execution time (user time) for the algorithm **SITA**, run on randomly generated genomes of increasing length, (*i.e* primers are uniformly distributed over the domain), but of fixed primer density for each curve. We performed our computational experiments on a Pentium 4 (1.6 Ghz) machine on Linux. Each point on the curves is the average of ten runs.

If we denote by δ the average density of the primers in the domain we obviously have $\delta = \frac{n}{D_L} < 1$. Now, for any starting-primer in the domain we have on average $(\overline{L} - \underline{L})\delta$ compatible primers (i.e. each one of can built a different segment beginning with the same starting-primer). Thus, the total number of segments in the domain is $O((\overline{L} - \underline{L})\delta n)$. Similarly, for a given segment, there are on average $(\overline{O} - \underline{O})\delta$ potential primers to begin a compatible segment; for any of these starting-primers, there are on average $(\overline{L} - \underline{L})\delta$ potential ending-primers. The total number of pairs of compatible segments (remember it corresponds to $|A|$ in the graph terminology) is therefore $O((\overline{L} - \underline{L})^2(\overline{O} - \underline{O})\delta^3 n)$.

Therefore, we obtain that the algorithms proposed in this paper are *linear* in respect to the number of primers in this domain. More precisely, the average bounds for the maximum num-

ber of operations are: $O(|A|) = O((\overline{L} - \underline{L})^2(\overline{O} - \underline{O})\delta^3 n)$ for the **SPP** algorithm and $O(C|A|) = O((\overline{L} - \underline{L})^3(\overline{O} - \underline{O})\delta^3 n)$ for **SITA** algorithm. As we already mentioned $C = (\overline{L} - \underline{L})/2$ is a theoretical upper bound for the lengths of the lists associated with the vertices. It was quite intricated (but not unexpected) to observe how huge is the gap between this bound and the real ones (less then 10 in all runs depicted on Fig. 1.33). One can easily show that this bound is achieved for example on a graph of 11 vertices and costs from $\underline{L} = 10$ to $\bar{L} = 20$. If the pairs entering the vertex of cost 15 are $[10, 16], [11, 17], [12, 18], [13, 19], [14, 20]$ then all ($C = (\overline{L} - \underline{L})/2$) of them will survive the seep of * operation. But whatever is the graph with such costs there is no corresponding nucleotide sequence. In this sense, this theoretical upper estimate for C is indeed very pessimistic and unlikely to be reached in real life. For instance, real life values for the parameters are: 1 Mbp for the length of the domains; 5000 to 15000 for the number of primers; 10 Kbp $\pm$ 1000 for the length of the segments; 1 Kbp $\pm$ 500 for the overlap and $\delta < 10^{-2}$ (density). In all our runs with these real life parameters we observed that $C < 10$. In practice, the algorithm is fast and can segment whole genomes in very short time.

1.5.6 Conclusion

In this paper we pose and answer two questions about covering a genome by a sequence of overlapping segments. The quality of the covering is measured according to two criteria:

- the maximal deviation of the segment's length from a given length is minimal;
- the maximal spread between the longest and the shortest segment is minimal.

We propose two algorithms: **SPP** for solving the former problem and **SITA** for solving the later one. They take as input the set of starting and ending-primers, the genome domain to split into segments, and the parameters corresponding to the segment length and the overlap size. The result is an optimal list of segments satisfying the corresponding criterion. The algorithm **SITA** has been implemented using the Objective CAML language. It is part of a package called **GenoFrag** which also includes another software, jointly developed with the INRA[13] microbiology team, to generate the set of primers[14]. Actually, this software acts as a pipeline of filters fed by a complete genome: each filter, dedicated to some specific features, discard all the primers which do not satisfy user-specified constraints–GC-content, thermodynamic stability, hairpin loop size, etc. (see [63] for more details).

We tested the two algorithms on the *Staphylococcus Aureus* [59], a Gram-positive pathogenic bacterium. Primers were generated from the N315 *S. Aureus* strain using different filters. The largest domain represents 1.3 Mbp with an average primer density of 0.006. The computation time for generating the optimal list of overlapping segments on a standard Linux machine (PC running at 1.6 Ghz with 256 Mbytes of memory) does not exceed one minute. This is a very

[13] Laboratoire d'hygiène alimentaire,UMR STLO, INRA, ENSAR, 65 rue de Saint Brieuc, 35042 Rennes, France

[14] **GenoFrag** contains also a software for solving the problem when the length L is given. The complexity of the underlying algorithm is slightly weaker (logarithmic factor) than **SPP**, since it focuses on graphs with circuits. We do not present it here for the lack of space. The interested reader can find its description in [64].

fast process compared to the space of all potential solutions. Furthermore, as explained in the previous section, the complexity of the algorithms is linear in respect to the genome size.

Thanks to this property, the use of these algorithms is definitely not restricted to small genomes, but can be applied to significantly larger ones.

Bibliography

[1] T. Akutsu and S. Miyano. On the approximation of protein threading. *Theoretical Computer Science*, 210:261–275, 1999.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215:403–410, 1990.

[3] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res.*, 25:3389–3402, 1997.

[4] R. Andonov, S. Balev, and N. Yanev. Protein threading: From mathematical models to parallel implementations. *INFORMS Journal on Computing*, 16(4), 2004.

[5] S. Balev. Solving the protein threading problem by lagrangian relaxation. In *Proceedings of WABI 2004: 4th Workshop on Algorithms in Bioinformatics*, LNCS/LNBI. Springer-Verlag, 2004. To appear.

[6] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne. The protein data bank. *Nucleic Acids Research*, 28:235–242, 2000.

[7] C. Branden and J. Tooze. *Introduction to protein structure*. Garland Publishing, 1999.

[8] C. L. Brooks, M. Karplus, and B. M. Pettitt. *Proteins: A theoretical perspective of dynamics, structure, and thermodynamics*. John Wiley and Sons, 1990.

[9] C. Chothia. Proteins. one thousand families for the molecular biologist. *Nature*, 357:543–544, 1992.

[10] G. Cornuéjols, G. L. Nemhauser, and L. A. Wolsey. The uncapacitated facility location problem. In P Mirchandani and R. Francis, editors, *Discrete location theory*, pages 119–171. John Wiley and Sons, 1990.

[11] A. Godzik, A. Kolinski, and J. Skolnik. Topology fingerprint approach to the inverse protein folding problem. *J. Mol. Biol.*, 227:227–238, 1992.

[12] P. Veber, N. Yanev, R. Andonov, V. Poirriez. Optimal protein threading by cost-splitting *Lecture Notes in Bioninformatics*, 3692, pp.365-375, 2005

[13] H. J. Greenberg, W. E. Hart, and G. Lancia. Opportunities for combinatorial optimization in computational biology. *INFORMS Journal on Computing*, 16(3), 2004.

[14] T. Head-Gordon and J. C. Wooley. Computational challenges in structural and functional genomics. *IBM Systems Journal*, 40:265–296, 2001.

[15] N. Krasnogor, W. E. Hart, J. Smith, and D. A. Pelta. Protein structure prediction with evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1596–1601. Morgan Kaufmann, 1999.

[16] R.H. Lathrop. The protein threading problem with sequence amino acid interaction preferences is NP-complete. *Protein Engineering*, 7(9):1059–1068, 1994.

[17] R.H. Lathrop, R.G. Rogers Jr., J. Bienkowska, B.K.M. Bryant, L.J. Buturovic, C. Gaitatzes, R. Nambudripad, J.V. White, and T.F. Smith. Analysis and algorithms for protein sequence-structure alignment. In S.L. Salzberg, D.B. Searls, and S. Kasif, editors, *Computational Methods in Molecular Biology*, chapter 12, pages 227–283. Elsevier Science, 1998.

[18] R.H. Lathrop and T.F. Smith. Global optimum protein threading with gapped alignment and empirical pair potentials. *J. Mol. Biol.*, 255:641–665, 1996.

[19] C.E. Lawrence, S. F. Altschul, J. S. Boguski, A. F. Neuwald, and J. C. Wootton. Detecting subtle sequence signals: a gibbs sampling strategy for multiple alignment. *Science*, 262:208–214, 1993.

[20] T. Lengauer. Computational biology at the beginning of the post-genomic era. In R. Wilhelm, editor, *Informatics: 10 Years Back – 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 341–355. Springer-Verlag, 2001.

[21] A. Marin, J. Pothier, K. Zimmermann, and J.-F. Gibrat. FROST: a filter-based fold recognition method. *Proteins*, 49(4):493–509, 2002.

[22] A. Marin, J. Pothier, K. Zimmermann, and J.-F. Gibrat. Protein threading statistics: an attempt to assess the significance of a fold assignment to a sequence. In I. Tsigelny, editor, *Protein structure prediction: bioinformatic approach*. International University Line, 2002.

[23] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. Wiley, 1988.

[24] C. A. Orengo and W. R. Taylor. A rapid method for protein structure alignment. *J. Theor. Biol.*, 147:517–551, 1990.

[25] C.A Orengo, T. D. Jones, and J. M. Thornton. Protein superfamilies and domain superfolds. *Nature*, 372:631–634, 1994.

[26] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci. USA*, 85:2444–2448, 1988.

[27] F. Plastria. Formulating logical implications in combinatorial optimization. *Eur. J. Oper. Res.*, 140:338–353, 2002.

[28] J. Pley, R. Andonov, J.-F. Gibrat, A. Marin, and V. Poirriez. Parallèlisation d'une méthode de reconnaissance de repliements de protéines (frost). In *Proceedings des Journées Ouvertes Biologie Informatique Mathématiques*, pages 287–288, 2002.

[29] A. A. Rabow and H. A. Scheraga. Improved genetic algorithm for the protein folding problem by use of a cartesian combination operator. *Protein Science*, 5:1800–1815, 1996.

[30] R. B. Russel and G. J. Barton. Structural features can be unconserved in proteins with similar folds. an analysis of side-chain to side-chain contacts secondary structure and accessibility. *J. Mol. Biol.*, 244:332–350, 1994.

[31] D. Sankoff and J. B. Kruskal, editors. *Time wraps, string edits and macromolecules*. Addison-Wesley Reading, 1983.

[32] J. C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997.

[33] J. D. Szustakowski and Z. Weng. Protein structure alignment using a genetic algorithm. *Proteins*, 38(4):428–440, 2000.

[34] W. R. Taylor and C. A. Orengo. Protein-structure alignment. *J. Mol. Biol.*, 208:1–22, 1989.

[35] J. Xu. Speedup LP approach to protein threading via graph reduction. In *Proceedings of WABI 2003: Third Workshop on Algorithms in Bioinformatics*, volume 2812 of *Lecture Notes in Computer Science*, pages 374–388. Springer-Verlag, 2003.

[36] J. Xu, M. Li, G. Lin, D. Kim, and Y. Xu. Protein structure prediction by linear programming. In *Proceedings of The 7th Pacific Symposium on Biocomputing (PSB)*, pages 264–275, 2003.

[37] J. Xu, M. Li, G. Lin, D. Kim, and Y. Xu. RAPTOR: optimal protein threading by linear programming. *Journal of Bioinformatics and Computational Biology*, 1(1):95–118, 2003.

[38] Y. Xu and D. Xu. Protein threading using PROSPECT: design and evaluation. *Proteins: Structure, Function, and Genetics*, 40:343–354, 2000.

[39] Y. Xu, D. Xu, and E. C. Uberbacher. An efficient computational method for globally optimal threading. *Journal of Computational Biology*, 5(3):597–614, 1998.

[40] N Yanev. Solution of a simple plant-location problem. *USSR Comput. Math. Math. Phys.*, 21, 1981.

[41] N. Yanev and R. Andonov. The protein threading problem is in P? Research Report 4577, Institut National de Recherche en Informatique et en Automatique, 2002.

[42] N. Yanev and R. Andonov. Solving the protein threading problem in parallel. In *HiCOMB 2003 – Second IEEE International Workshop on High Performance Computational Biology*, 2003.

[43] R. Andonov, S. Balev and N. Yanev, Protein Threading Problem: From Mathematical Models to Parallel Implementations, INFORMS Journal on Computing, 2004, 16(4), pp. 393-405

[44] Stefan Balev, Solving the Protein Threading Problem by Lagrangian Relaxation, WABI 2004, 4th Workshop on Algorithms in Bioinformatics, Bergen, Norway, September 14 - 17, 2004

[45] D. Fischer, http://www.cs.bgu.ac.il/ dfishcer/CAFASP3/, Dec. 2002

[46] A. Caprara, R. Carr, S. Israil, G. Lancia and B. Walenz, 1001 Optimal PDB Structure Alignments: Integer Programming Methods for Finding the Maximum Contact Map Overlap *Journal of Computational Biology*, 11(1), 2004, pp. 27-52

[47] Ilog cplex. http://www.ilog.com/products/cplex

[48] R. Lathrop, The protein threading problem with sequence amino acid interaction preferences is NP-complete, Protein Eng., 1994; 7: 1059-1068

[49] A. Marin, J.Pothier, K. Zimmermann, J-F. Gibrat, FROST: A Filter Based Recognition Method, Proteins, 2002 Dec 1; 49(4): 493-509

[50] G. Lancia. Integer Programming Models for Computational Biology Problems. *J. Comput. Sci. & Technol.*, Jan. 2004, Vol. 19, No.1, pp.60-77

[51] V. Poirriez, A. Marin, R. Andonov, J-F. Gibrat. FROST: Revisited and Distributed, HiCOMB 2005, Fourth IEEE International Workshop on High Performance Computational Biology, April 4, 2005, Denver, CO

[52] R: A language and environment for statistical computing, R Foundation for Statistical Computing, Vienna, Austria, 2004, http://www.R-project.org

[53] J.C. Setubal, J. Meidanis, Introduction to computational molecular biology, 1997, Chapter 8: 252-259, Brooks/Cole Publishing Company, 511 Forest Lodge Road, Pacific Grove, CA 93950

[54] N. Yanev and R. Andonov, Parallel Divide and Conquer Approach for the Protein Threading Problem, Concurrency and Computation: Practice and Experience, 2004; 16: 961-974

[55] G. Lancia, Integer programming models for computational biology problems, J. Comput. Sci. & Technol. 19 (2004) 60–77.

[56] A. Caprara, G. Lancia, B. Carr, B. Walenz, S. Istrail, 1001 optimal PDB structure alignments: Integer programming methods for finding the maximum contact map overlap, Journal of Computational Biology 11 (2004) 27–52.

[57] V. Poirriez, R. Andonov, A. Marin, J.-F. Gibrat, Frost: Revisited and distributed, In IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 7, page 200a. IEEE Computer Society, 2005.

[58] N. Yanev, P. Veber, R. Andonov and S. Balev. Lagrangian approaches to a class of matching problems *INRIA-00090635-v2*, 2006 and in *Journal of computational and applied mathematics*, 2007 (to appear)

[59] Kuroda et al., Whole genome sequencing of methicillin-resistant Staphylococcus aureus, *The Lancet* 2001; 357:1225-1240.

[60] Ohnishi M, Terajima J, Kurokawa K, Nakayama K, Murata T, Tamura K, Ogura Y, Watanabe H, Yayashi T. Genomic diversity of enterohemorrhagic Escherichia coli O157 revealed by whole genome PCR scanning, *Proc. Natl. Acad. Sci. USA* 2002; (**26**): 17043-17048.

[61] Christofides N. *Graph Theory. An algorithmic approach*. Academic Press, London, 1975.

[62] Gondran M, Minoux M. *Graphs and Algorithms*. John Willey & Sons, 1984

[63] Ben Zakour N, Gautier M, Andonov R, Lavenier D, Cocher M, Veber P, Sorokin A, Le Loir Y. GenoFrag: software to design primers optimized for whole genome scanning by long-range PCR amplification. *Nucleic Acids Research* 2004; (**32**)1:17-24.

[64] Andonov R, Lavenier D, Yanev N, Veber P. Combinatorial approaches for segmenting bacterium genomes. *INRIA Research report RR-4853* 2003.

[65] D. Goldman, C.H. Papadimitriu, and S. Istrail. Algorithmic aspects of protein structure similarity *FOCS 99: Proceedings of the 40th annual symposium on foundations of computer science* IEEE Computer Society,1999

[66] J. Xu, F. Jiao, B. Berger. A parametrized Algorithm for Protein Structer Alignment *RECOMB 2006, Lecture Notes in Bioninformatics*, 3909,pp. 488-499,2006

[67] I. Halperin, B. Ma, H. Wolfson, et al. Principles of docking: An overview of search algorithms and a guide to scoring functions *Proteins Struct. Funct. Genet.*, 47, 409-443,2002

[68] D. Goldman, S. Israil, C. Papadimitriu. Algorithmic aspects of protein structure similarity *IEEE Symp. Found. Comput. Sci.* 512-522,1999

[69] M. Garey, D. Johnson. Computers and Inractability: A Guide to the Thoery of NP-completness *Freeman and company*, New york, 1979

[70] D. Pelta, N. Krasnogor, C. Bousono-Calzon, J. L. Verdegay, J. Hirst, E. Burke. A fuzzy sets based generalization of contact maps for the overlap of protein structures *Journal of Fuzzy Sets and Systems,* 152(2):103-123, 2005.

Chapter 2

Tiling problems

Tiling is a technique used to group elemental computation points so as to increase computation granularity and thereby to reduce computation time when dealing with distributed memory parallel computers. This technique is restricted to perfect loop nests with uniform dependence as in the example below.

for $i = 2$ to N do
for $j = 2$ to N do
for $k = 2$ to $N - 1$ do
$a(i, j, k) = a(i - 1, j, k) + a(i, j - 1, k) + a(i, j, k - 1) + a(i - 1, j - 1, k + 1)$

This is an example of a perfect loop nest of depth $n = 3$ on the computational domain (iteration space) containing all integer points (i, j, k) $2 \leq i \leq N, 2 \leq j \leq N, 2 \leq k \leq N - 1$, called nodes. We are not interested in the detail of the computation performed by the statements within the nest. Of interest here are the four dependence vectors $d_1 = (1, 0, 0), d_2 = (0, 1, 0), d_3 = (0, 0, 1), d_4 = (1, 1, -1)$ which we capture in the dependence matrix $D = (d_1, d_2, d_3, d_4)$ column-wise. A tile (supernode) is a set of nodes defined as a n-dimensional parallelepiped box defined by cutting hyperplanes as:
Consider n linearly independent vectors h_i. A tile is translated copy of the canonical tile defined as follows: a node p belongs to the tile if and only if:
$0 \leq h_i.p < \beta_i, i = 1, 2, \ldots, n$
The other definition is by change of basis. Consider n linearly independent vectors p_i to define the edges of the tile. The tile is then the set of nodes whose coordinates are positive and strictly less than one in the basis defined by p_i's.
If P is the matrix whose column vectors are p_i and H is the matrix with rows h_i then $P^{-1} = H$.
Tiling is simply paving the iteration space with translated copies of the canonical tile defined above. Classical constraints on tiles are the following:
Tiles are bounded This means H is non singular.
Tiles are identical by translation This constraint is imposed to allow for automatic code generation. To ensure this constraint, we impose that the edges of the tile have integral components,

i.e. $\boldsymbol{P}$ is an integral matrix.
Tiles are atomic Each tile is a unit of computation: all synchronization points are beginnings and ends of tiles. The order on tiles must be compatible with the order of nodes. This is satisfied if all $\boldsymbol{d}_i$ belong to the cone defined by the edges of a tile. This condition is mathematically expressed as $\boldsymbol{Hd} \geq \boldsymbol{0}$.
Tiling is a good paradigm for parallel computers with distributed memory. In these multicomputers, the relatively high communication startup costs makes frequent communication very expensive. Tiling can be used to reduce the communication overhead between processors by grouping loop iterations into tiles so that communication takes place per each tile instead of per each node. But in general this is not only the communication time to be taken into account. The problem is to choose the best tile size and shape to minimize the execution time of a loop nest, called *time-minimal problem* , which is a difficult discrete non-linear optimization problem. In the following two sections, we present a close-form optimal tiling for the most used tile shapes: orthogonal and oblique (in two dimensional space).

2.1 Optimal Orthogonal Tiling

2.1.1 Introduction

Tiling the iteration space [17, 7, 14] is a common method for improving the performance of loop programs on distributed memory machines. It may be used as a technique in parallelizing compilers (see [6, 13] where it is also called coarse-grain pipelining), as well as in performance tuning of parallel codes by hand (see also [10, 15]). A *tile* in the iteration space is a (hyper) parallelepiped shaped collection of iterations to be executed as a single unit, with the communication and synchronization being done only once per tile. Typically, communication are performed by send/receive calls and also serve as synchronization points. The code for the body contains no communication calls.

The *tiling problem* can be broadly defined as the problem of choosing the tile parameters (notably the shape and size) in an optimal manner. It may be decomposed into two subproblems: *tile shape optimization* [5], and *tile size optimization* [2, 6, 9, 13] (some authors also attempt to resolve both problems under some simplifying assumptions [14, 15]). By its very nature, such a two step approach may not be globally optimal, but is often used in order to make the problem tractable.

In this paper, we address the tile size problem, which, for a given tile shape, seeks to choose the size (length along each dimension of the hyper parallelepiped) so as to minimize the total execution time. In its most general formulation, this is a hard discrete non-linear optimization problem, and there is currently no solution. However, optimal solutions can be found analytically under certain restrictions.

We assume that the dependencies are uniform, the iteration space (domain) is an n-dimensional hyper-rectangle, and the tile boundaries are parallel to the domain boundary (this is called *orthogonal tiling*). A sufficient condition for this that in all dependence vectors, all non-zero terms have the same sign. Whenever orthogonal tiling is possible, it leads to the simplest form of code;

indeed most compilers do not even implement any other tiling strategy.

Our approach is based on the two step model proposed by Andonov and Rajopadhye [2], and who addressed the 2-dimensional case, and which was later extended to 3 dimensions by Andonov et al. [3]. In this model we first abstract each tile by two simple parameters: tile period, P_t and inter-tile latency, L. We formulate and partially resolve the corresponding abstract optimization problem. We then "instantiate" the abstract model by accounting for specific architectural and program features, and analytically solve the resulting non-linear optimization problem, yielding the desired tile size.

The extension to n dimensions was an open question and which is resolved in this paper. We first extend and generalize the Andonov-Rajopadhye model to the case where an n-dimensional iteration space is implemented on a k-dimensional (hyper) toroid (for any $1 \leq k \leq n-1$). We then instantiate the functions for the tile period and latency, and solve the corresponding optimization problem. We also consider a more general form of the specific functions for L and P_t. These functions are general enough to not only include a wide variety of machine and program models, but also be used with the BSP model [11, 16], which is gaining wide acceptance as a well founded theoretical model for developing architecture independent parallel programs.

The remainder of this paper is organized as follows. In the following section we develop the model and formulate the abstract optimization problem. In Section 2.1.3 we instantiate the model for specific machine and program parameters, and show how the model is general enough to include most of the models used in the literature. In Section 2.1.4 we resolve the problem for the simplified HKT model which assumes that the communication cost is constant, independent of the message volume (this is very similar to the particular case of the BSP model where the network bandwidth is very high). Next, in Section 2.1.5 we resolve the more general optimization problem. We present our conclusions in Section 2.1.6.

2.1.2 Abstract Model Building

We now develop an analytical performance model for the running time of the tiled program. We introduce the notation required as we go along. The original iteration space is an $N_1 \times N_2 \times \ldots \times N_n$ hyper-rectangle, and it has (at least n linearly independent) dependency vectors, $d_1, d_2, \ldots$. The nonzero elements of the dependency vectors all have the same sign (say positive, without loss of generality). Hence, orthogonal tiling is possible (does not induce any cyclic dependencies between the tiles). Let the tiles be $x_1 \times x_2 \times \ldots \times x_n$ hyper-rectangles, and let $n_i = \frac{N_i}{x_i}$ be the number of tiles in the i-th dimension. The *tile graph* is the graph where each node represents a tile and each arc represents a dependency between tiles, and can be modeled by a uniform recurrence equation [8] over an $n_1 \times n_2 \times \ldots \times n_n$ hyper-rectangle. It is well known that if the x_i's are large as compared to the elements of the dependency vectors[1], then the dependencies between the tiles are *unit vectors* (or binary linear combinations thereof, which can be neglected for analysis purposes without any loss of generality). A tile can be identified by an index vector $z = [z_1, \ldots, z_n]^{\mathrm{T}}$. Two tiles, z and z', are said to be *successive*, if $z - z'$

[1]In general this implies that the feasible value of each x_i is bounded from below by some constant. For the sake of clarity, we assume that this is 1.

is a unit vector, i.e, one of them depends directly on the other. A simple analysis [7] shows that the earliest "time instant" that tile z can be executed (counting one tile as one macro step) is $t_z = z_1 + \ldots + z_n$.

We map the tiles to $p_1 \times p_2 \times \ldots \times p_k$ processors, arranged in a k dimensional hyper-toroid[2] (for $k < n$). The mapping is by projection onto k of the n canonical axes. To visualize this mapping, first consider the case when $k = n - 1$. Assume (without loss of generality) that the allocation of tiles to processors is by projection along the n-th dimension, i.e., tile z is executed by processor $[z_1, \ldots, z_{n-1}]^{\mathrm{T}}$. This yields a (virtual) array of $n_1 \times n_2 \times \ldots \times n_{n-1}$ processors, each one executing a *macro column* of the tile graph (i.e., all the tiles in the n-th dimension). However, we do not insist that $n_i = p_i$, otherwise we would (unnecessarily) constrain x_i to be $\frac{N_i}{p_i}$, so this array is emulated by our $p_1 \times p_2 \times \ldots \times p_{n-1}$ hyper-toroid by using multiple passes. Note that there will be $\frac{n_i}{p_i}$ passes in the i-th dimension. The case when $k < n$ is modeled by simply letting $p_{k+1} = \ldots = p_{n-1} = 1$.

The *period*, P_t of a tile is defined as the time between executions of corresponding instructions in two successive tiles in *the same* macro column (i.e., they are mapped to the same processor). The *latency*, L between tiles is defined to be the time between executions of corresponding instructions in two successive tiles mapped to *different* processors. Depending on the volume of the data transmitted and the nature of the program dependencies, the latency may be different for different dimensions of the hyper-toroid, and we use L_i to denote the latency in the i-th dimension, for $i - 1 \ldots k$ (in dimensions $k + 1$ to $n - 1$, successive tiles are executed on the same processor, and the notion of latency is moot).

Now, to model the running time of this implementation, we assume that by the time the first processor, $[1, \ldots, 1]^{\mathrm{T}}$ finishes computing its macro column, at least one of the "last" processors (i.e., one of $[1, \ldots, p_i, \ldots, 1]^{\mathrm{T}}$, for $1 \leq i \leq k$) has finished its first tile. In this case, processor $[1, \ldots, 1]^{\mathrm{T}}$ can *immediately* start another pass. If this were not the case, the first (and hence *all*) processors would be idle between passes, and this would lead to a sub-optimal solution. This was formally proved for the 2-dimensional case by Andonov and Rajopadhye [2], and the same argument can be easily extended. Let $W = N_1 N_2 \ldots N_n$ denote the total computation volume, $v = x_1 x_2 \ldots x_n$ be the tile volume and $P = p_1 p_2 \ldots p_k$ be the total number of processors. The total number of tiles is $n_1 n_2 \ldots n_n = \frac{W}{v}$. Let $\widetilde{p_i}$ denote $p_i - 1$, $\widetilde{P_k} = \sum_{i=1}^{k} \widetilde{p_i}$ and $v_{\max} = (\prod_{i=1}^{k} N_i)/P$.

Let us first analyze a single pass. Each processor must execute n_n tiles, and this takes $n_n P_t$ time. However, the last processor (i.e., the one with coordinates $[p_1, p_2, \ldots p_k]^{\mathrm{T}}$) cannot start because of the dependencies of the tile graph. Indeed, processor $[p_1, 1 \ldots, 1]^{\mathrm{T}}$ can only start at time $(p_1 - 1)L_1$, processor $[p_1, p_2 \ldots, 1]^{\mathrm{T}}$, at time $(p_1 - 1)L_1 + (p_2 - 1)L_2)$, and hence processor $[p_1, p_2 \ldots, p_k]^{\mathrm{T}}$ can only start its first pass at time $\sum_{i=1}^{k} \widetilde{p_i} L_i$.

There are $\frac{W}{v}$ tiles, of which $P n_n$ are executed in each pass, and hence there are $\frac{W}{P v n_n}$ passes.

[2]This is just an abstract machine architecture. Our machine model is independent of the topology, since we will later assume that communication time is independent of distance and/or contention.

Because a new pass can begin as soon as the first processor is done with its macro column, the last pass can start at time $\left(\frac{W}{Pvn_n} - 1\right) P_t n_n$. Thus, the last processor starts executing its *last* macro column at time instant $\left(\dfrac{W}{Pvn_n} - 1\right) P_t n_n + \sum_{i=1}^{k} \widetilde{p}_i L_i$. It takes another $P_t n_n$ time, and hence the total running time is as follows.

$$T(x_1, \ldots x_n) = \frac{WP_t}{Pv} + \sum_{i=1}^{k} \widetilde{p}_i L_i \tag{2.1}$$

Our optimization problem is formulated as follows.
Prob. 1: Minimize (2.1) in the feasible space, R given below (recall that the lower bounds may be other than 1, based on the dependencies of the original recurrence).

$$R = \left\{ [x_1 \ldots x_n]^{\mathrm{T}} \in Z^n \mid 1 \leq x_i \leq u_i \right\} \tag{2.2}$$

where $u_i = \frac{N_i}{p_i}$ for $i = 1 \ldots k$, and $u_i = N_i$ for $k < i \leq n$.

2.1.3 Machine and Program Specific Model

We now "instantiate" **Prob. 1** for a specific program and machine architecture. The code executed for a tile is the standard loop:

```
repeat
 receive(v1); receive(v2), ...,receive(vk) ;
 compute(body);
 send(v1); send(v2), ..., send(vk) ;
end
```

where we denote by `vi` the message transmitted in the i-th dimension. We will now determine P_t and L_i. Our development is based on Andonov & Rajopadhye [1], and uses standard assumptions about low level behavior of the architecture and program [4]. The sole distinction for the k-dimensional machine is that each tile now makes k systems calls to send (and receive) messages. A tile depends directly on its n neighbors in *each* dimension. The volume of data transfer along the i-th dimension is proportional to the (hyper) surface of the tile in that dimension, i.e., $\prod_{j \neq i} x_j = \dfrac{v}{x_i}$. In the first k dimensions, this corresponds to an inter-processor communication, whereas in the dimensions $k + 1 \ldots n$, the transfer is achieved through local memory. Hence the period of a tile can be written as follows.

$$P_t = k(\beta_r + \beta_s) + \left(2\tau_c v \sum_{i=1}^{k} \frac{1}{x_i} \right) + \alpha v \tag{2.3}$$

Here β_s (resp. β_r) is the overhead of the `send` (resp. `receive`) system call, τ_c is the time (per byte) to copy from user to system memory, α is the computation time for a single instance

of the loop body (we neglect the overhead of setting up the loop for each tile, as well as cache effects), and $1/\tau_t$ is the network bandwidth. Similarly the latency is given by (see [2] for details)

$$\begin{aligned} L_i &= P_t + \tau_t \frac{v}{x_i} - k\beta_r \\ &= k\beta_s + \left(2\tau_c v \sum_{i=1}^{k} \frac{1}{x_i}\right) + \alpha v + \tau_t \frac{v}{x_i} \end{aligned} \quad (2.4)$$

Note that the β_r's are subtracted because the `receive` occurs on a *different* processor, and when the sender and receiver are properly synchronized, the calls are overlapped. Substituting in Eqn. (2.1) and simplifying, we obtain

$$\begin{aligned} T_k(\vec{x}) &= \alpha v \widetilde{P_k} + 2\tau_c \widetilde{P_k} v \sum_{i=1}^{k} \frac{1}{x_i} + \tau_t v \sum_{i=1}^{k} \frac{\widetilde{p_i}}{x_i} + \frac{2\tau_c W}{P} \sum_{i=1}^{k} \frac{1}{x_i} + \\ &+ k(\beta_r + \beta_s) \frac{W}{Pv} + k\beta_s \widetilde{P_k} + \frac{\alpha W}{P} \end{aligned} \quad (2.5)$$

Simplifying assumptions and particular cases

The model of (2.5) is very general. One may want to specialize it for a number of reasons — say rendering the final optimization problem more tractable, or modeling a certain class of architectures or computations. It turns out that many of these simply consist of choosing the parameters appropriately in the above function.

The *HTK* model, first used by Hiranandani et al. [6], corresponds to setting $\beta_r = \tau_c = \tau_t = 0$ (a slightly more general version consists of letting β_r be nonzero). This model assumes that the computation cost is independent of the message size, but is dominated by the startup time(s) β_s (and β_r). At first sight this may seem an oversimplification. However, in addition to making the mathematical problem more tractable, it is not far from the truth, as corroborated by other authors [12, 13]. Indeed, experimental as well as analytic evidence [1] shows that on machines such as the Intel Paragon the more accurate models yield *no observable difference* in the predictions. With $\tau_c = \tau_t = 0$, we obtain the ***HKT cost function***:

$$T_k(\vec{x}) = \frac{\alpha W}{P} + k(\beta_r + \beta_s) \frac{W}{Pv} + (\alpha v + k\beta_s) \widetilde{P_k} \quad (2.6)$$

The *BSP* model [11, 16] has been proposed as a formal model for developing architecture independent parallel programs. It is a bridge between the PRAM model which is general but somewhat unrealistic, and machine-specific models which lead to lack of portability and predictability of performance. Essentially, the computation is described in terms of a sequence of "super-steps" executed in parallel by all the processors. A super-step consists of (i) some local computation, (ii) some communication events *launched* during the super-step, and (iii) a synchronization which ensures that the communication events are *completed.* The time for a super-step is the sum of the times for each of the above activities. This is very similar to our tile

model: indeed, if we simply set $\tau_t = \beta_r = 0$ and $\beta_s = \frac{\beta}{k}$ (β is the BSP synchronization cost) we obtain the running time of the program under the BSP model. With this simplification, we obtain the ***BSP cost function*** as follows.

$$T_k(\vec{x}) = \frac{\alpha W}{P} + \frac{\beta W}{Pv} + (\alpha v + \beta)\widetilde{P_k} + 2\tau_c \left(\widetilde{P_k} v + \frac{W}{P}\right) \sum_{i=1}^{k} \frac{1}{x_i} \tag{2.7}$$

In the BSP model, the communication startup cost is replaced by the synchronization cost. However, in our general cost function (2.5) we incur the startup cost k times. As a result, if we take a particular case of the BSP model where the network bandwidth is extremely high (and it is the synchronization cost which dominates the communication time), then this ***high bandwidth BSP cost function*** is not exactly the same as the HKT model (2.6) but given as follows

$$T_k(\vec{x}) = \frac{\alpha W}{P} + \frac{\beta W}{Pv} + (\alpha v + \beta)\widetilde{P_k} \tag{2.8}$$

With some other simple modifications, our cost function can also model the overlap of communication and computation, which is often used as a performance tuning strategy. This is not detailed here due to space constraints.

2.1.4 Solution for the simple models

In this section, we will focus only on the HTK and the high bandwidth BSP models. our main results are that the optimal tile volume can be determined as a closed form solution, that the optimal virtual architecture is an $n-1$ dimensional hyper toroid. These results serve two important purposes, in spite of the apparent simplicity of the model. First, they are valid for a number of machines where the communication latency and network bandwidth are both relatively high (such as the Intel paragon, and a number of similar machines as well as networks of workstations). Second, they give a good indication of our solution method for the more general results.

The first observation that we can make from (2.6) is that the running time depends only on the *tile volume* and not on the specific tile size parameters – for a given tile volume, any set of values for x_i (provided that they yield a feasible solution) will give the same running time. Hence the problem to solve is to determine the optimal volume. It is easy to see that (2.6) is a strongly convex function of the form $T(x) = \frac{A}{x} + Bx + C$, which attains its optimal value of $C + 2\sqrt{AB}$ at $x = \sqrt{\frac{A}{B}}$. Let us define

$$\widetilde{v} = \sqrt{\frac{k(\beta_r + \beta_s)W}{P\alpha\widetilde{P_k}}} \tag{2.9}$$

$$\widetilde{T_k} = \frac{\alpha W}{P} + k\beta_s\widetilde{P_k} + 2\sqrt{\frac{k\alpha(\beta_r + \beta_s)W\widetilde{P_k}}{P}} \tag{2.10}$$

The optimal solution will be as given above if $\widetilde{v}$ is a feasible tile volume. Now, observe that each x_i is bounded from above by $\frac{N_i}{p_i}$, and hence $v \le \frac{W}{P}$. Asymptotically, $W \gg P$, and clearly $1 \le \widetilde{v} \le \frac{W}{P}$. Hence we have the following result.

Theorem 10. *The optimal tile volume and the corresponding running time for the* HKT *model are given by by (2.9-2.10).*

So far, we have assumed that k is a fixed constant as are each of the p_i's. In practice, we typically have P processors, and the values of each p_i are not specified. Let us therefore consider two different P-processor architectures, one corresponding to a k-dimensional torus of $p_1 \dots p_k$ processors and another a $k-1$ dimensional torus of $p'_1 \dots p'_{k-1}$ processors. The optimal running time on the first architecture is given by (2.10) and on the latter, it is

$$\widetilde{T_{k-1}} = \frac{\alpha W}{P} + (k-1)\beta_s \widetilde{P'_{k-1}} + 2\sqrt{\frac{(k-1)\alpha(\beta_r+\beta_s)W\widetilde{P'_{k-1}}}{P}}$$

where $\widetilde{P'_{k-1}} = \sum_{i=1}^{k-1} p'_i$. The difference between the two is thus

$$\begin{aligned}\Delta(k) = \widetilde{T_{k-1}} - \widetilde{T_k} &= k\beta_s\left(\sum_{i=1}^{k-1}\widetilde{p'_i} - \sum_{i=1}^{k}\widetilde{p_i}\right) - \beta_s\sum_{i=1}^{k}\widetilde{p_i} \\ &\quad + 2\sqrt{\frac{2\alpha\beta_s W}{P}}\left(\sqrt{(k-1)\sum_{i=1}^{k-1}\widetilde{p'_i}} - \sqrt{k\sum_{i=1}^{k}\widetilde{p_i}}\right)\end{aligned} \quad (2.11)$$

Observe that the last term is asymptotically dominant here (since it is proportional to $\sqrt{W}$, and hence we have the following result.

Corollary 1. The optimal architecture is a balanced $n-1$ dimensional hyper-torus

Proof. For any k, we can show with a simple counting argument that the last term in the function $\Delta(k)$ is positive (for large enough P). Hence, the globally optimal running time will be obtained for $k = n-1$. Now, we also see from (2.10) that the optimal running time is an increasing function of $\widetilde{P_k}$, and this is minimized by a *balanced* hyper-torus, i.e., one in which, $p_i = \sqrt[n-1]{P}$ in each dimension. □

We leave it as an exercise to the reader to show that this result holds even for the high bandwidth BSP cost function (2.8). Indeed, the proof is simpler since the k and the $k-1$ terms drop off.

We also note that the optimization of the processor architecture yields in improvement in the last two terms of (2.10), but leave the first term (the ideal parallel running time) unchanged. Hence, the optimal architecture yields only a second order improvement. Nevertheless, a strong result that states that a certain architecture is always better, is interesting by itself.

2.1.5 Solution for the BSP cost function

We will now solve our optimization problem using the BSP cost function defined by (2.7) in the feasible space specified by (2.2). We will show that our problem can be decomposed into two special cases. The first case is very similar to the HTK model, but the one is more complicated, for which we first solve the corresponding *unconstrained optimization problem* and then determine where the constrained solution lies. We will also discuss which of the two cases is more likely in practice. Our results are based on the following key observation.

Lemma 1. The solution of **Prob. 2** satisfies

$$\begin{array}{lrcl} \text{either} & x_i & = & \frac{N_i}{p_i}, \text{ for } i = 1 \ldots k \\ \text{or} & x_i & = & 1, \text{ for } i = k+1 \ldots n \end{array}$$

Proof. Consider a feasible solution, $\vec{x}$ with volume $\prod x_i$, and suppose that we follow a "constant volume" trajectory as follows. Increase any or all of x_i for $i = 1 \ldots k$ and (correspondingly) decrease any or all of x_i for $i = k+1 \ldots n$, so as to maintain the volume constant. Observe that along this trajectory the cost function (2.7) decreases monotonically. We can continue to do so until we reach a boundary of the feasible space, R, i.e., one of the above conditions holds. □

Based on this, we have to look for the solution in the two regions of R corresponding to the above two conditions.

Case I

Let $R_1 = R \cap x_i = \frac{N_i}{p_i}$ for $i = 1 \ldots k$, and $R_2 = R \cap x_i = 1$ for $i = k+1 \ldots n$. The cost function in region R_1 can be simplified to

$$T_k(\vec{x}) = (\alpha + 2\tau_c \widetilde{N_k})\frac{W}{P} + \beta\widetilde{P_k} + \frac{\beta W}{Pv} + (\alpha + 2\tau_c\widetilde{N_k})\widetilde{P_k}v \tag{2.12}$$

where $\widetilde{N_k} = \sum_{i=1}^{k} \frac{p_i}{N_i}$. We can now use the same reasoning as in Section 2.1.4 leading to Theorem 10 and obtain the optimal volume and running time as follows:

$$\widetilde{v} = \sqrt{\frac{\beta W}{P(\alpha + 2\tau_c\widetilde{N_k})\widetilde{P_k}}} \tag{2.13}$$

$$\widetilde{T_k} = (\alpha + 2\tau_c\widetilde{N_k})\frac{W}{P} + \beta\widetilde{P_k} + 2\sqrt{\frac{\beta W\widetilde{P_k}(\alpha + 2\tau_c\widetilde{N_k})}{P}} \tag{2.14}$$

As before, appropriate values for $x_{k+1} \ldots x_n$ in R_1 that yield the optimal volume are all equivalent. Observe that in (2.14) we have a factor $2\tau_c\widetilde{N_k}$ in the dominant term. It is thus very important to minimize this term, by choosing the architecture and mapping. We will address this later.

Case II

Let us now consider that $x_i = 1$ for $i = k+1 \dots n$. Here, $v = \prod_{i=1}^{k} x_i$, but the cost function remains the same as (2.7), except that we have only k variables to solve for. We obtain the solution in two steps.

Unconstrained optimization We first solve the problem in the entire positive orthant, without any constraints. This can be formulated as follows.

Prob. 2: Minimize (2.7) in the feasible space $R_+^k = \left\{ [x_1 \dots x_k]^{\mathrm{T}} \mid x_i \geq 0 \right\}$.

Let $H_v = \{\vec{x} \in R_+^k \mid \prod_{i=1}^{k} x_i = v\}$ be the hyperboloid with volume v. Observe that the set of families H_v is a partition of R_+^k, i.e., $R_+^k = \bigcup_v H_v$, and hence $\min_{R_+^k} T_k(\vec{x}) = \min_v \min_{H_v} T_k(\vec{x})$. Thus, we first minimize (2.7) for a given tile volume (i.e., over a given hyperboloid H_v) and then choose the volume. Now, observe that for a fixed v, (2.7) is of the form $A + B \sum_i \frac{1}{x_i}$. Hence we need to minimize $\sum_i \frac{1}{x_i}$. We have the standard inequality for any set of non-negative y_i

$$\frac{1}{k} \sum_{i=1}^{k} y_i \geq \sqrt[k]{\prod_{i=1}^{k} y_i}$$

and we know that the relation is an equality when $y_1 = y_2 = \dots = y_k$. Hence, for a given tile volume, v, the optimal tile shape is (hyper) cubic with $x_i = \sqrt[k]{v}$, for each $i = 1 \dots k$. Thus, $\sum_{i=1}^{k} \frac{1}{x_i} = \frac{k}{\sqrt[k]{v}}$, and we can define,

$$f(v) = \min_{H_v} T(\vec{x}) = \frac{A}{v} + Bv + 2k\tau_c \widetilde{P_k} v^{1-\frac{1}{k}} + kDv^{-\frac{1}{k}} + \frac{\alpha W}{P} + \beta \widetilde{P_k} \tag{2.15}$$

where $A = \frac{\beta W}{P}$, $B = \alpha \widetilde{P_k}$, and $D = \frac{2\tau_c W}{P}$. Now we have to determine the optimal tile volume by minimizing $f(v)$ in the feasible space, $1 \leq v \leq v_{\max}$. The first derivative of $f(v)$ is given by

$$f'(v) = \frac{-A}{v^2} + B + 2\tau_c(k-1)\widetilde{P_k} v^{-\frac{1}{k}} - Dv^{-\frac{1}{k}-1} \tag{2.16}$$

We can solve $f'(v) = 0$ exactly by either a symbolic mathematics package, or by numerical means. However, the following approximate solution is much easier. We define a function $h(v)$ which is an approximation of $f'(v)$ from above (in the sense that $f'(v) \leq h(v)$ for $v \geq 1$) as follows:

$$h(v) \;=\; \frac{-A}{v^2} + B + 2\tau_c(k-1)\widetilde{P_k} - \frac{D}{v^2} \tag{2.17}$$

Its zero is at $\sqrt{\frac{A+D}{2\tau_c(k-1)\widetilde{P_k}+B}}$, and substituting and simplifying, we obtain the following (approximate) optimal solution of the unconstrained problem.

$$\widetilde{v} \approx \sqrt{\frac{\beta + 2\tau_c}{(2\tau_c(k-1)+\alpha)\widetilde{P_k}}\frac{W}{P}} \tag{2.18}$$

$$\begin{aligned}\widetilde{T_k} \approx\ & \frac{\alpha W}{P} + 2k\tau_c\left(\frac{(2\tau_c(k-1)+\alpha)\widetilde{P_k}}{\beta+2\tau_c}\right)^{\frac{1}{2k}}\left(\frac{W}{P}\right)^{\frac{2k-1}{2k}} \\ & + \sqrt{\widetilde{P_k}}\left(\beta\sqrt{\frac{(2\tau_c(k-1)+\alpha)}{\beta+2\tau_c}} + \alpha\sqrt{\frac{\beta+2\tau_c}{2\tau_c(k-1)+\alpha}}\right)\sqrt{\frac{W}{P}} \\ & + 2k\tau_c\left(\frac{\beta+2\tau_c}{2\tau_c(k-1)+\alpha}\widetilde{P_k}^{\frac{k+1}{k-1}}\right)^{\frac{k-1}{2k}}\left(\frac{W}{P}\right)^{\frac{k-1}{2k}} + \beta\widetilde{P_k}\end{aligned} \tag{2.19}$$

Of course we could always determine an exact solution if needed, but we have found the approximate solution to be reasonable in practice. Moreover, as we shall see later, it illustrates some interesting points. Also recall that in the problem as we have resolved so far, we assume that k and the p_i's are fixed, as also the choice of which of the N_i's to map to the processor space.

Constrained Optimization We now address the question of the restrictions on the optimal solution imposed by the feasibility constraints (2.2) namely $1 \leq x_i \leq u_i$. Note that the unconstrained (global) optimal solution is on the intersection of the line $L \equiv x_i = x_j$, for $0 < i, j \leq k$ and the hyperboloid $H_{\widetilde{v}}$, where $\widetilde{v}$ is the optimal tile volume (2.18). If this intersection is outside the feasible space (2.2) we will need to solve the constrained optimization problem. We have observed that the optimal running time is extremely sensitive to the tile volume, and much less dependent on the particular values of x_i (indeed this is predicted by the HTK model). We have the following cases

Case A: $\widetilde{v} > v_{\max}$ In this case, the optimal volume hyperboloid does not intersect the feasible space, and (given that it is more important to choose the volume optimally) we choose the tile size given by $x_i = \frac{N_i}{p_i}$, for $i = 1 \ldots k$.

Case B: $\widetilde{v} \leq v_{\max}$ Now, $H_{\widetilde{v}}$ has a non-empty intersection with (2.2) and so our heuristic is to choose a solution by moving one of the x_i's such that we move within the feasible region.

Local or Global: Let us now consider which of the two cases is more likely (of course it is preferable to be in B since this gives us a global optima). Assuming that each of the N_i grow asymptotically and that P grows much more slowly, and since $\widetilde{v}$ is proportional to $\sqrt{W/P}$, it is obvious that $w_{\max} > \widetilde{v}$ (asymptotically, for a large enough k, typically $k > n/2$). This leads to the following

Proposition 1. *Asymptotically, the globally optimal tile size is feasible.*

Proof. We have seen that we will always (asymptotically) be in *Case B*. Now consider the case when the intersection of the line L with $H_{\tilde{v}}$ is outside the feasible region (which is a rectangular (hyper) parallelepiped, with vertices $\vec{1}$ and $\left[\frac{N_1}{p_1}, \ldots, \frac{N_k}{p_k}\right]$. Now observe that the intersection of L and H_v is a feasible point for **any** feasible tile volume v, iff L is the *diagonal* of this parallelepiped. Hence, by choosing the architecture such that each p_i is proportional to N_i, we achieve our objective. □

Where is the solution?

The optimal solutions for the two cases are given, respectively by (2.13-2.14) for region R_1 and by (2.18-2.19) for region R_2. Most authors [6, 13, 12] have only considered region R_1 either implicitly by not posing the problem in full generality, or by erroneously claiming that the solution is always in R_1. In fact, not only is this incorrect, but the final solution always asymptotically be in R_2! Indeed, this is obvious when we see the additional factor, $2\tau_c\widetilde{N_i}$ in the dominant term in (2.14) as compared to (2.19).

Let us now consider the other terms in (2.19). We see that the second most dominant term is $(\frac{W}{P}^{\frac{2k-1}{2k}}$, whose degree increases with k. Hence a smaller dimensional architecture is better (in the limit, a linear array will reduce this term to just $K\sqrt{W/P}$). However, with a linear array, it may be impossible to make $\tilde{v}$ a feasible volume (see discussion above). Hence our strategy is to choose an architecture that minimizes the dimension k, but still satisfies $\tilde{v} \leq v_{\max}$.

2.1.6 Conclusions

We addressed the problem of finding the tile size that minimizes the running time of SPMD programs. We formulated a discrete non-linear optimization problem using first an abstract model and then specific machine model. The resulting cost function is general enough to subsume most of those in the literature, including the BSP model. We then analytically solved the resulting discrete nonlinear optimization problem, yielding the desired solution.

We first developed a solution for a simplified version of the cost function, and then extended it to resolve the more general BSP model, which is gaining widespread acceptance as a suitable model for architecture independent parallel programming.

There are a number of open questions. The first one is the direct extension to the non orthogonal case (when the tiles boundaries cannot be parallel to the domain boundaries). We have addressed this elsewhere (for the 2-dimensional case) and formulated a non-linear optimization problem [2], but a closed form solution is not available. Finally, experimental validation on a number of target machines is the subject of our ongoing work.

2.2 Optimal Semi-Oblique Tiling

2.2.1 Introduction

Iteration space tiling [17, 7] (also called loop blocking, partitioning, etc.) is a well-known technique used by compilers and automatic parallelizers to improve data locality and to control program granularity by increasing the computation to communication ratio (see also [10, 25, 14, 36, 38]). It is also used by programmers for manually tuning parallel and sequential codes. On distributed memory machines it may be implemented as follows. The program executes in SPMD (Single Program Multiple Data) fashion, communication being performed by send/receive calls. A *tile* in the iteration space is a (hyper parallelepiped shaped) collection of iterations to be executed as a single unit with the following protocol—all the (non-local) data required for each tile is first obtained by appropriate communication calls (eg. with libraries such as MPI, OpenMP or BSPLib). The body of the tile is executed next, and this is repeated iteratively.

The *tiling problem* can be broadly defined as that of choosing all the tile parameters (notably the tile shape and tile size) in an optimal manner. In its most general formulation, this is a hard, discrete, non-linear optimization problem, and there is currently no solution. However, optimal solutions can be found analytically under certain restrictions. The problem is usually decomposed into two subproblems: *tile shape optimization* [22], and *tile size optimization* [20, 6, 9, 13]. By its very nature, such a two step approach is not globally optimal, but often makes the problem tractable. Some authors simultaneously resolve both problems under certain simplifying assumptions [14, 36, 28].

Whether tiling is used for locality enhancement (i.e., optimizing the performance of a multiple-level memory hierarchy on a single processor) or in the context a parallel machine, the overall problem is identical, but there are important differences in the details. For the former, the models are usually statistical (due to the nature of caches), there are few exact models of program performance, and the solutions are often heuristic. This is not the case in the latter context, the case that we treat in this paper.

Most of the work on optimal tiling (we defer a detailed discussion to Section 2.2.8) considers perfect loop nests with (hyper) parallelepiped shaped iteration spaces (domains) and uniform dependences. Furthermore, with a few exceptions, all *pragmatic* work on the subject—where theoretical predictions are backed with experimental evaluations on real machines—further restricts the tiles boundaries to be parallel to the domain boundaries (Andonov and Rajopadhye call this *orthogonal* tiling [20]). When this restriction is relaxed we have *oblique* tiling.

Though oblique tiling is used by expert programmers, the common arguments *against* it are that

- automatic code generation for arbitrary slopes of the tile boundaries is difficult;
- the resulting code will need to test for many boundary cases and hence is likely to be inefficient;
- most programs can always be tiled orthogonally, albeit in a degenerate manner by not tiling certain loops (i.e, by letting the tile size be 1 in these dimensions)

It is not therefore obvious that oblique tiling will systematically yield improved performance.

Recently the gains of oblique tiling (eg. more than 20% improvement over the best orthogonal tiling) were demonstrated through analytic models and also experimentally verified by a number of authors [19, 30]. Wonnacott [39] even observed significant gains in cache performance on uniprocessors. All authors considered a restricted case where all but *one* set of tile boundaries are parallel to the domain boundaries. We call such a tiling *semi-oblique*.

Restricting ourselves to semi-oblique tiling is motivated by many factors: (i) most cases illustrating the advantages of oblique tiling actually use semi-oblique tiling; (ii) after semi-oblique tiling, the resulting tiled program can be easily parallelized with good (provably optimal) load balance through either block or block-cyclic allocation of rows (or "stacks") of tiles to processors, while this is not necessarily true for all tilings; (iii) code-generation for the tiled program is not as difficult as for the general case; and (iv), it renders our mathematical development more tractable.

In this paper, we address the problem of optimal semi-oblique tiling, but restricted to a *two dimensional* parallelogram shaped iteration domain. Such a tiling is completely specified by four parameters: the tile size (i.e., its width and height), the number of processors, and the tile shape (i.e., the slope of the oblique tile boundary). We resolve the problem of choosing *all* these parameters optimally, an improvement over previous results where one or more parameters were fixed, leading to a sub-optimal solution. Our formulation is based on the BSP model [11, 16], a widely accepted portable model for performance prediction on parallel machines. Our solution can be trivially incorporated into a compiler with almost no overhead, since it is *exact* and *analytical* (a closed form solution given as a formula). It may even be possible to envisage its use in a run-time system: if some instrumentation code is added at the beginning of the program (say for the first few iterations), then the machine and program parameters used to determine the tile size and shape can be more precisely measured, and the optimal tiling parameters can be instantiated at that time.

Our specific contributions are as follows.

- We prove, by comparing with a PRAM lower bound, that for *any* semi oblique tiling, our proposed allocation of tiles to processors is optimal.
- We formulate and analytically solve a non-linear optimization problem to determine the *tile size* that minimizes the total execution time of the tiled program on a parallel machine for a given number of *processors* and given tile *shape*. As part of the solution, we also provide a means to analytically determine the optimal number of processors for a given problem instance (i.e., the number beyond which there is no reduction in the running time). Finally, we give a gradient-like algorithm to determine the optimal tile *shape*.
- Among numerous computational experiments we validate our predictions by parallelizing Fickett's k-band algorithm for finding the best global alignment of two *similar* sequences [27, 37]. This is a common and often repeated task in computational molecular biology, and has proved difficult to parallelize, and hence our parallelization is interesting in its own right. The optimal performance was obtained simply by plugging the problem parameters into our general solution.

The remainder of this paper is organized as follows. In Section 2.2.2 we describe semi-oblique tiling and describe the tiling problem by tiling parameters. In Sections 2.2.3–2.2.5 we show how to determine in an optimal way all the tiling parameters. In Section 2.2.7 we present experimental results based on the sequence comparison application described in Section 2.2.6. In Section 2.2.8 we give a detailed discussion of the related work. In section 2.2.9 we conclude.

2.2.2 Problem Formulation

We first describe and formalize two-dimensional semi-oblique tiling. Next, we parallelize the resulting *tile graph* on an ideal machine (a p-processor PRAM), determine a lower bound for its running time, and then describe a parallelization on a pragmatic machine using a simple block-cyclic distribution of tiles to processors. We show that if we model the communication costs of the machine using the BSP model, our block-cyclic strategy remains optimal. Finally, using the running time predicted by the BSP model, we formulate the discrete non-linear optimization problem to determine the tiling parameters so as to minimize the running time.

Semi-oblique tiling

We consider the problem of parallelizing a uniform recurrence [33] of the form

$$a[i,j] = f(a[i-d_1, j-d'_1], \ldots, a[i-d_w, j-d'_w]) \tag{2.20}$$

over the $N \times M$ rectangular[3] domain $D = \{(i,j) \mid 0 \le i < N,\ 0 \le j < M\}$, with a set of dependence vectors,

$$\delta_1 = \begin{pmatrix} d_1 \\ d'_1 \end{pmatrix}, \ldots, \delta_w = \begin{pmatrix} d_w \\ d'_w \end{pmatrix}$$

We denote the largest i-component of any dependence vector by $\delta = \max_x(d_x)$. We assume that all the dependence vectors belong to the cone generated by the extreme vectors $\binom{1}{-\gamma}$ and $\binom{0}{1}$ for some *rational* γ (which could have *any* sign). It is well known [40] that the following tiling is legal: tiles are parallelograms with one pair of sides parallel to j axis, and the other pair with a slope $-\gamma$. Each tile contains s rows of iteration domain points and there are r points in each row. We call the integers s and r as *tile height* and *width* respectively.

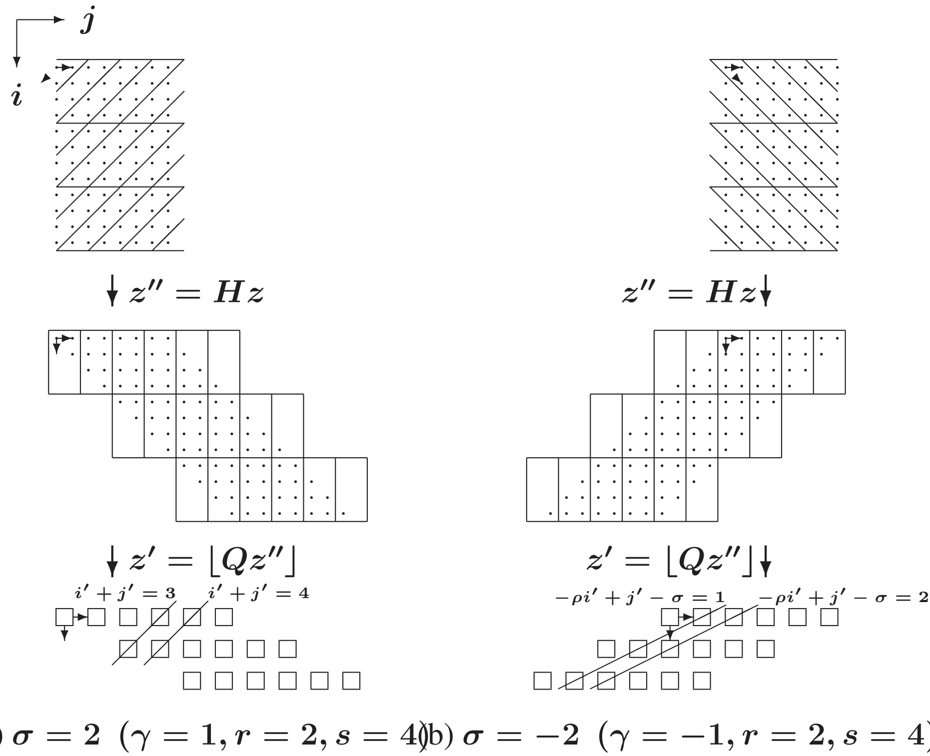
Such a tiling is formally described as follows. Let

$$H = \begin{pmatrix} 1 & 0 \\ \gamma & 1 \end{pmatrix}, \quad Q = \mathrm{diag}\left(\frac{1}{s}, \frac{1}{r}\right), \quad \text{and } T = QH$$

where the rows of H are given by the normals of the parallel lines specifying the tiles. For any two integers, i' and j' the following set is called the (i', j')-th tile.

$$\{(i,j) \mid i's \le i < (i'+1)s,\ j'r \le \gamma i + j < (j'+1)r\}$$

[3] A parallelogram shaped domain can be easily transformed to a rectangular one, without loss of generality (indeed, the program for Fickett's algorithm has such a domain).



(a) $\sigma = 2$ $(\gamma = 1, r = 2, s = 4)$ (b) $\sigma = -2$ $(\gamma = -1, r = 2, s = 4)$

Figure 2.1: Transformation of the domain D and optimal schedules in the transformed domain

It is easy to verify that the transformation $z' = \lfloor Tz \rfloor$ maps this set to the point (i', j'). The integer points of the transformed domain connected by the transformed dependence vectors represent the so called *tile graph*. Like the tiles, the transformed domain is also a parallelogram with one side parallel to the j' axis. The other side has a slope $\sigma = \frac{\gamma s}{r}$. This slope, which is identical to the *rise* as defined by Högstedt et al. [30], turns out to be a critical parameter. It is well known [7] that under standard assumptions, the extreme dependences in the tile graph are the canonical ones (provided the tiles are large enough compared to the original dependences). The tile graph contains $n = \frac{N}{s}$ rows of tiles and there are $m = \frac{M}{r} + |\sigma|$ tiles per row.

Fig. 2.1 illustrates two typical cases (for $M = 8$, $N = 12$, $r = 2$ and $s = 4$). In Fig. 2.1a, $\gamma = 1$, which gives $\sigma = 2$; in Fig. 2.1b $\gamma = -1$, resp. $\sigma = -2$. We use the convention that the origin is top-left, i (resp. i') is vertical downwards, j (resp. j') is horizontal to the right. Diagonals in the bottom diagrams are optimal schedule lines (as explained later). For simplicity, we do not show all tile graph dependencies and schedule lines. The rise, σ is explained intuitively by the fact that successive rows of the tile graph are shifted to the right (left, if σ is negative) by precisely σ.

So far, we implicitly assumed that σ is integer. If this is not so, the picture changes and a number of complications arise—the rows of the tile graph do not remain identical, the shift between successive rows is not constant, etc. Even the number of tiles in different rows may be different. If we want to give exact formulæ for this case, we have to use floor and ceiling func-

tions, greatest common divisors and other integer arithmetic tricks. In the interest of simplicity, we prefer to use a *rational approximation* and proceed as outlined above, since it describes the average case. For example, though the number of tiles in each row may be different, it repeats with some periodicity and the *average* number of tiles in each row is m. Similarly, the *average* shift between successive rows (as we shall shortly see) is $\sigma + 1$ (which we shall henceforth denote by $\bar{\sigma}$). This approximation simplifies our mathematical model, without straying too far from reality. The same remark is true and for the case when m or n is not integer. The effect of these approximations will be discussed again in Section 2.2.7 when we present our experimental results.

Our notational conventions are as follows: we use upper case letters, N and M to denote, respectively, the height and width of the original domain, and lower case letters (n and m) for the corresponding parameters of the tile graph. The height and width of the tile itself are s and r, respectively. The slopes of the non-horizontal boundaries of the original domain, the tile graph, and the tile itself, are respectively, 0, σ and γ.

Parallelizing the tile graph on a PRAM

We now investigate the parallelization of the tile graph, first on an unbounded number of processors, and on a fixed number of processors. We also give a mapping of tiles to processors that achieves optimal performance.

In our analysis we will assume that all tiles, including boundary (i.e., "partial") tiles, take a single "time step" to execute. This is justified by the fact that although boundary tiles have fewer computations to perform, other processors are simultaneously computing "full" tiles, and the data dependences between the tiles ensure the necessary synchronization. We will see later that it is also consistent with the BSP model that we use, since the model assumes a global synchronization at each step.

We will consider the following two cases (see Fig. 2.1).

Case a: $\bar{\sigma} > 0$. In this case, because of the dependencies between the tiles, the optimal schedule on a unbounded number of processors is $(1, 1)$, i.e., all tiles on the line $i' + j' = t$ start at instant t. According to this schedule, the first tile of the i'th row, namely tile $(i', \sigma i')$, can start at the moment $\bar{\sigma} i'$. Thus the delay of each row relative to the previous one is $\bar{\sigma}$.

We may visualize the schedule by *skewing* the tile graph, i.e., shifting each row to the right relative to the previous one (by 1), yielding another parallelogram (called the *scheduled* tile graph) whose oblique boundary has slope $\bar{\sigma}$. We interpret the horizontal axis of this parallelogram as the time.

Taking into account that the number of rows is n and that number of tiles in a row is m, we can easily verify that the make-span of this schedule is $\Sigma = (n - 1)\bar{\sigma} + m$. It is also easy to see that if $m \geq n\bar{\sigma}$, the maximum degree of parallelism is n, otherwise it is $\frac{m}{\bar{\sigma}}$, and hence the above make-span can be achieved on $\min(n, \frac{m}{\bar{\sigma}})$ or more processors.

Now assume that we have $p < \min(n, \frac{m}{\bar{\sigma}})$ processors (we let $\bar{p}$ denote $p - 1$, henceforth; we also assume that $p|n$ whenever $p \leq n$). Observe that for any of the first (c.f. last) $\bar{p}\bar{\sigma}$

steps, the number of tiles that can be executed at any instant is less than p. The number of tiles during this "startup" (c.f. "wrap-up") phase is $\frac{1}{2}\bar{p}\bar{\sigma}p$. Thus, a total of $\bar{p}\bar{\sigma}p$ tiles take $2\bar{p}\bar{\sigma}$ time steps. The *remainder* of the tile graph contains $nm - \bar{p}\bar{\sigma}p$ tiles. Obviously, any parallel implementation of these tiles on p processors will take at least $\frac{nm}{p} - \bar{p}\bar{\sigma}$ time. Hence, a lower bound on the make-span of our tile graph on p processors is $\frac{nm}{p} + \bar{p}\bar{\sigma}$ (see Fig. 2.2(a)).

Case b: $\bar{\sigma} \leq 0$. In this case, tiles on the line $t = j' - \sigma i' - \sigma$ can start at the instant t, and the optimal schedule is $(-\sigma, 1)$. There is no delay between successive rows (all rows can start simultaneously). The make-span is $\Sigma = m$, the width of the tile graph, and it can be achieved on n or more processors. Furthermore, if we have only $p < n$ processors, the make-span is bounded from below by $\frac{nm}{p}$. Thus, in general, the make-span is bounded from below by $\max(\frac{n}{p}, 1)m$. Also observe that for $\bar{\sigma} = 0$, the two formulæ are identical.

We now give a specific mapping of our tile graph, and show that it achieves the above lower bound. We allocate tiles to processors by rows in the so-called *block-cyclic* manner and adopt a *multiple-pass* execution. In other words, the i'th row of the tile graph is allocated to the i'th (virtual) processor. The n virtual processors are then mapped cyclically to the p physical processors. The mapping may be visualized as horizontally splitting the $n \times m$ tile graph into $\frac{n}{p}$ slices, each of size $p \times m$ (Fig. 2.2(b) and (c) illustrate two possibles cases between two successive slices).

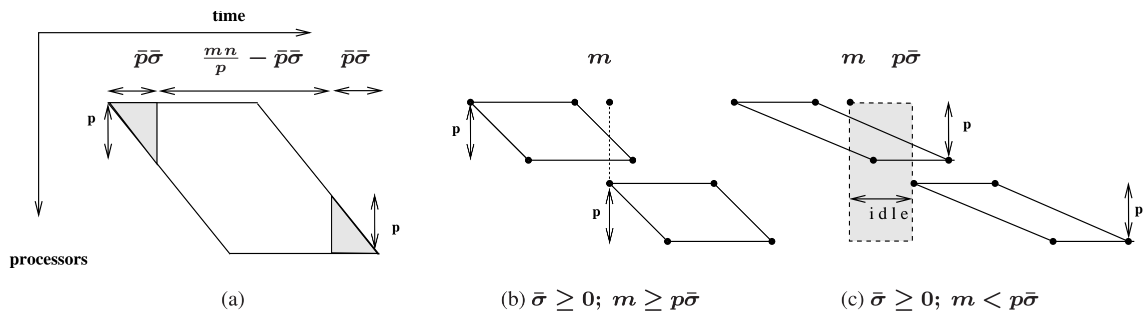


Figure 2.2: Scheduled tile graph: (a) Illustration for the PRAM make-span lower bound. (b) and (c) Scheduling on limited number of processors - two possibles cases for successive passes.

Theorem 11. *The above mapping is optimal.*

Proof. Each slice is executed as a *pass* on the p processors. It is clear that for case (b) above, this strategy achieves the optimal make-span, $\max(\frac{n}{p}, 1)m$.

For case (a) i.e., when $\bar{\sigma}$ is positive, the make-span of a single pass is clearly $\bar{p}\bar{\sigma} + m$. Note however, that successive passes can be pipelined—a pass may start (i.e., its *first* tile is executed by the *first* processor) even before the previous pass is completely finished (i.e., its *last* tile is executed by the *last* processor). In fact, there are two constraints that determine when a pass can

start: the first processor must have finished executing all the m tiles of the previous pass, and the last processor must have finished executing its σ-th tile from the previous pass (since the first tile in the *next* pass depends *directly* on it). Hence the interval between the start of two successive passes is $\max(m, p\bar{\sigma})$. Both possible cases are illustrated on Fig. 2.2(b) and (c). Note that when $m < p\bar{\sigma}$ there is an idle time between two successive passes. Since there are $\frac{n}{p}$ passes, the last pass can only start at time $(\frac{n}{p} - 1)\max(m, p\bar{\sigma})$, and this last pass itself takes $\bar{p}\bar{\sigma} + m$ additional time steps. Putting this together and simplifying, we get the following expression for the make-span of the tile graph on p processors.

$$\Sigma = \begin{cases} \Sigma_1 = \frac{mn}{p} + \bar{p}\bar{\sigma} & \text{if } 0 \leq \bar{\sigma} \leq \frac{m}{p} \\ \Sigma_2 = m + (n-1)\bar{\sigma} & \text{if } \frac{m}{p} \leq \bar{\sigma} \\ \Sigma_3 = \max\left(1, \frac{n}{p}\right) m & \text{if } \bar{\sigma} \leq 0 \end{cases} \tag{2.21}$$

Comparing this with the PRAM make-span, we see that block-cyclic allocation is optimal. □

Performance modeling on a BSP machine

The above make-span Σ reflects the time in "number of parallel steps" where each step corresponds to the execution of a single tile (and communication of its results). To convert it into running time on a realistic machine, we need to model communication. A detailed machine model allows the possibility of overlapping communication and computation. However, such approach complicates matters and renders the resulting discrete non-linear optimization problem extremely difficult to resolve analytically. More on this subject in the experimental section 2.2.8

To simplify things we use the BSP model [16], where computation proceeds in a sequence of *macrosteps*. Each macrostep consists of independent local computation by the individual processors interspersed with the *initiation* of communication activity, through so-called *one-sided* communication primitives such as `get` and `put`. A macrostep terminates with a global barrier synchronization, and it is during the synchronization that the communication is actually effected, the results being available for the next macrostep. In addition to the number of processors, p, a BSP machine is uniquely characterized by two parameters, the synchronization cost of the machine, β' and the *network permeability*, τ' (both parameters defined relative to the processor speed). The permeability is defined in terms of the time for the interconnection network to realize what is called an h-relation—an arbitrary (all-to-all) communication between processors where each processor sends/receives no more than h bytes of data. The permeability is τ' if an h-relation takes $h\tau'$ processor cycles.

In our parallelization of the tile graph, each macrostep is identical, and corresponds to what we have called a step in determining our PRAM make-span, Σ. Hence the total execution time is simply ΣP, where P is the time for performing a macrostep. As a direct consequence, the above result of the optimality of tile allocation remains valid (other models which allow communication-computation overlap do not necessarily retain this optimality).

We now determine a precise analytic formulation of the execution time of our program, and formulate our discrete non-linear optimization problem.

Two tiles are said to be *successive* if one of them depends directly on the other. Our mapping of tiles to processors ensures that successive tiles are mapped either to the same processor (in which case there is no communication involved) or to processors that are adjacent (modulo p). As a result, it is easy to see that during a macrostep, each processor executes rs instances of the original loop body. Moreover, the communication during a macrostep consists of each processor, i' sending a number of data elements to $i' + 1$ (modulo p). The number of data values sent is proportional to r. Indeed, if we return to the description of the semi-oblique tiling of Eqn (2.20) as given in Section 2.2.2, we can easily see that the exact number of transmitted elements is δr (recall that $\delta = \max_x(d_x)$ is the largest i-component of the dependence vectors in the original program). Note that even even though there may be oblique dependences in the tile graph, the corresponding data values come from only a "corner" of the tile, and moreover, they are subsumed by the previous tile's communication if each tile sends precisely δ rows of values. As a result, we have

$$P = \beta + \tau r + \alpha r s \tag{2.22}$$

for appropriate constants α, β and τ. Therefore, in order to find the optimal tile size, we need to minimize $T(r, s) = \Sigma P$, where

$$T(r,s) = \begin{cases} T_1(r,s) = \left(\frac{mn}{p} + \bar{p}\bar{\sigma}\right) P & \text{if } 0 \le \bar{\sigma} \le \frac{m}{p} \\ T_2(r,s) = (m + (n-1)\bar{\sigma})\, P & \text{if } \frac{m}{p} \le \bar{\sigma} \\ T_3(r,s) = \max\left(1, \frac{n}{p}\right) mP & \text{if } \bar{\sigma} \le 0 \end{cases} \tag{2.23}$$

Although we leave only r and s as explicit variables, p and γ are also unspecified parameters of the tiling and must be determined. The continuity of this function follows trivially from the facts that $T_1 = T_2$ at the boundary $m = p\bar{\sigma}$, and $T_1 = T_3$ at the $\bar{\sigma}$ boundary.

Each of the cases of Eqn. 2.23 implicitly defines a sub-domain of the feasible space of our optimization problem. Before resolving the problem, we first show that the third case of Eqn. 2.23, can be eliminated. Observe that this case is only possible for $\gamma < 0$. Note that for any fixed s the associated timing function, can be simplified to

$$T_3(r,s) = \max\left(1, \frac{n}{p}\right)(M - \gamma s)\left(\frac{\beta}{r} + \tau + \alpha s\right)$$

This function monotonically decreases with r, and hence reaches its minimum for the largest possible value of r, i.e., the one that satisfies $\bar{\sigma} = 0$. This straightforward observation shows that we are not interested in any r and s such that $\bar{\sigma} < 0$. In other words, we can eliminate the third case of Eqn. (2.23) and solve our optimization problem under the constraint $\bar{\sigma} \ge 0$.

2.2.3 Optimizing the tile size

In this section we suppose that the tile shape γ and the number of processors p are fixed, and consider the running time as a function of only r and s. We will show how to minimize this function over the space of feasible tile sizes.

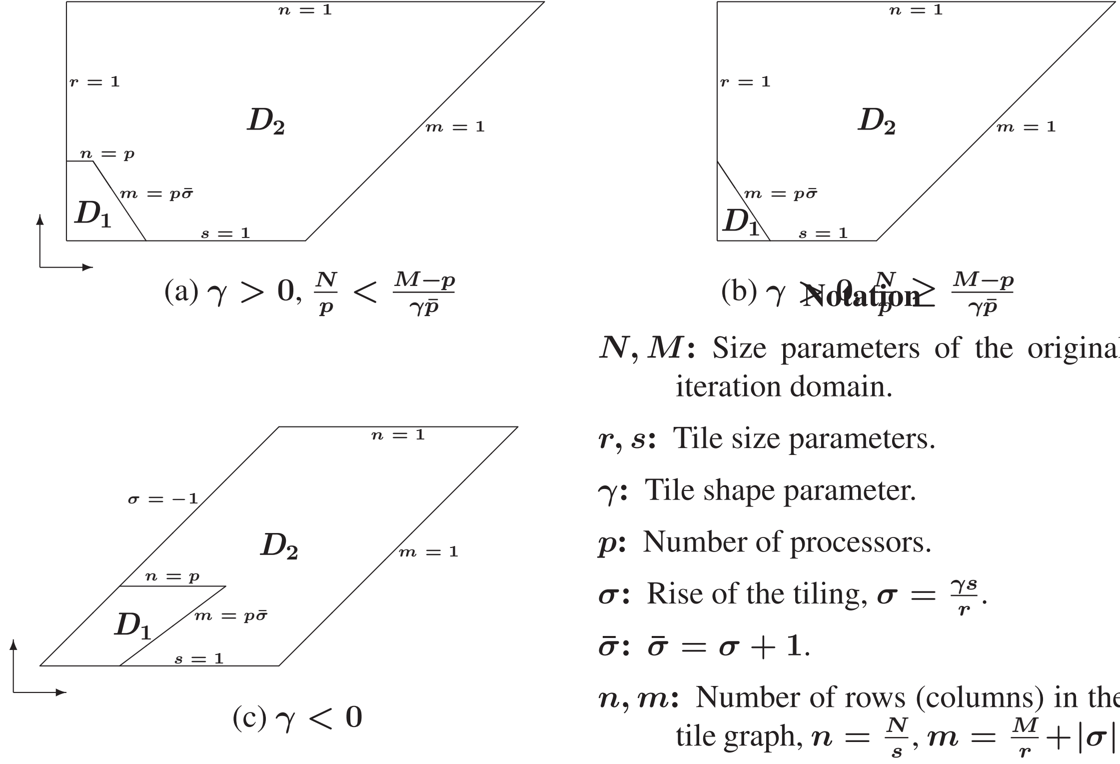


Figure 2.3: Subdivision of the feasible domain R

The restrictions $n \geq 1$ and $m \geq 1$ (at least one row in the tile graph and at least one tile per row) give upper bounds on r and s. Let r_- and s_-, be the respective lower bounds, i.e., $r \geq r_-$ and $s \geq s_-$. These bounds are imposed by the validity conditions for a tiling, namely that tiles must be large enough to span the original dependences. We suppose henceforth that these bounds are 1; however all our results can be directly generalized. So the feasible domain of our optimization problem is

$$R = \{(r, s) \mid r \geq 1,\ s \geq 1,\ m \geq 1,\ n \geq 1,\ \bar{\sigma} \geq 0\}$$

This domain is subdivided by the straight line $m = p\bar{\sigma}$. Note next that the first case of Eqn. (2.23) is meaningful only for $n \geq p$ since the factor $\frac{n}{p}$ in it corresponds to the number of passes. If $n < p$ then the number of passes is 1 and the running time is given by the second case of Eqn. (2.23). If we define the two sub-domains by $D_1 = R \cap \{(r, s) \mid m \geq p\bar{\sigma},\ n \geq p\}$ and $D_2 = R \setminus D_1$ (see Fig. 2.3), the optimization problem is naturally decomposed into the following two subproblems.

Problem P_1 : Minimize $T_1(r, s) = \Sigma_1 P$	over D_1
Problem P_2 : Minimize $T_2(r, s) = \Sigma_2 P$	over D_2

The objective functions of the two problems are equal on the boundary between their feasible domains.

The global solution is the smaller of the solutions of P_1 and P_2, both non-linear discrete optimization problems. We consider them as continuous problems and take the closest integer point to the solution. This turns out to be reasonable in practice, since our function is fairly flat near the optimum.

To solve the optimization problems P_i, $i = 1, 2$ we proceed as follows. First we find the unconstrained (not necessarily global) optima of the function T_i. Those of them which are in D_i are candidate solutions of the problem. Then we minimize T_i on each of the boundaries of D_i, from where we get more candidate solutions. Finally we take the best of these, i.e., the candidate for which the value of T_i is minimal. The following remainder of this section is devoted to these subproblems. Readers not interested in the mathematical details may skip directly to Section 2.2.3 where we summarize the results and explain how to use them.

Before we enter into the details of our solution, let us make several remarks.

Remark 7. All the domain boundaries $r = 1$, $s = 1$, $m = 1$, $n = 1$, $\sigma = -1$, $n = p$, and $m = p\bar{\sigma}$ are linear functions of r and s (straight lines in Fig 2.3).

Remark 8.

$$T_1(r, s) \geq T_2(r, s) \text{ for } (r, s) \in \mathcal{D}_1$$
$$T_2(r, s) \geq T_1(r, s) \text{ for } (r, s) \in \mathcal{D}_2$$

Our problem then consists in minimizing $T(r, s) = \max(T_1(r, s), T_2(r, s)) \in \mathcal{D}_1 \bigcup \mathcal{D}_2$ where T is continuous function.

Remark 9. Since P does not change with p, the function T_2 is *independent* of p, the number of processors.

As a direct consequence of this, if ever the solution to our optimization problem is in the subdomain corresponding to the second case of Eqn. 2.23, it should be possible to do no worse with *fewer* processors. The mathematical details how to detect when this occurs, and hence determine the optimal number of processors are given in Section 2.2.4.

Finding the unconstrained optima of T_1

From (2.22) we have $r = \frac{P-\beta}{\tau+\alpha s}$. Using this and (2.23) we can express T_1 as a function of P and s:

$$T_1(r, s) = T(P, s) = \frac{P}{P - \beta} f(s) + \bar{p}P$$

where

$$f(s) = (\tau + \alpha s)\left(\frac{MN}{sp} + \frac{|\gamma|N}{p} + \bar{p}\gamma s\right).$$

Now, from the necessary optimality condition $\nabla T(P, s) = 0$ we obtain

$$2\alpha p\bar{p}\gamma s^3 + (\tau p\bar{p}\gamma + \alpha|\gamma|N)s^2 - \tau MN = 0 \tag{2.24}$$

$$P = \sqrt{\frac{\beta f(s)}{\bar{p}}} + \beta \quad \text{or} \quad r = \frac{1}{\tau + \alpha s}\sqrt{\frac{\beta f(s)}{\bar{p}}} \tag{2.25}$$

Eqn. (2.24) is a cubic equation of s and can be solved analytically using the Cardano formulæ. From (2.24) and (2.25) we obtain (1 or 3) candidate points (r, s) for unconstrained optimum. We consider only those which are in D_1.

Optimizing T_1 on the boundaries of D_1

All the possible boundaries of D_1 are segments of the lines $s = 1$, $n = p$, $r = 1$, $\sigma = -1$ and $m = p\bar{\sigma}$. We have to consider only some of them depending on the values of the problem parameters M, N, γ and p (see Fig. 2.3). On each of these boundaries we can express T_1 as a one variable function. To optimize such a function we take into account the zeros of its derivative and the end points of the corresponding segment.

On the boundaries $s = 1$ and $n = p$ (or more generally $s = c$, where c is some constant) the function has the form

$$T_1(r, c) = T(r) = Ar + \frac{B}{r} + \text{const},$$

where $A = \beta\left(\frac{MN}{pc} + \frac{|\gamma|N}{p} + \bar{p}\gamma c\right)$ and $B = \bar{p}(\tau + \alpha c)$. The condition $T'(r) = 0$ gives $Br^2 - A = 0$. Another possible boundary is $r = 1$, for which

$$T_1(1, s) = T(s) = \frac{A}{s} + Bs + Cs^2 + \text{const},$$

where $A = \frac{MN(\beta+\tau)}{p}$, $B = \alpha\left(\frac{\gamma N}{p} + \bar{p}\right) + \bar{p}\gamma(\beta+\tau)$, $C = \alpha\gamma\bar{p}$. In this case the condition $T'(s) = 0$ gives $2Cs^3 + Bs^2 - A = 0$, which is a cubic equation of s. At the boundary $\sigma = -1$ (or $r = -\gamma s$) we have

$$T_1(-\gamma s, s) = T(s) = \frac{N}{p}\left(\frac{A}{s^2} + \frac{B}{s} + Cs\right) + \text{const},$$

where $A = -\frac{M\beta}{\gamma}$, $B = (M\tau + \beta)$, $C = -\gamma\alpha$. The condition $T'(s) = 0$ gives $Cs^3 - Bs - 2A = 0$ which is again a cubic equation of s. The last boundary $m = p\bar{\sigma}$ between D_1 and D_2 will be analyzed in Section 2.2.3 when we solve the problem P_2 since $T_1 = T_2$ on this boundary.

Finding the unconstrained optima of T_2

The function T_2 can be represented in the form

$$T_2(r, s) = \left(\tfrac{M}{r} + n\bar{\sigma}\right)P + (|\sigma| - \sigma - 1)P$$

If $\gamma \geq 0$ the last term in this representation is $-P$. If $\gamma < 0$ we have $-1 \leq \sigma \leq 0$ and from here we obtain that it is between $-P$ an P. In both cases the last term contributes at most one period. Therefore we can drop it and instead of T_2 we can consider the function

$$\hat{T}_2(r, s) = \left(\tfrac{M}{r} + n\bar{\sigma}\right)P$$

Now using $r = \frac{\gamma s}{\sigma}$ we can express $\hat{T}_2$ as a function of σ and s

$$\hat{T}_2(r,s) = T(\sigma,s) = ((M+\gamma N)\sigma + \gamma N)\left(\frac{\beta}{\gamma s} + \frac{\tau}{\sigma} + \alpha\frac{s}{\sigma}\right)$$

The necessary condition for optimum $\nabla T(\sigma, s)$ gives

$$\frac{\partial T}{\partial s} = ((M+\gamma N)\sigma + \gamma N)\left(\frac{\alpha}{\sigma} - \frac{\beta}{\gamma s^2}\right) = 0 \quad (2.26)$$

$$\frac{\partial T}{\partial \sigma} = \frac{\beta(M+\gamma N)}{\gamma s} - \frac{\gamma N(\tau + \alpha s)}{\sigma^2} = 0 \quad (2.27)$$

If the first multiple of (2.26) is zero, we substitute $\sigma = -\frac{\gamma N}{M+\gamma N}$ in (2.27) and obtain a quadratic equation of s. If the second multiple is zero, substituting $\sigma = \frac{\alpha\gamma s^2}{\beta}$ in (2.27) leads to a cubic equation of s. In this way we obtain several candidate solutions. We consider only those which are in D_2.

Optimizing T_2 on the boundaries of D_2

The possible boundaries of D_2 are segments of the lines $r = 1$, $s = 1$, $\sigma = -1$, $m = 1$, $n = 1$, $n = p$ and $m = p\bar{\sigma}$. On the boundaries $r = 1$, $s = 1$ and $\sigma = -1$ it is easy to express T_2 as a one variable function and to optimize it as we did in for T_1 in Section 2.2.3. When there is one row of tiles or one tile per row ($n = 1$ or $m = 1$) there are no tiles which can be executed in parallel. That is why the best thing to do is to execute a sequential code on a single processor and to obtain a running time αMN. The boundary $n = p$ was already investigated in Section 2.2.3. The only boundary left is $m = p\bar{\sigma}$. Although it is possible to use more sophisticated techniques and to optimize on this boundary by solving a quartic equation (which is still analytically resolvable), here we prefer to give an approximate method which is simpler, but precise enough. Once again we optimize the function $\hat{T}_2$ instead of the original function T_2. From $m = p\bar{\sigma}$, we obtain $r = \frac{M-\hat{p}\gamma s}{p}$, where $\hat{p} = p - \operatorname{sgn}\gamma$. Now we can express $\hat{T}_2$ as a function of s:

$$\hat{T}_2\left(\tfrac{M-\hat{p}\gamma s}{p}, s\right) = T(s) = \left(M + \frac{|\gamma|N}{p} + \frac{MN}{ps}\right)\left(\frac{\beta p}{M-\gamma\hat{p}s} + \tau + \alpha s\right)$$

Denote the first multiple in the last equation by $f(s)$ and the second one by $g(s)$. We have to minimize the function $T(s)$ for $s \in [1, s_{\max}]$, where $s_{\max} = \frac{N}{p}$ if $\gamma \leq 0$ and $s_{\max} = \min(\frac{N}{p}, \frac{M-p}{\gamma\bar{p}})$ otherwise. The function f is decreasing in this segment. Consider the function g. Its derivative is

$$g'(s) = \alpha + \frac{\beta p\hat{p}\gamma}{(M-\gamma\hat{p}s)^2}$$

If $\gamma \geq 0$ then $g'(s)$ is positive else $g'(s)$ has two zeros

$$s_{1,2} = \frac{M}{\gamma\hat{p}} \pm \sqrt{-\frac{\beta}{\alpha}\frac{p}{\gamma\hat{p}}}$$

The root s_2 is negative and it is out of the considered segment $[1, s_{\max}]$. There are three possibilities:

1. If $\gamma \geq 0$ or $\gamma < 0$ and $s_1 \leq 1$ then g is increasing in $[1, s_{\max}]$.
2. If $\gamma < 0$ and $s_1 \in (1, \frac{N}{p})$ then g is decreasing in $[1, s_1]$ and it is increasing in $[s_1, s_{\max}]$.
3. If $\gamma < 0$ and $s_1 \geq s_{\max}$ then g is decreasing in $[1, s_{\max}]$.

When g is decreasing in some segment, T reaches its minimum at the right end point of the segment, but when g is increasing the minimum of T may be anywhere in the segment. So we have to minimize $T(s) = f(s)g(s)$ in the segment $[s_{\min}, s_{\max}]$ where $s_{\min} = 1$ (case 1) or $s_{\min} = s_1$ (case 2), f is decreasing and g is increasing in $[s_{\min}, s_{\max}]$.

We linearize g taking instead of it the straight line $\hat{g}$ passing through the points $(s_{\min}, g(s_{\min}))$ and $(s_{\max}, g(s_{\max}))$, $\hat{g}(s) = Ks + L$, where

$$K = \frac{g(s_{\max}) - g(s_{\min})}{s_{\max} - s_{\min}}, \ L = g(s_{\max}) - Ks_{\max}.$$

Now we consider the function

$$f(s)\hat{g}(s) = As + \frac{B}{s} + \mathrm{const},$$

where $A = \left(M + \frac{|\gamma|N}{p}\right)K$, $B = \frac{MNL}{p}$, which we can easily minimize on the segment $[s_{\min}, s_{\max}]$. To find a more precise solution, we can subdivide the considered segment $[s_{\min}, s_{\max}]$ into several segments and to apply a linearization of g in each of them.

Summary

The results results above are summarized in Table 2.1. Each row of this table explains how to minimize the function named in Column 2 subject to the constraints mentioned in Column 1. We now explain how to interpret this table and how to find the optimal tile size for each problem instance (determined by M, N, γ) and for each architecture parameter setting (p, β, τ, α). Each row (we only consider those rows which are relevant to the parameters M, N, γ, and p, see Fig. 2.3) is to be read as follows: to minimize, under the constrains mentioned in Column 1, the function named in Column 2, we find the solutions of the cubic equation $Ax^3 + Bx^2 + Cx + D = 0$, with the coefficients given in Column 5 (missing coefficients are taken as 0). For each solution, x (and also for the endpoints of the segment if the feasibility condition is given as a segment), we use the formulæ in Columns 3 and 4 to determine the candidate r and s. Then we retain only those that satisfy the feasibility test in Columns 6 (the feasibility test is given either in terms of r and s or in terms of x). To find the optimal solution we take the best solution over all the relevant rows.

In row 2, the function $h(x) = \beta(\frac{MN}{px} + \frac{|\gamma|N}{p} + \gamma\bar{p}x)$. In row 6 onwards, $\hat{p} = p - \operatorname{sgn}\gamma$. The parameters L, K, $s_{\min}$ and $s_{\max}$ are defined in Section 2.2.3.

Constraints	T	r	s	coefficients	feas. set	#
none	1	$\sqrt{\frac{h(x)}{\bar{p}(\tau+\alpha x)}}$	x	$A = 2\alpha p\gamma\bar{p}, B = \tau p\bar{p}\gamma + \alpha\vert\gamma\vert N, D = -\tau MN$	$(r, s) \in D_1$	3
	2	$-\frac{x(M+\gamma N)}{N}$	x	$B = \alpha(M + \gamma N), C = \tau(M + \gamma N), D = -\beta N$	$(r, s) \in D_2$	1
		$\frac{\beta}{\alpha x}$	x	$A = \alpha^2(M + \gamma N), C = -\alpha\beta N, D = \tau\beta N$	$(r, s) \in D_2$	2
$n = p$	1,2	x	$\frac{N}{p}$	$B = \bar{p}(\tau + \alpha\frac{N}{p}), D = -h(\frac{N}{p})$	$[1, \frac{M}{p} - \frac{\gamma\hat{p}N}{p^2}]$	4
$m = p\bar{\sigma}$	1,2	$\frac{M-\hat{p}\gamma x}{p}$	x	$B = (M + \frac{\vert\gamma\vert N}{p})K, D = \frac{MNL}{p}$	$[s_{\min}, s_{\max}]$	5
$\sigma = -1$	1	$-\gamma x$	x	$A = \alpha\gamma, B = M\tau + \beta, D = -\frac{2M\beta}{\gamma}$	$[1, \frac{N}{p}]$	6
	2	$-\gamma x$	x	$A = 2\alpha\gamma, B = M\alpha - \tau\gamma, D = \frac{M\beta}{\gamma}$	$[\frac{N}{p}, N]$	7
$s = 1$	1	x	1	$B = \bar{p}(\tau + \alpha), D = -h(1)$	$[1, \frac{M-\gamma\hat{p}}{p}]$	8
	2	x	1	$B = (N - 1)(\tau + \alpha), D = \beta(M + \gamma N + \vert\gamma\vert - \gamma)$	$[\frac{M-\gamma\hat{p}}{p}, M + \vert\gamma\vert]$	9
$r = 1$	1	1	x	$A = 2\alpha\gamma p\bar{p}, B = p\bar{p}(\alpha + \gamma\beta + \gamma\tau) + \alpha\gamma N,$ $D = -MN(\beta + \tau)$	$[1, \min(\frac{N}{p}, \frac{M-p}{\gamma\bar{p}})]$	10
	2	1	x	$B = (\tau + \alpha)(M + \gamma N - 1), D = -\beta N$	$[\min(\frac{N}{p}, \frac{M-p}{\gamma\bar{p}}), N]$	11
$m = 1, n = 1$	2	M	N	$T = \alpha MN$		12

Table 2.1: Solution of the optimization problem.

Table 2.1 lists 12 separate cases, and in general, we have to consider all of them. Indeed, Table 2.2 gives a set of problem instances which cover all rows of Table 2.1. However, some of these instances are extremely unrealistic. For "normal" problems only the first six rows are relevant.

2.2.4 Optimizing the number of processors

Our model can be used not only to determine the optimal tile size and to predict the performance of the parallel code, but also to analyze the sensitivity and the influence of the problem and architecture parameters on the performance. In this section we will analyze the parameter p, the

M	N	p	γ	α	β	τ	r^*	s^*	row
20000	50000	8	1	2.6×10^{-7}	4.03×10^{-4}	2.02×10^{-7}	1498	111	3
20000	50000	30	1	2.6×10^{-7}	4.03×10^{-4}	2.02×10^{-7}	600	69	5
20000	50000	300	1	2.6×10^{-7}	4.03×10^{-4}	2.02×10^{-7}	46	34	2
20000	50000	10	-1	2.6×10^{-7}	4.03×10^{-4}	2.02×10^{-7}	410	410	6
20000	50000	1500	-1	2.6×10^{-7}	4.03×10^{-4}	2.02×10^{-7}	39	39	7
20000	120	8	1	2.6×10^{-7}	4.03×10^{-4}	2.02×10^{-7}	531	15	4
20000	50000	8	1	2.6×10^{-3}	4.03×10^{-4}	2.02×10^{-7}	1663	1	8
20000	50000	8	1	2.6×10^{-11}	4.03×10^{-4}	2.02×10^{-7}	20000	50000	12
40000	50000	8	1	2.6×10^{-7}	4.03×10^{-7}	2.02×10^{-3}	1	4065	10
40000	50000	600	1	2.6×10^{-7}	4.03×10^{-7}	2.02×10^{-3}	1	66	11

Table 2.2: Problem instances covering the rows of Table 2.1

number of processors.

Recall that $T_2(r, s)$ is *independent* of p, the number of processors (see Remark 9)—we can obtain the same running time for our program using fewer processors! This can be easily explained by looking once again at the tile graph at Fig. 2.1. The maximal number of tiles that can be executed in parallel is equal to the maximal number of points on the schedule lines. For any fixed r and s the optimal number of processors is equal to this "thickness" of the tile graph and one cannot expect better performance with more processors. For all points in the interior of D_2 the thickness of the tile graph is less than the declared number of processors. In other words, the points in D_1 correspond to tile sizes for which the available resources (processors) are insufficient, while the points in D_2 give tile sizes for which there is redundancy of resources. The balance between the available and the necessary resources is achieved on the boundary between D_1 and D_2.

If ever the global solution of our problem is in D_2, we are using more processors than necessary for that problem size. Nevertheless, it is desirable (and easy) to know whether this condition arises (and at least to flag it to the user and prescribe a more appropriate p). Observe that when p is reduced the boundary between D_1 and D_2 moves to the interior of the initial domain D_2. In this way each interior point of D_2 can be made a boundary point if we choose an appropriate p. Based on this observation, our general strategy is as follows. Solve P_1 and P_2. Let the optimal values be attained at the points (r_1, s_1) and (r_2, s_2). If the point (r_2, s_2) is in the interior of D_2 and $T_2(r_2, s_2) < T_1(r_1, s_1)$ then reduce the number of processors so that the point (r_2, s_2) falls on the boundary between D_1 and D_2. The point (r_2, s_2) is still optimal for the reduced number of processors and the optimal running time is the same (see Remark 8).

Each problem instance has a certain degree of parallelism, in other words some number of processors p^*, such that no improvement of the performance can be obtained using more than p^* processors. Now we are going to find this number of processors.

Given a problem instance, suppose we have sufficiently many processors (our allocation of tiles allows using of at most N processors). If there are no resource limitations the running time of the program is given by the function T_2. In the terms of our model the domain D_1 becomes empty and the domain D_2 becomes the whole problem domain R. Let (r^*, s^*) be

the point where the function T_2 reaches its minimum over the entire domain R (as explained in Sections 2.2.3 and 2.2.3). Then $T_2(r^*, s^*)$ is the best possible running time for that problem instance. Now we are interested which is the minimal number of processors for which we can obtain the same running time. It suffices to choose a number of processors so that the point (r^*, s^*) falls on the boundary between D_1 and D_2. If we choose less processors, this running time is impossible, because the point (r^*, s^*) falls in the interior of D_1 where $T_1 > T_2$ (see Remark 8). Since the boundary between D_1 and D_2 is composed of the two lines $n = p$ and $m = p\bar{\sigma}$, we have

$$p^* = \min\left\{\frac{N}{s^*}, \frac{M + |\gamma| s^*}{r^* + \gamma s^*}\right\}$$

2.2.5 Optimizing Tile Shape

In this section we study the question of choosing the tile shape (determined by the parameter γ) in an optimal manner, i.e., so that the resulting running time is minimal. In order for the tiling to be valid, all dependence vectors must be in the $\mathbf{cone}(\binom{1}{-\gamma}, \binom{0}{1})$. Let γ_0 be such that the resulting cone is the tightest possible, i.e., such that $\binom{1}{-\gamma_0}$ coincides with one of the extreme dependencies. Then the possible tile shapes are given by $\gamma > \gamma_0$. Let (r_0, s_0) be the optimal tile size with $\gamma = \gamma_0$ and let $(r_0, s_0) \in D_1$ (if necessary the number of processors is reduced as explained in Section 2.2.4). Now consider $T_1(r_0, s_0)$ as a function of γ:

$$T^0(\gamma) = T_1(\gamma; r_0, s_0) = \left(\tfrac{N}{p}|\gamma| + \bar{p}s_0\gamma + \mathbf{const}\right)\tfrac{P}{r_0}$$

Let $\gamma_0 \geq 0$. Then for any $\gamma > \gamma_0$ we obtain

$$T^0(\gamma) - T^0(\gamma_0) = (\gamma - \gamma_0)\left(\tfrac{N}{p} + \bar{p}s_0\right)\tfrac{P}{r_0} > 0$$

i.e., $T^0(\gamma) > T^0(\gamma_0)$. The same is true for arbitrary fixed $(r_0, s_0) \in D_1$. In other words, the minimal value of T_1 is obtained for $\gamma = \gamma_0$ or we have to choose the tightest possible cone.

Now consider the case $\gamma_0 < 0$. In this case the optimal tile shape is not obvious since r_0 and s_0 depend on γ) but we are going to give a gradient-like method to improve it. For any γ, $\gamma_0 < \gamma \leq 0$ we have

$$T^0(\gamma) - T^0(\gamma_0) = (\gamma - \gamma_0)\left(-\tfrac{N}{p} + \bar{p}s_0\right)\tfrac{P}{r_0}$$

Now if $s_0 \geq \frac{N}{p\bar{p}}$ it is impossible to improve the running time in the point (r_0, s_0) by increasing γ and we stop because γ_0 is (locally) optimal. Otherwise the function T^0 decreases when γ is increased. As was shown above we must keep γ negative and we have to leave the point (r_0, s_0) in D_1 respecting the inequality $m \geq p\bar{\sigma}$. In other words we can take $\gamma_1 = \min\{0, \frac{M - pr_0}{\hat{p}s_0}\}$ (where $\hat{p} = p - \operatorname{sgn}\gamma$) and to continue the process iteratively.

At kth step, let we have some γ_k and let $(r_k, s_k) \in D_1$ be the solution of the optimization problem for $\gamma = \gamma_k$.

- If $s_k > \frac{N}{p\bar{p}}$ we have to decrease γ in order to improve the running time at the point (r_0, s_0). We have to respect the constraints $\gamma \geq \gamma_0$ and $\sigma \geq -1$ so we take $\gamma_{k+1} = \max\{-\frac{r_k}{s_k}, \gamma_0\}$.

- If $s_k < \frac{N}{p\bar{p}}$ we have to increase γ. We have to take into account the constraints $\gamma \leq 0$ and $m \geq p\bar{\sigma}$ so we set $\gamma_{k+1} = \min\{0, \frac{M - pr_k}{\hat{p}s_k}\}$.

If $\gamma_{k+1} = \gamma_k$, stop, γ_k is locally optimal. Otherwise set $k = k + 1$ and iterate.

The above algorithm does not ensure globally optimal tile shape but it allows to improve it locally.

Example Let $M = 20000$, $N = 50000$, $p = 8$, $\gamma_0 = -1$, $\alpha = 2.6 \times 10^{-7}$s, $\beta = 4.03 \times 10^{-4}$s, $\tau = 2.02 \times 10^{-7}$s. We obtain $(r_0, s_0) = (1081, 154)$ which is the interior of D_1 and optimal running time $T^0 = 33.48$s. Since $s_0 < \frac{N}{p\bar{p}} = 893$ we set $\gamma_1 = \min\{0, 8.2\} = 0$. Resolving the optimization problem with γ_1 we obtain $(r_1, s_1) = (27, 6250)$ which is on the boundary $n = p$ and optimal running time $T^1 = 33.11$s. Since $s_1 > 893$ we set $\gamma_2 = \max\{-0.004, -1\} = -0.004$. The new solution is the same, $(r_2, s_2) = (27, 6250)$ with running time $T^2 = 32.87$s but it is already at the line $\sigma = -1$ and that is why we have to stop, i.e., we choose $\gamma_3 = \max\{-0.004, -1\} = \gamma_2$ and stop. The obtained value $\gamma = -0.004$ is locally optimal. It turns out to be also globally optimal as checked by enumeration.

An even more drastic improvement is obtained for $M = 200, N = 5000000$, all other parameters as above (although this may seem an extreme example, it is not that unlikely in some biological sequence comparison problems). Here for $\gamma = -1$, the optimal tile size is given by $r = 112, s = 77$, yielding a running time of 53.6 seconds. However, for $\gamma = 0$, (which is optimal) and the corresponding optimal tile size ($r = 1, s = 166410$), the running time is only 33.1 seconds, a gain of nearly 40%.

2.2.6 The sequence global alignment application: Fickett's algorithm

In order to validate the derivations from the model we write and run a code based on an algorithm for solving the *sequence global alignment* problem [37]. This problem originates from the computational molecular biology and nowadays it is one of the most frequently solved problems in this domain. Below we give a brief formalization of the problem and the algorithm.

Let $a = a_1 a_2 \dots a_{|a|}$ and $b = b_1 b_2 \dots b_{|b|}$ be two sequences of characters over some alphabet Ω. An arbitrary insertion of gaps ($\sqcup$) into a and b results into new sequences $\bar{a}$ and $\bar{b}$. If the lengths $|\bar{a}|$ and $|\bar{b}|$ are equal (say of size L) we call $(\bar{a}, \bar{b})$ an *alignment* of the sequences a and b. Let $f(x, y)$ be a function on $\bar{\Omega} \times \bar{\Omega}$, where $\bar{\Omega} = \Omega \bigcup \{\sqcup\}$, such that $f(\sqcup, \sqcup) = 0$. Then $\sum_{i=1}^{L} f(\bar{a}_i, \bar{b}_i)$ is called *score* of the alignment $(\bar{a}, \bar{b})$. The alignment with the minimal score is optimal. The well known dynamic programming algorithm [37] finds $l(|a|, |b|)$, the score of an optimal alignment by the recursion:

$$l(i,j) = \min \begin{cases} l(i-1,j-1) + f(a_i,b_j) \\ l(i,j-1) + f(\sqcup,b_j) \\ l(i-1,j) + f(a_i,\sqcup) \end{cases}$$

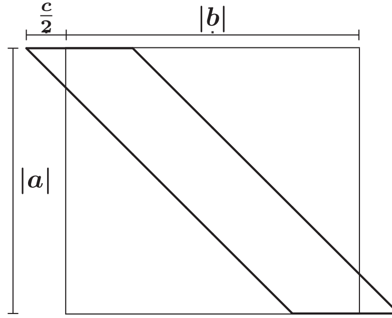


Figure 2.4: Domain of a k-band global alignment of two sequences

The domain of this recursion is $\{(i,j) \mid 1 \le i \le |a|, 1 \le j \le |b|\}$ with $l(i,0)$ and $l(0,j)$ fixed properly, and $l(i,j)$ is the score of the best alignment of the subsequences $a_1 \dots a_i$ and $b_1 \dots b_j$. The time complexity of this algorithm is $O(|a| \times |b|)$. For "k-similar" sequences, which means that the best alignment has less than k gaps, we can adopt the same algorithm discarding some subset of the domain of the recursion. This subset can be determined as follows. The definition of the alignment implies that we must insert at least $|b| - |a|$ gaps in the shorter sequence a plus c gaps into both sequences evenly distributed. Then the set of relevant indices is given by the constrains $-\frac{c}{2} \le j - i \le |b| - |a| + \frac{c}{2}$. These constrains restrict the computations only over the indices belonging to a band-like subset of the rectangular $|a| \times |b|$ index set (see Fig. 2.4). This band could be easily extended to a parallelogram domain, which can be considered as an image of some rectangle transformed by the unimodular matrix H from Section 2.2.2 with $\gamma = 1$. This allows a direct application of the proposed optimization technique to the parallelization of this k-band algorithm. In the next section we present experimental results on a problem instance with $|b| = 55000$ and $|a| = 50000$. The sequence b is randomly generated over the DNA alphabet and a is obtained from b by deleting some 5000 characters at random positions. In this way the best alignment of the two sequences is obvious. The minimal number k of gaps in each alignment is $k = |b| - |a| = 5000$ but we take a broader band choosing $c = 15000$. If we return now to the notation from Section 2.2.2 this corresponds to an index domain with $M = |b| - |a| + c = 20000$ and $N = |a| = 50000$ with 10^9 index points.

2.2.7 Experimental Validation

We performed our computational experiment on the IBM SP2 machine at CINES[4] and on the Cray T3E at the EPCC [5]. The program is written in C. As a communication library we use MPI, following the BSP model and performing a global barrier synchronization at the end of each macrostep.

Determining the Machine-Dependent Parameters

We first detail some of our calibration experiments to determine machine and program parameters in it—α, β and τ. Ideally, it would have been necessary to implement the program using a library such as BSPLib. However, the behavior of tiled loops is simple enough that we can achieve the BSP protocol by simply using the standard `send-receive` libraries and a single additional synchronization.

To measure the communication parameters we use the following method. For a given message size r we measured the communication time of r integers around a ring (each processor sends a message of size r integers to its right neighbor and receives a message of the same size from its left neighbor), followed by a synchronization (this is the additional synchronization corresponding to a BSP macrostep, although it is redundant for our code since the `send-receive` calls ensure the necessary synchronization). We repeated this 1000 times and took the average time, t_r. We found t_r for different values of r (from 1000 to 10000 with step 100). According to our model $t_r = \beta + \tau r$ and we used a least squares fit to obtain β and τ. It turned out that these parameters depend on the number of processors (communication and synchronization of more processors is more expensive, as has been widely reported for the BSP model). For example, when the number of processors is 8 the obtained values are $\beta = 4.03 \times 10^{-4}$s and $\tau = 2.02 \times 10^{-7}$s and when $p = 16$ we have $\beta = 6.05 \times 10^{-4}$s and $\tau = 2.22 \times 10^{-7}$s.

To measure the parameter α, we simply ran the main loop for varying tile size on a single processor and determined the average time to execute each iteration. We obtained $\alpha = 2.6 \times 10^{-7}$s. Note that we *did not* attempt to optimize the performance of the single-processor code, (eg. for caches by loop blocking, strip mining, etc.) Hence, the value of α that we obtained (2.6×10^{-7}s) was surely pessimistically high. More on this later, when we discuss our experimental results.

Experimental Results

Using the computed parameters, we measured the performance of the k-band algorithm. Fig. 2.5(left) gives the theoretical (solid line) and experimental (each point being the average of 10 runs) running times for different s, with r fixed to its theoretical optimal value (1500) and $p = 8$. Fig. 2.5(right) shows the same figures when s is fixed to its optimal value of 110 (note that this

[4] `http://www.cines.fr`: Centre Informatique National de l'Enseignement Supérieur, Montpellier, France, project mih1931

[5] `http://www.epcc.ed.ac.uk/tracs/` Edinburgh Parallel Computing center, grant number HPRI-CT-1999-00026 (the TRACS Programme at EPCC)

requires 57 passes). We observe that (i) the function is quite flat for a large range of r and s; (ii) the two functions match reasonably; and (iii) that the discrepancy is more sensitive to changes of r than s.

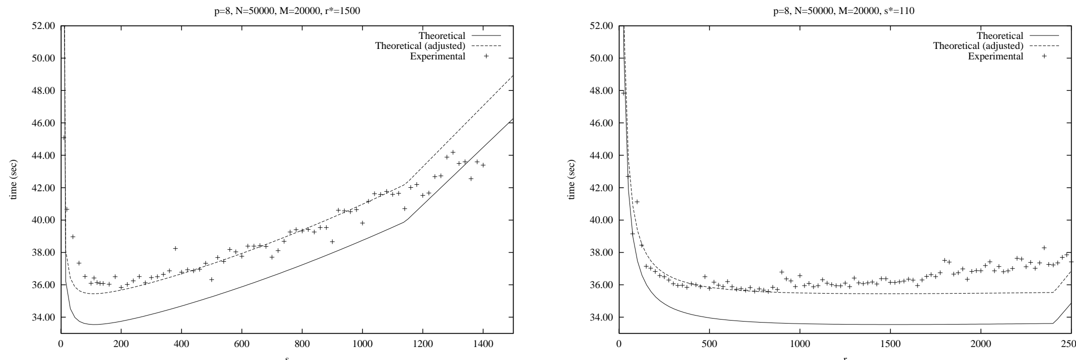


Figure 2.5: Running times for different s and r with the other parameter fixed to the predicted optimum.

The observed discrepancy resembles a shift, plus small periodic fluctuations. We hypothesize that the shift can be explained by cache effects on *each* processor which renders α dependent on the tile size (more precisely, roughly linear with r, given that the order of loop execution *within* a tile has the `j` loop as the inner one). To verify this, we simply plugged a different value of α in the theoretical function (the dashed curve in Fig. 2.5), and this matches the experimental data much more closely (and the variation of the discrepancy with r seems to be linear).

α	r^*	s^*	T_{exp}	α	r^*	s^*	T_{exp}
.01	447	1887	37.11	0.5	1445	83	35.76
.05	230	1647	35.76	0.6	1441	76	35.61
.1	1568	171	35.42	0.7	1428	71	35.74
.2	1505	126	35.38	0.8	1437	66	35.75
.3	1489	104	35.51	0.9	1419	63	35.72
.4	1458	92	35.59	1.0	1414	60	35.91

Table 2.3: Sensitivity to α (shown in μsecs).

Table 2.3 shows the fluctuation of the *predicted* optimal tile size with α, and the experimental running time of the program for these tile sizes. Observe that though the optimal tile size itself changes a lot, the corresponding running time is insensitive, even when α is four times our estimate of α. However, for an optimistically low estimate, we move far from the true optimal.

We also hypothesize that the periodic fluctuations are due to the rational approximation we use, namely the inherent step-wise dependence of the timing function of the tile size. We would have the same effect even for the theoretical timing function in the case of an integer arithmetic.

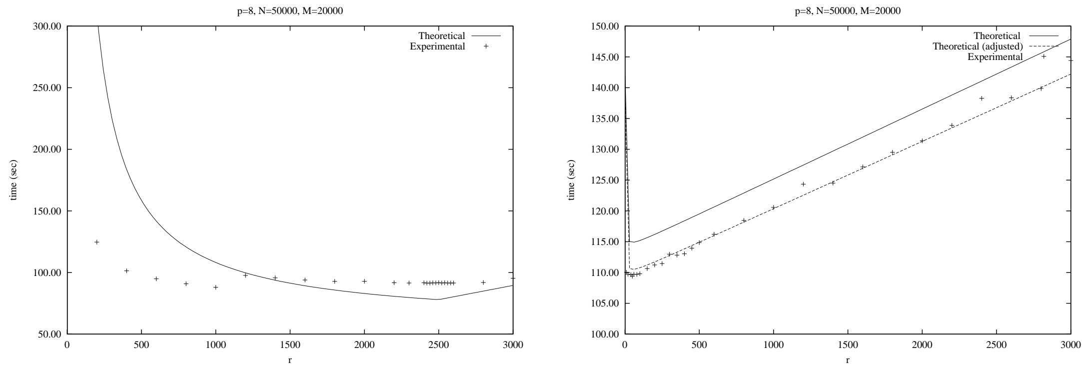


Figure 2.6: Predicted and observed running times, with s fixed for (a) degenerate orthogonal tiling ($s = 1$), and (b) single pass, or block distribution ($s = \frac{N}{p} = 6250$). Although each curve leads to an optimal value which is also more or less corroborated by the experimental data, they are merely local minima. The true optimal (Fig. 2.5) is 2.5 (respectively 3) *times* faster.

Fig. 2.6 gives the predicted and experimental running times for two special cases: $s = 1$, (degenerate orthogonal tiling) and $s = 6250$ (the one-pass solution, or block distribution of tiles to processors). The best running times for these cases are respectively about 2.5 and 3 *times* slower than the optimal running time. Fig. 2.7 gives the running times and the speedup for varying p, each time for the (predicted) optimal r and s, showing nearly linear speedup.

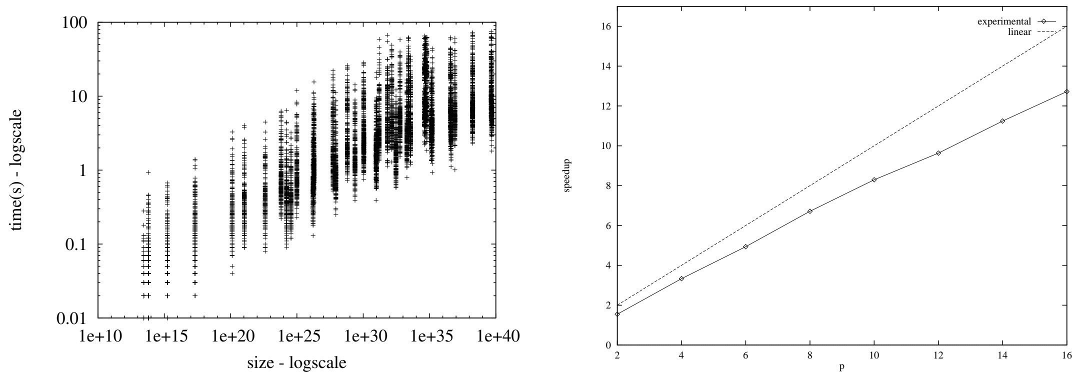


Figure 2.7: Running times and speedup for different p

Experimental validation on other machines

We ran similar experiments on two other parallel machines, a Cray T3E and a Sun HPC 6500. On the former, we measured the following constants: $\alpha = 1.98 \times 10^{-7}$s, $\beta = 1.09 \times 10^{-4}$s and $\tau = 6.6 \times 10^{-8}$s. On this machine we observed an extremely stable behavior of our codes and the experimental running times corroborated our theoretical predictions even closer than on the

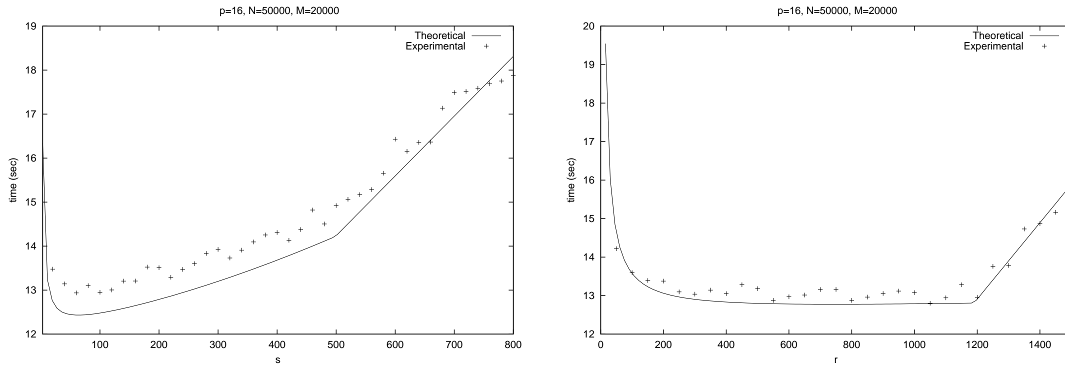


Figure 2.8: Theoretical and experimental results for the parallelization of Fickett's algorithm for comparison of two similar sequences on Cray T3E. Iteration space – 10^9 points, 16 processors. The running time is shown as a function of one of the tile parameters (width, height) when the other one is fixed at its theoretically predicted optimal value.

IBM SP2 (see Fig. 2.8). However, the experiments which we conducted on the shared memory machine Sun HPC 6500 at the EPCC showed that our results, originally developed for distributed memory machines, cannot be directly applied to shared memory architectures.

2.2.8 Related Work

The tiling problem has received the extensive scrutiny of a number of authors, but with differing hypotheses, problem formulations, and cost functions. Therefore, comparisons must be made carefully. Furthermore, although tiling is also extensively used by many authors for data locality, these results are perfectly complementary to the subject of this paper. Additional level(s) of tiling can be used on *each* processor, completely independently of the optimal solution proposed here. The only parameter that this would change for our analysis is α, and as noted above, the results are relatively insensitive to its value. Space constraints preclude a detailed discussion of tiling for locality, literature (e.g. [25, 38]) which is at least as extensive as on tiling for parallelism.

The word tiling was coined by Wolfe [17], though the transformation was described with varying degrees of generality by a number of authors: Schreiber and Dongarra [36], and Ramanujam and Sadayappan [14]. Similar notions had previously been developed in the context of *fixed-size* systolic array synthesis by Fortes & Moldovan [34], and Navarro et al. [35]. The foundations of tiling were formulated in the seminal paper of Irigoin and Triolet [7]. Xue [40] presents an excellent overview of these results as well as a formulation of tiling as a loop transformation, and a precise description of the legality conditions of a tiling.

Although the early papers were more concerned with legality, they also discussed the problem of the choice of the tiling parameters. Different cost functions associated with the tile volume and tile surface area were proposed in some of the above work [7, 36, 14], and also by Boulet et al. [22] and Calland and Risset [24]. However, such a cost function is approximate at many

levels: first, the tile's surface area is but an approximation of the *number of dependence vectors* that cross a tile boundary; this itself is an approximation of the *communication volume* (since the allocation of tiles to processors could result in localizing some of the inter-tile dependences to the same processor); the communication volume itself is but one aspect of the *time* for the communication (e.g. the startup cost, or network latency may often be dominant). It is not surprising that these results (some of which are very elegant mathematically) are not validated experimentally.

Independently, a number of authors address the problem of optimizing a much more realistic cost measure, namely the running time of a tiled program on a parallel machine. For 2-D orthogonal tiling, King et al. [10] consider the case of square tiles (i.e., the height and width of the tiles are both equal). Hiranandani et al. [6] and Palermo et al. [13] consider rectangular tiles, but only for a block distribution of tile-rows to processors (i.e., the tile height is fixed to be $s = \frac{N}{p}$) under slightly different machine models. The latter two results were in the context of prototype compiler implementations, and although the compiler handles arbitrarily deep loops, only the two innermost loops are tiled. Andonov and Rajopadhye [20] resolve the problem when block-cyclic allocations are permitted. Ohta et al. [12] also address the same problem, but incorrectly claim that the block distribution is *always* optimal (for a detailed discussion on this see [21, 23]).

A drawback of such a precise machine model is that of "portability". Even slight variations in a few parameters (e.g. allowing communication computation overlap) significantly alter the optimization problem, and analytic solutions may not be easily found. Andonov et al. generalize the orthogonal tile sizing results to n-D loop nests [21], and also use the BSP model [16], which is portable across a wide range of machines.

We thus have two sets of results, one seeking to optimize the tile shape using very approximate cost functions, and another seeking, for a given, restricted class of tile shapes, the tile size that minimizes a realistic cost function. An interesting result is due to Hodzic and Shang who optimize both shape and size (for n-D loops) but assume an unbounded number of processors [28].

The case of oblique tiling has not received much attention, principally because of the difficulty of analytically modeling the running time of the tiled program. Högstedt et al. [30] develop a cost model (later simplified somewhat by Desprez et al. [26]) for 2-D semi-oblique tiling of parallelogram (or trapezium) shaped domains. They define and identify the rise as a critical parameter, and give analytic formulæ for the idle time of the program (from which its running time can be easily computed). Their model is different from BSP, and in particular, allows communication-computation overlap.

Högstedt et al. [31, 32, 29] further generalize their results to n-D (so-called *rectilinear*) loops, and also determine the optimal schedule using linear programming. This work is complementary to ours—more general in some aspects and more restrictive in others. They consider higher dimensional loops, and multiple levels of tiling, and obtain precise functions for the running time of the program. They do not address the choice of the parameters in its *full* generality, but consider a single independent parameter, the rise, $\sigma = \frac{\gamma s}{r}$. They then suppose that the tile slope γ, the number of processors, p, and the number of passes (and hence the tile height, s) are all given, which reduces their problem to optimally selecting r. Although fixing the first two may be justifiable, fixing s could take them them far from the true optimum (recall that the best single pass solution of the k-band algorithm is about three times slower than the true optimal),

or require an oracle (to guess that 57 passes are needed).

Wonnacott [39] introduces a transformation called *time-skewing*. He considers n-D loops (not necessarily perfectly nested, but those that are amenable to an exact value-based dependence analysis), and addresses the use of semi-oblique tiling in the context of cache performance improvement on sequential and also parallel machines. For parallel machines, although he identifies that tiles larger than a certain size will allow complete overlap of communication and computation, he does not address the problem of the optimal tile size. Moreover, his proposed allocation of tiles to processors could lead to load imbalance.

Agarwal et al. [18] address a more general case in the context of distributed memory machines. They consider affine rather than just uniform dependences, and like Wonnacott , they tackle both cache optimization and parallelization. Their main contribution is that they give an algorithm to estimate an analytic formula that gives what is called the "cache footprint" of a tile. Using this formula, they suggest that a compiler would be able to find the optimal tiling parameters automatically. They also implement such a compiler and show that it gives excellent results, but precise details of its implementation are missing (e.g. for a simple 2-D uniform dependence example in the paper, they use the method of Lagrangian multipliers to analytically find the optimal tiling parameters; it is not clear how their compiler would automatically solve this, and other more complicated problems). As we have seen, the optimization problems even for "simple" cases are fairly involved, and we expect that their compiler uses heuristics for common cases.

BSP versus `send-receive` model

In this section, we compare the results presented here with some of our preliminary results presented at CPC 2000 [19]. There we considered a case study to determine the optimal tile size when the shape is given (i.e., semi-oblique tiling with $\gamma = 1$). The machine model was based on `send-receive` system calls, rather than BSP. This allows for a precise modeling of computation-communication overlap, but renders the non-linear optimization problem difficult, and the proposed solution was approximate. We present here the analytical behavior of the two functions and how their optima compare.

The `send-receive` cost function [19] had parameters that are different from the three-parameter BSP model (the arithmetic time, α is of course common to both). For the `send-receive` model, the communication time is modelled by a message "startup cost" (which is similar in spirit, but not identical, to the BSP synchronization cost, β), and two parameters τ_c and τ_t, which represent "transfer rate" per byte. The first one represents the transfer time that *cannot* be overlapped with computation (e.g. the time to "copy" a message from the user memory space to system memory) and the other is the time which *can* be overlapped (e.g. the actual "transfer" time, during which the sending processor is free to continue on to the next tile, but the receiving processor still has to wait, because the transfer is not yet complete).

We first compared both theoretical curves using the same arithmetic time, $\alpha = 0.26 \times 10^{-6}$, and the values for the two transfer rates as measured[6] in our earlier experiments [19]: $\tau_t =$

[6]Note that the experiments, although on the same IBM SP2 machine, were performed by different people, at different times (possibly under different versions of the OS and programming environment).

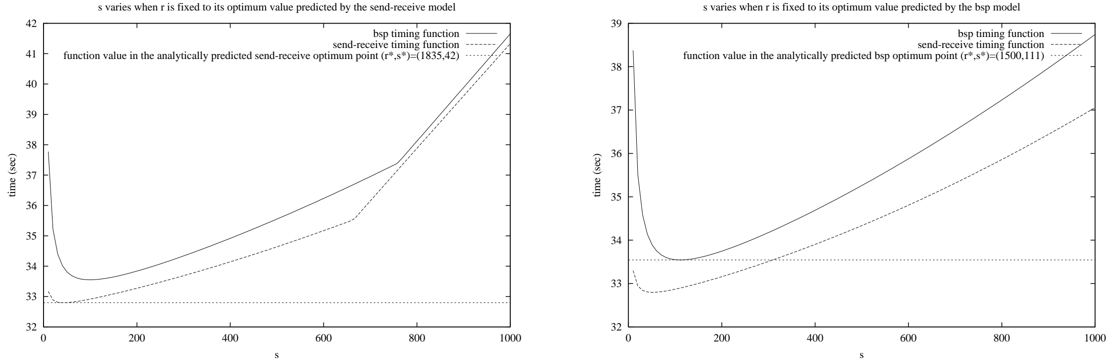


Figure 2.9: Comparison of `send-receive` vs BSP analytical models: running time as a function of s with r fixed to its optimal value as predicted by the `send-receive` (left) and the BSP (right) models. The overall behavior of the curves is very similar. Both analytically predicted function minima (visualized as horizontal lines) correspond perfectly to the minima of the curves.

0.0022×10^{-6}s and $\tau_c = 0.0011 \times 10^{-6}$s respectively. For β we first took **realistic** values: $\beta_{bsp} = 403.0 \times 10^{-6}$s and $\beta_{sr} = 45.0 \times 10^{-6}$s (as measured for the `send-receive` model in [19]). The obtained results are given in Fig. 2.9 and illustrate that: (i) the analytically predicted function minimum match perfectly with the theoretical curve minimum in both models; and (ii) the behavior of both timing functions in general, is very close.

Then we artificially increased the values of both β keeping $\beta_{bsp} = 10 \times \beta_{sr}$. In the very large interval, $0.45 \times 10^{-4} \leq \beta_{sr} < 0.225 \times 10^{-2}$, we did not observe any significant difference in both models. However, for $\beta_{sr} \geq 0.225 \times 10^{-2}$ (see Fig. 2.10), the `send-receive` model formulas (with its approximate solution) accumulate visible error, while the BSP analytical solution give always the true minimum. However, note that these values of β are extremely large and are not realistic (except maybe for very large latency parallel machines, maybe for clusters).

We therefore reiterate that the approximation in [19] is sufficiently good for the simple cases considered ($\gamma = 1$, and a tightly couple parallel machine), but does not hold for arbitrary γ and β. For reasonable values of β the results of both models are very close. The fact that in the current study we chosen to use the relatively simpler BSP model, enabled us to *completely* and *exactly* resolve the general 2-D semi-oblique problem. This choice is not to the detriment of any precision, as confirmed by our computational experiments.

2.2.9 Conclusions

In this section we addressed and resolved the problem of optimally tiling a 2-D rectangular (or parallelogram shaped) iteration domain using (semi) oblique tiles—one set of tile boundaries are parallel to the domain boundary, and the other is oblique. Although the benefits of such tiling (wherever possible) have been well known, there was no systematic method to choose the tiling parameters optimally. We have shown analytically and experimentally, how *all* of them, namely

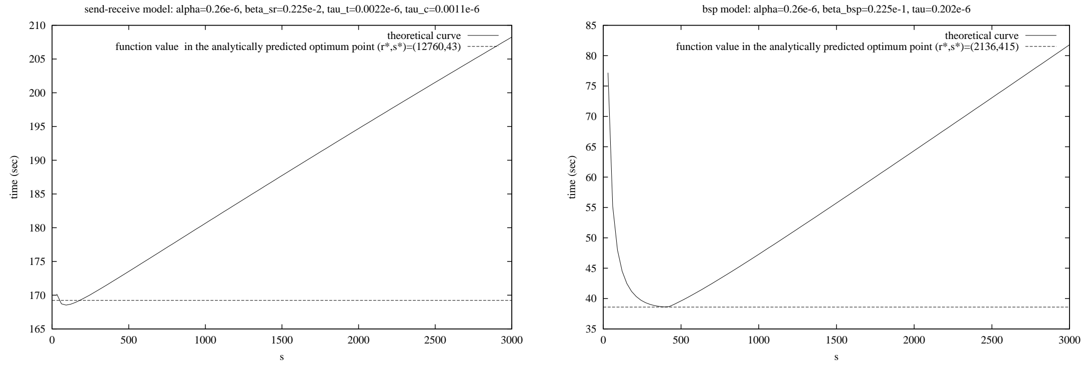


Figure 2.10: Sensitivity to β, which is artificially increased. β_{sr} is fixed to 0.225×10^{-2} and $\beta_{bsp} = 0.225 \times 10^{-1}$. Note that the (approximate) analytically predicted function minimum `send-receive` (the horizontal line on left) does not correspond to the real minimum of the curve. In contrast, we have perfect match in the BSP model (right).

the tile height and width, the tile slope and also the number of processors can be determined so as to minimize the total execution time on a parallel BSP machine. Another key aspect is that our solution is in closed form as simple formulæ, and hence can trivially incorporated into a compiler (or even a run-time system, if one desires to dynamically alter the tile size to exploit fluctuations "in the field"). Moreover, our solution enables us to analyze the impact of the problem and architectural parameters, $p, \alpha, \beta, \tau, N$ and M (some of whose effects are beyond the intuition of even advanced programmers).

We validated our results on a number of examples, both synthetic and real. One of the examples in particular is interesting in its own right, and we have therefore presented it here. We derived a parallel sequence alignment program which achieves optimal performance on an IMP SP2. Similar results are expected for all dynamic programming algorithms with banded domains, and problems such as SOR relaxation.

Note that though tiling is a well studied problem, and incremental gains of successive results are relatively small, our running times are 2.5 and 3 *times* faster than the best performance using previously known techniques.

Limiting ourselves to 2-D loops is not as restrictive as it may seem at first glance. Many problems can profit from the techniques as we have seen. Moreover, for an n-D loop nest, we could choose to tile only the two innermost loops using semi-oblique tiling. The results in this paper would then serve a heuristics since they would not guarantee global optimality, but this is a useful starting point. It is interesting to mention that even for orthogonal tiling, such a strategy is often used (e.g. the Fortran-D compiler at Rice University [6] and the Paradigm project at the University of Illinois [13]).

The main open problems include extensions to fully oblique tiling, to higher dimensions, and to multiple levels of tiling.

Bibliography

[1] R. Andonov, H. Bourzoufi, and S. Rajopadhye. Two-dimensional orthogonal tiling: from theory to practice. In *International Conference on High Performance Computing*. IEEE, December 1996. (to appear).

[2] R. Andonov and S. Rajopadhye. Optimal orthogonal tiling of 2-d iterations. *Journal of Parallel and Distributed Computing*, 45(2):159–165, September 1997.

[3] R. Andonov, N. Yanev, and H. Bourzoufi. Three-dimensional orthogonal tile sizing problem: Mathematical programming approach. In *ASAP 97: International Conference on Application-Specific Systems, Architectures and Processors*, Zurich, Switzerland, July 1997. IEEE, Computer Society Press. (to appear).

[4] S. Bokhari. Communication overheads on the Intel iPSC-860 Hypercube. Technical Report Interim Report 10, NASA ICASE, May 1990.

[5] P. Boulet, A. Darte, T. Risset, and Y. Robert. (pen)-ultimate tiling? *INTEGRATION, the VLSI journal*, 17:33–51, Nov? 1994.

[6] S. Hiranandani, K. Kennedy, and C-W. Tseng. Evaluating compiler optimizations for Fortran D. *Journal Of Parallel and Distributed Computing*, 21:27–45, 1994.

[7] F. Irigoin and R. Triolet. Supernode partitioning. In *15th ACM Symposium on Principles of Programming Languages*, pages 319–328. ACM, Jan 1988.

[8] R. M. Karp, R. E. Miller, and S. V. Winograd. The organization of computations for uniform recurrence equations. *JACM*, 14(3):563–590, July 1967.

[9] C-T. King, W-H. Chou, and L. Ni. Pipelined data-parallel algorithms: Part I–concept and modelling. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):470–485, October 1990.

[10] C-T. King, W-H. Chou, and L. Ni. Pipelined data-parallel algorithms: Part II–design. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):486–499, October 1990.

[11] W. F. McColl. Scalable computing. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000, pages 46–61. Springer Verlag, 1995.

[12] H. Ohta, Y. Saito, M. Kainaga, and H. Ono. Optimal tile size adjsutment in compiling general DOACROSS loop nests. In *International Conference on Supercomputing*, pages 270–279, Barcelona, Spain, July 1995. ACM.

[13] D. Palermo, E. Su, J. Chandy, and P. Banerjee. Communication optimizations used in the PARADIGM compiler for distributed memory multicomputers. In *International Conference on Parallel Processing*, pages xx–yy, St. Charles, IL, August 1994. IEEE.

[14] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for non shared-memory machines. In *Supercomputing 91*, pages 111–120, 1991.

[15] R. Schreiber and J. Dongarra. Automatic blocking of nested loops. Technical Report 90.38, RIACS, NASA Ames Research Center, Aug 1990.

[16] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[17] M. J. Wolfe. Iteration space tiling for memory hierarchies. *Parallel Processing for Scientific Computing (SIAM)*, pages 357–361, 1987.

[18] A. Agarwal, D. Kranz, and V. Natarajan. Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessor. *IEEE Trans. Parallel and Distrib. Syst.*, 6(9):943–962, September 1995.

[19] R. Andonov, P-Y. Calland, S. Niar, S. Rajopadhye, and N. Yanev. First steps towards optimal oblique tiling of two-dimensional iterations. In *CPC2000, Compilers for Parallel Computers*, Aussois, France, January 2000. (Also available at `http://www.univ-valenciennes.fr/LAMIH/ROI/andonov/pub/rech/`).

[20] R. Andonov and S. Rajopadhye. Optimal Orthogonal Tiling of 2-D Iterations. *Journal of Parallel and Distributed Computing*, 45:159–165, September 1997.

[21] R. Andonov, S. Rajopadhye, and N. Yanev. Optimal Orthogonal Tiling . In *Euro-Par'98 Parallel Processing, Lecture Notes in Computer Science, 1470*, pages 480–490. Springer, 1998.

[22] P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-Ultimate Tiling? *Integration, the VLSI Journal*, 17:33–51, 1994.

[23] H. Bourzoufi, S. Boulenoir, and R. Andonov. Tiling and processors allocation for three dimensional iteration space. In *HiPC'99 International Conference on High Performance Computing*, pages 125–129, December, 17-20, 1999, Calcutta, India. ACM SIGARCH.

[24] P-Y. Calland and T. Risset. Precise tiling for uniform loop nests. In P. Cappello, C. Mongenet, G-R. Perrin, P. Quinton, and Y. Robert, editors, *Application Specific Array Processors*, pages 330–337, Strasbourg, France, July 1995. IEEE.

[25] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, june 1995.

[26] F. Desprez, J. Dongarra, F. Rastello, and Y. Robert. Determining the idle time of a tiling: New results. *Journal of Information Science and Engineering*, 14:167–190, 1998.

[27] J. Fickett. Fast optimal alignement. *Nucleic Acids Research*, 12(1):175–179, 1984.

[28] E. Hodzic and W. Shang. On supernode transformation with minimized total running time. *IEEE Transactions on Parallel and Distributed Systems*, 9(5):417–428, May 1998.

[29] K. Högstedt. *Predicting Performance for Tiled Perfectly Nested Loops*. PhD thesis, University of California, San Diego, Department of Computer Science and Engineering, December 1999.

[30] K. Högstedt, L. Carter, and J. Ferrante. Determining the idle time of a tiling. In *Principles of Programming Languages*, Paris, France, January 1997. ACM.

[31] K. Högstedt, L. Carter, and J. Ferrante. Selecting tile shape for minimal execution time. In *SPAA 1999: Eleventh ACM Symposium on Parallel Algorithms and Architectures*, pages 201–211, St. Malo, France, June 1999. ACM.

[32] K. Högstedt, L. Carter, and J. Ferrante. An analysis of the execution time of tiled loops. journal submission, available at `http://www-cse.ucsd.edu/˜ferrante/karjour.ps`, March 2000.

[33] R. M. Karp, R. E. Miller, and S. Winograd. The Organization of Computations for Uniform Recurrence Equations. *JACM*, 14(3):563–590, July 1967.

[34] D. I. Moldovan and J. A. B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Transaction on Computers*, C-35(1):1–12, January 1986.

[35] J. J. Navarro, J. M. Llabería, and Valero. Computing size-independent matrix problems on systolic array processors. In *International Symposium on Computer Architecture*, number 13. IEEE-ACM, May 1986.

[36] R. Schreiber and J. Dongarra. Automatic blocking of nested loops. Technical Report 90.38, RIACS, NASA Ames Research Center, Aug 1990.

[37] J. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology* . ITP, 1997.

[38] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Totonto, CA, june 1991.

[39] D. Wonnacott. Time skewing for parallel computers. Technical Report TR-388, Rutgers University, Department of Computer Science, June 1999.

[40] J. Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):490–424, 1997.

Chapter 3

Knapsack Problems

3.1 A Dynamic Programming Based Reduction Procedure for the Multidimensional 0-1 Knapsack Problem

This section presents a preprocessing procedure for the 0-1 multidimensional knapsack problem. First, a non-increasing sequence of upper bounds is generated by solving LP-relaxations. Then, a non-decreasing sequence of lower bounds is built using dynamic programming. The comparison of the two sequences allows either to prove that the best feasible solution obtained is optimal, or to fix a subset of variables to their optimal values. In addition, a heuristic solution is obtained. Computational experiments with a set of large-scale instances show the efficiency of our reduction scheme. Particularly, it is shown that our approach allows to reduce the CPU time of a leading commercial software.

3.1.1 Introduction

Here, we present a preprocesssing scheme for the 0-1 Multidimensional Knapsack Problem (MKP), which can be formulated as

$$\begin{aligned} \max \quad & \sum_{j\in N} c_j x_j \\ \text{s.t.} \quad & \sum_{j\in N} a_{ij} x_j \le b_i, && i \in M \\ & x_j \in \{0,1\}, && j \in N \end{aligned}$$

where $N = \{1, 2, \dots, n\}$ is the set of items, $M = \{1, 2, \dots, m\}$ is the set of knapsack constraints with capacities b_i, associated weights a_{ij} and profits c_j. The objective is to find a subset of items that yields a maximum profit. We assume that all the data a_{ij}, b_i, c_j are non-negative integers and, without loss of generality, that $c_j > 0$, $b_i > 0$, $a_{ij} \le b_i$ for all $j \in N$ and all $i \in M$ and $\sum_{j\in N} a_{ij} > b_i$ for all $i \in M$.

The MKP is typically encountered in the areas of capital budgeting and resource allocation. The paper by Manne and Markowitz [32] is probably one of the earliest references to this

problem. Other applications include project selection, cutting stock, loading problems, determining the optimal investment policy for the tourism sector of a developing country, and, more recently, delivery of groceries in vehicles with multiple compartments, approval voting, multi-project scheduling, satellite communications. It also appears in a collapsing problem and as a subproblem in large models for allocating processors and data bases in a distributed computer system. Finally, the MKP model is more and more frequently used as a benchmark to compare general purpose methods as metaheuristics.

In this paper we will often use shortcut notations for the problem like

$$\max\{cx:\ a_i x \leq b_i,\ i \in M\ x \in B^n\}$$

or

$$\max\{\sum_{j \in N} c_j x_j:\ \sum_{j \in N} A_j x_j \leq b,\ x_j \in B,\ j \in N\}$$

or simply

$$\max\{cx:\ Ax \leq b,\ x \in B^n\}$$

3.1.2 Related work

The Multidimensional Knapsack Problem generalizes the well-known Knapsack Problem (KP) which deals with only one constraint. As the single constraint case, the MKP is $\mathcal{NP}$-hard but not strongly $\mathcal{NP}$-hard. Polynomial approximation schemes exist for the single knapsack problem and some of them are generalized for the MKP [9, 17]. But while there are fully polynomial approximation schemes for $m = 1$, finding fully polynomial approximations for $m > 1$ is $\mathcal{NP}$-hard [20, 31].

Most of the research on knapsack problems deals with the much simpler single constraint case ($m = 1$). This problem is very well studied and efficient exact and approximate algorithms have been developed for obtaining optimal and near-optimal solutions. An extensive overview of exact and heuristic algorithms is given by Martello and Toth [34]. Randomly generated instances up to 250000 variables may be solved to optimality. Important recent advances can be found in [35, 40].

Exact methods

In contrast, the MKP is significantly harder to solve in practice than the KP. As soon as the number of knapsack constraints increases, exact algorithms usually fail to provide an optimal solution of moderate size instances in a reasonable amount of time. For example, one of the recent versions of CPLEX (6.5.2) is not able to solve difficult problems with 100 variables and 5 constraints to optimality, because of the memory requirements of the search tree [38].

All general 0-1 integer programming techniques may be applied to the MKP [10, 36, 37]. Only the nonnegativity of the coefficients distinguishes this problem from the general 0-1 integer programming problem. The dense constraint matrix and the absence of special constraints, such as generalized upper bounds, special-ordered sets, etc., complicate the development of efficient algorithms for the MKP. That is why relatively few special-purpose algorithms address the MKP.

The development of exact algorithms for 0-1 integer programming began several decades ago [4, 5, 21, 23]. Typically, these approaches start with preprocessing phase, finding lower and upper bounds of the objective value and trying to reduce the problem size by fixation of variables and elimination of constraints. The second phase is an implicit enumeration, which uses the preprocessing information.

The first special-purpose branch and bound algorithm for the MKP is published by Shih [46]. An upper bound is obtained using the minimum of the LP-relaxation values associated with each of the m single constraint knapsack problems. Gavish and Pirkul [19] develop another branch-and-bound algorithm using the surrogate relaxation of the problem. Their algorithm was proved to be faster than the Shih's one for randomly generated instances containing up to 200 variables and 5 constraints. Other interesting results are given by Fréville and Plateau [16] for the case of two constraints ($m = 2$). They solve instances up to 2000 variables by using an efficient preprocessing phase based on an exact solving of the surrogate dual.

Other kinds of special-purpose exact methods, with only limited success being reported, include the dynamic programming based methods [22, 51], an enumerative algorithm based on the Fourier-Motzkin elimination [7] and an iterative scheme using linked LP-relaxations, disjonctive cuts and implicit enumeration [47].

3.1.3 Heuristic methods

A lot of heuristic methods were developed during the last three decades, the best of them being able to provide near optimal solutions for instances with sizes up to $n = 500$ and $m = 30$, in a reasonable amount of CPU time. The main ideas developed for the MKP can be roughly classified into four categories. The greedy algorithms construct a feasible solution step by step by fixing one (or more) variables at each iteration. The first algorithm using the greedy principle was published by Senju and Toyoda [45]. Relaxation-based methods combine local search and information provided by the LP-relaxation, the lagrangean relaxation or the surrogate relaxation, to generate feasible solutions (a comprehensive review of these methods is given by Fréville and Plateau [14]). The last two groups concern metaheuristics, which are very popular methods since the last decade. Simulated annealing, threshold method, tabu search and GRASP are used to enhance the basic local search by overcoming bad local optima. Particularly, good results have been obtained with tabu search embedded into a strategic oscillation scheme [24, 28]. Finally, promising approaches were recently developed concerning genetic algorithms for constrained optimization [12].

3.1.4 Preproccesing techniques

Preprocessing techniques play a fundamental role in the development of efficient integer programming methods. The basic techniques try, among other things, to fix variables, to identify infeasibility and constraint redundancy, to tighten the LP-relaxation by modifying coefficients and by generating strong valid inequalities. Most of the research deals with branch-and-cut algorithms which were successful for solving large scale 0-1 linear programming problems. Seminal ideas to generate strong valid inequalities are given by Crowder et al. [10]. Savelsbergh

[44] presents a framework of basic techniques to improve the representation of a mixed integer programming problem. Constraint pairing [27], probing techniques and logical reduction methods [26], are also very useful techniques to improve enumeration procedures and branch-and-cut methods. More recently, Glover et al. [25] generate cuts from surrogate constraint analysis. Constraint pairing and surrogate analysis are used by Osorio et al. [38] to generate logic cuts. Their computational experiments show that the preprocessing approach improves the performance of CPLEX.

The reduction technique presented in our paper tries to fix some of the variables to their optimal values. The common variable fixation techniques are based on a good lower bound $l = c\underline{x}$ associated with a feasible solution $\underline{x}$. They use the following property:

For any $j \in N$ and for any $\varepsilon \in \{0, 1\}$, if

$$z_j = \max\{cx : \ Ax \leq b, \ x \in B^n, \ x_j = \varepsilon\} \leq l$$

then either $x_j = 1 - \varepsilon$ in any optimal solution of the MKP, or $\underline{x}$ is optimal.

Of course, the above MKP is as hard to solve as the original problem, but it should be clear that any upper bound on z_j suffices. Fayard and Plateau [13] use reduced costs associated with lagrangean relaxation to derive such upper bounds. Fréville and Plateau [15] propose more refined upper bounds induced by additivity of the reduced costs and separation on the optimal basic fractional variable of surrogate relaxations.

In this paper we propose an alternative approach based on dynamic programming and LP upper bounds. The idea of this approach is contained in [50] for the single constraint case. We extend it for the MKP and introduce a number of improvements and new ideas.

3.1.5 Dynamic programming approaches

Dynamic programming (DP) was introduced by Bellman [6]. Toth presents the early DP-based approaches for the KP in [48] and reports numerical experiments with limited success. More recently, Pisinger proposes a DP algorithm for KP which constructs a core problem of minimal size, thus minimizing the sorting and reduction efforts. Hybrid methods, combining dynamic programming and implicit enumeration, were developed for the KP. The first approach was published by Plateau and Elkihel [41]. A recent approach, the so-called combo algorithm, is able to solve very large instances up to 10 000 variables within less than one second, with basically no difference in the solution times of "easy" and "hard" instances [35].

Marsten and Morin [33] proposed the first hybrid method for the MKP, which combines heuristic algorithms, dynamic programming and branch-and-bound approaches. More sophisticated methods, such as a successive sublimation procedure can be found in [30].

3.1.6 General method

Consider the function

$$f(k, g) = \max\{\sum_{j=1}^{k} c_j x_j : \ \sum_{j=1}^{k} A_j x_j \leq g, \ x_j \in \{0, 1\}, \ j = 1, \ldots, k\} \qquad (3.1)$$

Obviously, $f(n, b)$ is the optimal value of the MKP. This value can be found using the recursion

$$f(k, g) = \max\{f(k-1, g), c_k + f(k-1, g - A_k)\}$$

for $k = 1, \ldots, n$ and $A_k \leq g \leq b$ with boundary conditions

$$\begin{aligned} f(0, g) &= 0, & 0 \leq g \leq b \\ f(k, g) &= f(k-1, g), & k = 1, \ldots, n,\ g \geq 0,\ g \not\geq A_k \end{aligned}$$

>From a practical point of view, to solve the MKP problem we have to fill a table of size $n \times b_1 \times \cdots \times b_m$ which can be very memory- and time-consuming even for medium-size instances of the single knapsack problem ($m = 1$). The first step to overcome the problem is to use a sparse representation of the table as described in the next section.

3.1.7 List representation

This kind of representation is used in [1, 29, 48] for single knapsack problems. It is based on the step-wise growing nature of the knapsack function and uses a list containing only the points where the value of the knapsack function changes. In this section we present a natural generalization of this idea for the MKP.

During the solution process we keep a list of $(m+1)$-tuples. Each tuple corresponds to a feasible solution. It represents the objective value and the residual capacities of this solution. We start with the list $\{(0, b)\}$ corresponding to the solution $x = 0$. At kth step, $k = 1, \ldots, n$ we construct new solutions with $x_k = 1$ using the ones constructed at the $(k-1)$th step (if possible) and add them to the list. The process is formally described in Algorithm 1.

Algorithm 1 DP Using List Representation

```
1   L = (0, b)
2   for k = 1, ..., n do
3     L_1 = ∅
4     for (f, g) ∈ L
5       if g ≥ A_k then L_1 = L_1 ∪ {(f + c_k, g − A_k)}
6     end for
7     L = L ∪ L_1
8   end for
```

To restore the optimal solution, it suffices to keep in each element of the list a pointer to the element from which it was obtained or simply the step on which it was obtained.

Even for small problem instances the computational effort of the DP algorithm is considerable. This complexity is in the nature of the method because in fact it generates *all* the feasible solutions of the problem. Hence it is not much better than an explicit enumeration of all 0-1 vectors.

There are different strategies to recognize the solutions which are not perspective and to remove them from the list at earlier stages. The most straightforward technique is to evaluate the new solutions using bounds and then the dynamic programming becomes nothing but a kind of breadth-first branch-and-bound algorithm. We will show a different way to use bounds in Section 3.1.8.

Another way to reduce the size of the list is to use a dominance. This method uses the step-wise growing behavior of the knapsack function (3.1). It is especially suitable for single knapsack problems. Several variants of dominance for single 0-1 knapsack can be found in [1, 29, 48]. The main idea is very simple. Consider two elements of the list $e_1 = (f_1, g_1)$ and $e_2 = (f_2, g_2)$. Suppose that $f_1 \geq f_2$ and $g_1 \geq g_2$, that is the solution corresponding to e_1 has a better objective function and more residual capacity than the one corresponding to e_2. In this case it is clear that any continuation of the solution corresponding to e_1 will be no worse than any continuation of e_2. That is why we can remove e_2 from the list without missing the optimal solution. The dominance is very efficient for single knapsack problems ($m = 1$). Unfortunately this is not the case for $m > 1$, where detecting the dominance is very time consuming and moreover, the dominance occurs very rarely in the solution process. The reason is that for each two numbers a and b, either $a \leq b$ or $b \leq a$, while for vectors we have not such a total order. Our preliminary computational experiments showed that it is not worthwhile to use dominance for the MKP, because the high price paid to detect it is not compensated by considerable elimination of elements of the list.

The analysis of the DP method shows that it is not directly applicable in the context of the MKP. In the next section we show a combination of this method with a bounding technique which allows to fix some of the variables at their optimal values and in many cases even to solve the entire problem.

3.1.8 A new procedure to fix variables

We propose a new method which combines LP-relaxation, upper bounds, and dynamic programming in order to obtain efficient reduction and heuristic procedures for the MKP.

As we have seen in the previous sections, the DP method is not directly applicable. Nevertheless the first several steps of the algorithm are very fast since there are no expensive computations. In this section we will show how to use this advantage.

Main results

Let $\underline{x}$ be a *feasible* solution of our problem

$$z = \max\{cx : \; Ax \leq b, \; x \in B^n\} \tag{3.2}$$

and let u_j, $j = 1, \ldots, n$ be upper bounds of the problems where x_j is fixed at the opposite value of $\underline{x}_j$:

$$u_j \geq z_j = \max\{cx : \; Ax \leq b, \; x_j = 1 - \underline{x}_j, \; x \in B^n\} \tag{3.3}$$

Let us reorder the variables so that

$$u_1 \geq u_2 \geq \cdots \geq u_n \tag{3.4}$$

We introduce the variables in the DP algorithm using this order. At the end of each step of the algorithm we calculate lower bounds l_k by completing with $\underline{x}_{k+1}, \dots, \underline{x}_n$ the best possible entry of the list L so that the obtained solution is feasible. To be more precise

$$l_k = \max_{(f,g)\in L} \{f : \ g \geq \sum_{j=k+1}^{n} A_j \underline{x}_j\} + \sum_{j=k+1}^{n} c_j \underline{x}_j \tag{3.5}$$

The meaning of the bounds l_k is explained by the next proposition (see [35] for proof in the case $m = 1$).

Proposition 1. For each $k = 1, \dots, n$

$$l_k = \max\{cx : \ Ax \leq b, \ x \in B^n, \ x_j = \underline{x}_j, \ j = k+1, \dots, n\} \tag{3.6}$$

>From this proposition it follows that

$$l_1 \leq l_2 \leq \cdots \leq l_n \tag{3.7}$$

In this way we obtain a non-increasing sequence of upper bounds (3.4) and a non-decreasing sequence of lower bounds (3.7). At the moment when the both sequences “meet” each other we are done and the optimal solution is found as shows the next proposition.

Proposition 2. If $l_k \geq u_{k+1}$ for some k then $z = l_k$.

Proof. Let x^* be an optimal solution of (3.2).

1. Let $x^*_j = \underline{x}_j$ for all $j = k+1, \dots, n$. Then from (3.2) and (3.6) the proposition is obvious.

2. Let $x^*_j = 1 - \underline{x}_j$ for some $j = k+1, \dots, n$. Then

$$z = z_j \leq u_j \leq u_{k+1} \leq l_k$$

On the other hand we have $z \geq l_k$, hence $z = l_k$. □

Algorithm 2 gives a formal description of the method. Note that if the list L is sorted by f then l (line 12) can be computed easier. The only essential complication comparing to Algorithm 1 comes from the lines 1-3.

Even in the version with bounds the list may become too long before we fix all the variables (this is in fact the common case). But the first several steps are very fast and they can help to fix many variables. In this way we can use the proposed method as an efficient preprocessing procedure which reduces the size of the problem instead of an exact algorithm. The next proposition explains the mechanism of fixing the variables.

Proposition 3. If $l_s \geq u_k$ for some $s < k$ then there exists an optimal solution x^* of the MKP such that $x^*_j = \underline{x}_j$, $j = k, \dots, n$.

Algorithm 2 DP with bounds

```
1   Find a feasible solution x̲
2   Compute the bounds u_1, ..., u_n
3   Reorder the variables to satisfy (3.4)
4   L = (0, b), p = cx̲, q = Ax̲
5   for k = 1, ..., n do
6     L_1 = ∅
7     for (f, g) ∈ L
8       if g ≥ A_k then L_1 = L_1 ∪ {(f + c_k, g − A_k)}
9     end for
10    L = L ∪ L_1
11    if x̲_k = 1 then p = p − c_k, q = q − A_k
12    l = p + max_{(f,g)∈L}{f : g ≥ q}
13    if l ≥ u_{k+1} then break
14  end for
```

Proof. Consider l_{k-1} and the corresponding solution. Use $l_{k-1} \geq l_s \geq u_k$ and apply Proposition 2. □

The first question is how to choose the moment to stop, in other words, the number of the DP steps to perform. This number of steps, say s, is an algorithm parameter that can be used to tune the code according to the available memory size and the size of the problem. For our computational experiments we used $s = 18 - \lfloor \log_2(m + 2) \rfloor$ and the time to perform the first s DP steps was practically 0. In general, when we choose the number of steps we have to consider the tradeoff between the speed of the algorithm and its quality. In the worst case each step is twice slower than the previous one. On the other hand, the bounds l_k become better at each step and we have the chance to fix more variables.

Another observation that allows to improve the code is that there is no need to compute the bounds l_k at each step. For large-size problems the chances to fix all the variables in several steps are small and that is why we prefer to perform the first s steps without computing the lower bounds (in this case the steps become faster) and to compute only the bound l_s after the sth step.

The last observation is that once we finish and fix some of the variables we can apply the same method for the reduced problem. Since part of the variables are fixed, the upper bounds for the reduced problem will be tighter and it is possible to fix some other portion of variables. We can continue in this way until we fix all the variables or no variable can be fixed.

The final version of our method is outlined in Algorithm 3.

3.1.9 Computing the bounds

In section 3.1.8 we give a general framework of the method without considering how to generate the feasible solution $\underline{x}$ or the upper bounds u_j. The choice of the initial feasible solution and

Algorithm 3 DP with bounds (revised)

```
 1  Find a feasible solution x̲
 2  Compute the bounds u_1, ..., u_n
 3  Reorder the variables to satisfy (3.4)
 4  Compute the number of DP steps s
 5  L = (0, b)
 6  for k = 1, ..., s do
 7    L_1 = ∅
 8    for (f, g) ∈ L
 9      if g ≥ A_k then L_1 = L_1 ∪ {(f + c_k, g − A_k)}
10    end for
11    L = L ∪ L_1
12  end for
13  l = Σ_{j=s+1}^{n} c_j x̲_j + max_{(f,g)∈L}{f : g ≥ Σ_{j=s+1}^{n} A_j x̲_j}
14  if l < u_n then stop (no fixation is possible)
15  else if l ≥ u_{s+1} then stop (all the variables are fixed)
16  else
17    let k be such that u_{k−1} > l ≥ u_k
18    fix x_j = x̲_j, j = k, ..., n
19    apply the same algorithm for the reduced problem
20  end if
```

the method of computing the upper bounds are crucial for the performance of our algorithm. A better initial solution and tighter bounds give the possibility to fix more variables, but on the other hand they are more computationally expensive.

The most commonly upper bounds used in integer programming are the ones produced by linear programming (LP). It seems that they have the best quality/complexity tradeoff. That is why we use this kind of bounds. More precisely

$$u_j = \lfloor \max\{cx : \; Ax \le b, \; 0 \le x \le 1, \; x_j = 1 - \underline{x}_j\} \rfloor \tag{3.8}$$

The next question is how to choose the feasible solution $\underline{x}$. The choice of LP bounds forces the use of the LP solution. Let $\bar{x}$ be the optimal solution of the LP relaxation of the MKP

$$\max\{cx : \; Ax \le b, \; 0 \le x \le 1\} \tag{3.9}$$

and let $u = \lfloor c\bar{x} \rfloor$. This solution contains m basic variables which are typically fractional and the rest $n - m$ variables are 0-1. Now suppose that for some j with integer $\bar{x}_j$ we choose $\underline{x}_j = 1 - \bar{x}_j$. Then the bound u_j will be equal to u and the fixation of the variable x_j will never work. That is why the only reasonable choice is $\underline{x}_j = \bar{x}_j$ whenever $\bar{x}_j$ is integer. For the fractional variables the only restriction is that $\underline{x}$ must be feasible. In the other hand, we are

interested to have a good feasible solution. Let $\mathcal{B}$ be the set of the basic (fractional) variables. We can consider the restricted problem

$$\max\{cx: \ Ax \le b, \ x_j = \bar{x}_j, j \notin \mathcal{B}, \ x_j \in \{0,1\}, j \in \mathcal{B}\}$$

with only m variables and less right hand sides. Since m is assumed to be small we can solve it exactly (using for example the simplest variant of DP) and take the optimal solution as values for $\underline{x}_j$, $j \in \mathcal{B}$.

Finding the bounds u_j, $j = 1, \ldots, n$ involves solving n linear programs which can be time consuming process. But we can use the solution of (3.9) as a starting point for the optimization of (3.8). In this case only a small number of simplex iterations is needed and finding the bounds is very fast.

Example 1. Consider the following instance of the MKP with 10 variables and 2 constraints

$$\left(\begin{array}{c|c} c & \\ \hline A & b \end{array}\right) = \left(\begin{array}{cccccccccc|c} 31 & 92 & 53 & 36 & 44 & 43 & 54 & 44 & 42 & 46 & \\ \hline 19 & 83 & 99 & 56 & 76 & 91 & 62 & 89 & 95 & 16 & 290 \\ 42 & 93 & 49 & 60 & 2 & 8 & 38 & 3 & 24 & 58 & 200 \end{array}\right)$$

The solution of its LP relaxation is

$$\bar{x} = (0.1744, 1, 0, 0, 1, 0, 1, 0.5582, 0, 1)$$

The only solution of the restricted problem

$$\max\{31x_1 + 44x_8: \ 19x_1 + 89x_8 \le 53, \ 42x_1 + 3x_8 \le 9, \ x_1, x_8 \in \{0,1\}\}$$

is $(0, 0)$ and hence the initial feasible solution is

$$\underline{x} = (0, 1, 0, 0, 1, 0, 1, 0, 0, 1)$$

The upper bounds are

$$(u_1, \ldots, u_{10}) = (264, 240, 243, 241, 257, 257, 260, 262, 246, 249)$$

which imposes the following order of the variables

$$x_1, x_8, x_7, x_5, x_6, x_{10}, x_9, x_3, x_4, x_2$$

Table 3.1 gives the list L, p, q and the lower bound l at the end of each phase of the algorithm. The tuple $(p; q)$ is placed next to the list element with which it is combined to obtain the lower bound. After the third iteration of the loop in lines 6-12 of Algorithm 3 we stop because the rest of the upper bounds are less or equal to the current lower bound. The optimal solution is already known. It is

$$x^* = (1, 1, 0, 0, 1, 0, 0, 1, 0, 1)$$

with objective value 257.

var	L	$(p;q)$	l
x_1	$(0;290,200)$ $(31;271,158)$	$(236;237,191)$	236
x_8	$(0;290,200)$ $(31;271,158)$ $(44;201,197)$ $(75;182,155)$	$(236;237,191)$	236
x_7	$(0;290,200)$ $(31;271,158)$ $(44;201,197)$ $(54;228,162)$ $(75;182,155)$ $(85;209,120)$ $(98;139,159)$ $(129;140,117)$	$(182;175,153)$	257

Table 3.1: An example of solving the MKP by Algorithm 3

It is easy to show that our reduction scheme is stronger than previous methods based on reduced costs only [15, 28]. Let $\overline{v}$ be the optimal values of the dual variables associated to the knapsack constraints. Let $r_0 = \sum_{i \in M} \overline{v}_i b_i$, $r_j = \sum_{i \in M} \overline{v}_i a_{ij}$ for $j \in N$,

$$S(\overline{v}): \ \max\{\sum_{j \in N} c_j x_j : \ \sum_{j \in N} r_j x_j \le r_0, \ 0 \le x \le 1\}$$

be the surrogate relaxation with relaxed integrality constraints and let

$$LR(\lambda): \ \lambda r_0 + \max\{\sum_{j \in N} (c_j - \lambda r_j) x_j : \ x \in B^n\}$$

be the lagrangean relaxation of $S(\overline{v})$. It is well known that, first $z(LR(1)) = z(S(\overline{v})) = c\overline{x}$, and second, that if $\lfloor z(LR(1): \ x_j = 1 - \underline{x}_j)\rfloor = \lfloor z(LR(1))\rfloor + (c_j - r_j) \le c\underline{x}$ for any j, such that $c_j - r_j < 0$, then the variable x_j can be fixed to $\underline{x}_j$ (there is a symmetric result for $c_j - r_j > 0$). As $u_j \le \lfloor z(LR(1): \ x_j = 1 - \underline{x}_j)\rfloor$ (since $\lambda = 1$ is not necessarily the optimal multiplier for the reduced problem), our reduction test $u_j \le c\underline{x}$ is stronger than the ones using the reduced costs only.

Example 2. Consider the following example ($n = 15$, $m = 4$) from [38]

$$\left(\begin{array}{rrrrrrrrrrrrrrr|r}
36 & 83 & 59 & 71 & 43 & 67 & 23 & 52 & 93 & 25 & 67 & 89 & 60 & 47 & 64 & \\
\hline
7 & 19 & 30 & 22 & 30 & 44 & 11 & 21 & 35 & 14 & 29 & 18 & 3 & 36 & 42 & 87 \\
3 & 5 & 7 & 35 & 24 & 31 & 25 & 37 & 35 & 25 & 40 & 21 & 7 & 17 & 22 & 75 \\
20 & 33 & 17 & 45 & 12 & 21 & 20 & 2 & 7 & 17 & 21 & 11 & 11 & 9 & 21 & 65 \\
15 & 17 & 9 & 11 & 5 & 5 & 12 & 21 & 17 & 10 & 5 & 13 & 9 & 7 & 13 & 55
\end{array}\right)$$

Our method provides the initial solution $\underline{x}$ with objective value $c\underline{x} = 301$ as shown in second row of Table 3.2 (the same initial solution is used in [38]). The reduced costs of the LP

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\underline{x}_j$	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0
red.costs	311	335	335	315	324	330	*292*	*294*	335	*299*	335	312	313	325	311
u_j^1	308	327	332	*232*	311	*294*	*277*	*277*	309	*285*	*285*	*299*	304	306	*295*
u_j^2	*301*	325	328		311				*290*				*300*	306	
u_j^3		$-\infty$	326		$-\infty$									$-\infty$	

Table 3.2: Upper bounds for Example 2

relaxation allow to fix only the variables x_7, x_8 and x_{10}. Our upper bounds u_j allow to fix 8 variables at their optimal values (in italic on the third row of Table 3.2). If we fix these variables and recompute the upper bounds, three more variables may be fixed. After one more iteration only x_3 is not fixed, but one can easily see that its optimal value is 1. In this way $\underline{x}$ is proved to be an optimal solution.

3.1.10 Experimental results

The implementation of Algorithm 3 is written in C. To obtain a faster code and to compare with the last achievements in 0-1 programming we use the commercial product CPLEX of ILOG. The code is compiled using `gcc` v. 2.96 with option `-O3` and linked to CPLEX callable library. The computational experiments are performed on Compaq AlphaServer DS20 with EV6/500MHz processor.

We tested our approach on several data sets. For the first experiment we generated random instances, following the procedure described in [15]. To generate the coefficients of the matrix A we use two types of probability distributions. The first one is the uniform $U(0, M)$ distribution. The second one, which we call $D(\alpha, p, M)$, has a density function

$$f(x) = \begin{cases} \lambda - \frac{\lambda-\theta}{\alpha}x & \text{if } 0 \leq x \leq \alpha \\ \frac{\theta}{M-\alpha}(M-x) & \text{if } \alpha \leq x \leq M \\ 0 & \text{otherwise} \end{cases}$$

The parameters θ and λ are chosen so that $F(\alpha) = p$ and $F(M) = 1$ where F is the distribution function.

In both cases the objective function c and the right-hand side b are generated in the following way:

$$c_j = \frac{\sum_{i=1}^m A_{ij}}{m} + 500r_j, \qquad r_j \in U(0,1), \qquad j = 1, \dots, n \tag{3.10}$$

$$b_i = r_i \sum_{j=1}^n A_{ij}, \qquad r_i \in U(0,1), \qquad i = 1, \dots, m \tag{3.11}$$

Table 3.3 gives the reduction results on this set of instances. Each row summarizes 10 instances. We give the minimum, maximum and average number of variables rest after the application of the DP algorithm. The average percentage of reduction is also given. For comparison

Distribution	m	n	reduced n (DP algorithm)				F & P
			min	max	avg	avg %	avg %
$U(0, 1000)$	5	100	0	49	15.1	84.9	63.6
		250	20	100	37.2	85.1	62.9
		500	0	51	21.4	95.7	68.1
	10	100	0	39	20.5	79.5	42.0
		250	0	68	17.2	93.1	49.4
		500	0	170	49.4	90.1	56.0
$D(\alpha, p, M)$	5	100	0	94	28.3	71.7	45.0
$\alpha = 100$		250	16	197	57.8	76.9	61.7
$p = 0.9$		500	0	403	111.8	77.6	60.9
$M = 1000$	10	100	0	78	28.1	71.9	19.8
		250	0	230	73.7	70.5	27.9
		500	0	243	82.6	83.5	34.8
$D(\alpha, p, M)$	5	100	0	94	44.9	55.1	56.6
$\alpha = 50$		250	45	234	138.9	44.4	44.2
$p = 0.9$		500	0	492	144.4	71.1	46.8
$M = 1000$	10	100	0	90	33.2	66.8	13.4
		250	0	217	116.5	53.4	12.0
		500	0	451	152.1	69.6	22.3

Table 3.3: Reduction Results

we include the same percentage reported by Fréville and Plateau in [15] after applying their reduction procedure.

In 44 of the 180 instances the problem is completely solved by the DP algorithm. A considerable reduction of the problem size is also observed for the remaining cases.

Table 3.4 shows the efficiency of the algorithm in terms of CPU time and quality of the solution over the same set of instances. To estimate the efficiency, we apply the DP algorithm and then solve the reduced problem using CPLEX. We compare this total solution time with the time to solve the initial problem by CPLEX.

Table 3.4 shows that a negative reduction of CPU times occurs for some of the smaller instances (that is, solving the problem by DP is slower than solving it by general methods). However, when n increases, the effect of the DP algorithm becomes obvious. Looking at the reduction of the problem size, one can expect greater reduction of the solution time. But as it is well known, the reduced (also called core) problem inherits most of the "complexity" of the original problem.

Another measure of efficiency is the quality of the DP solution. Recall that when DP algorithm stops, we have a feasible solution (it's value being l_s) even if it is not able to fix all of the variables. Table 3.4 also gives the relative gap between this solution and the optimal solution of the problem. For more than the half (95) of the cases this gap is 0. For none of the cases the gap is more than 2%.

For the second experiment we use the data set of Chu and Beasley [12], available at `http://msmga.ms.` The matrix coefficients in this set are drawn from uniform $U(0, 1000)$ distribution. The objective coefficients are the same as in (3.11) and the right-had side coefficients are the sum

Distribution	m	n	average time (ms)				%	gap %		
			t_1	t_2	t_3	t_4	r	min	max	avg
$U(0, 1000)$	5	100	78	55	133	153	13.0	0	0.10	0.02
		250	165	298	463	563	17.8	0	0.28	0.05
		500	333	420	753	2187	65.5	0	0.01	0.00
	10	100	92	87	178	208	14.4	0	0.06	0.01
		250	165	417	582	1287	54.8	0	0.08	0.01
		500	465	11678	12143	26648	54.4	0	0.08	0.02
$D(\alpha, p, M)$	5	100	97	25	122	60	-102.8	0	0.84	0.10
$\alpha = 100$		250	177	170	347	353	1.9	0	0.29	0.05
$p = 0.9$		500	427	1527	1953	2427	19.5	0	0.18	0.03
$M = 1000$	10	100	137	42	178	118	-50.7	0	0.52	0.05
		250	273	243	517	607	14.8	0	0.42	0.09
		500	508	1547	2055	3720	44.8	0	0.42	0.07
$D(\alpha, p, M)$	5	100	103	42	145	63	-128.9	0	0.94	0.16
$\alpha = 50$		250	203	307	510	342	-49.3	0	0.75	0.27
$p = 0.9$		500	438	465	903	1048	13.8	0	0.38	0.07
$M = 1000$	10	100	130	93	223	173	-28.8	0	1.92	0.30
		250	257	413	670	817	18.0	0	0.55	0.18
		500	588	980	1568	1750	10.4	0	0.51	0.13

Table 3.4: Time and Gap Results. t_1 – time of DP algorithm; t_2 – time to solve the reduced problem by CPLEX; t_3 – total time to solve the problem, $t_3 = t_1 + t_2$; t_4 – time to solve the initial problem by CPLEX; r – average reduction of the total solution time (in %), $r = \frac{t_4 - t_3}{t_4}$; gap – the gap between the solution given by DP and the optimal solution (in %), gap $= \frac{z - l_s}{z}$.

of the matrix coefficients in the corresponding row, multiplied by some constant α. For each $m = 5, 10, 30$, $n = 10, 250, 500$, and $\alpha = 0.25, 0.5, 0.75$ there are ten instances, or a total of 270 instances. The summarized results for this data set are presented in Table 3.5.

The results for the last data set show that the performance of our method is particularly good for problem instances with a small number of constraints and a big number of variables. To confirm this observation we generated random instances with $m = 2$, n up to 6000, objective coefficients determined by (3.11) and right-hand sides $b_i = 0.25 \sum_{j \in N} a_{ij}$. It is seen that the reduction of the time grows with the size of the problem (for example the total solution time with reduction is about 7 times less than the total solution time without reduction for $n = 4600$. For $n > 4600$ it is impossible to solve the problem without applying the reduction procedure (the search tree generated by CPLEX reaches the memory limit), while the total solution time when applying the reduction procedure is less than 1 min. Observe also that the time t_3 (DP + CPLEX) has relatively robust behavior and grows slowly with n, while the time t_4 (CPLEX only) fluctuates and grows much faster.

Finally, we tested our approach on the data set of Glover and Kochenberger [24], a set of difficult instances, for which the optimal solutions are unknown. Our algorithm fails to reduce the size of these problems, but at least we quickly obtain good feasible solutions. Table 3.7 shows the objective values of the solutions we found and the best known solutions [49]. The time to obtain these solutions is less than 0.1 s for all the cases, while the times reported in [49]

m		5			10			30		
n	α	(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)
100	.25	20.2	9	0.18	2.1	1	0.60	0.0	-	0.61
100	.50	21.5	9	0.09	0.1	5	0.29	0.1	-	0.20
100	.75	30.7	13	0.07	10.6	9	0.09	0.4	-3	0.12
250	.25	22.0	-	0.23	0.8	-	0.35	0.0	-	0.23
250	.50	21.1	-	0.09	0.5	-	0.18	0.0	-	0.14
250	.75	42.0	17	0.04	10.6	-	0.08	0.0	-	0.09
500	.25	23.5	-	0.12	2.4	-	0.22	0.0	-	0.09
500	.50	22.3	-	0.06	1.4	-	0.09	0.0	-	0.05
500	.75	47.8	-	0.02	7.3	-	0.05	0.0	-	0.03

Table 3.5: Results for the data set of Chu and Beasley. (1) – reduction of the number of variables (in %), (2) – reduction of the total solution time (in %), (3) – gap between the best solution and the solution found by DP (in %)

n	t_1	t_2	t_3	t_4	n	t_1	t_2	t_3	t_4
200	0.17	0.11	0.28	0.28	3200	8.47	4.70	13.17	69.97
400	0.30	0.28	0.57	1.55	3400	9.71	17.44	27.15	190.22
600	0.49	0.62	1.11	3.51	3600	11.02	10.36	21.38	122.90
800	0.76	0.92	1.68	8.35	3800	12.87	13.72	26.59	254.01
1000	1.11	1.57	2.68	14.84	4000	13.73	17.23	30.96	264.50
1200	1.45	1.54	2.99	13.29	4200	14.84	14.47	29.31	215.03
1400	1.88	3.13	5.01	26.96	4400	16.15	13.70	29.86	226.67
1600	2.49	2.14	4.63	25.67	4600	18.08	10.59	28.66	268.94
1800	3.32	3.96	7.28	52.81	4800	20.03	7.23	27.26	-
2000	3.60	3.91	7.51	44.18	5000	21.84	14.95	36.79	-
2200	4.16	2.44	6.60	34.14	5200	22.68	25.13	47.81	438.30
2400	5.12	4.00	9.11	61.70	5400	25.32	25.62	50.95	-
2600	6.03	5.48	11.50	74.64	5600	25.86	19.71	45.57	-
2800	6.51	8.04	14.56	90.09	5800	28.88	11.81	40.69	-
3000	7.72	10.46	18.18	133.40	6000	30.18	29.72	59.89	-

Table 3.6: Time results for $m = 2$, $\alpha = 0.25$. t_1 – time of DP algorithm; t_2 – time to solve the reduced problem by CPLEX; t_3 – total time to solve the problem, $t_3 = t_1 + t_2$; t_4 – time to solve the initial problem by CPLEX. The times are in seconds

problem	$n \times m$	found by DP	best known
GK018	100×25	4520	4528
GK019	100×25	3860	3869
GK020	100×25	5175	5180
GK021	100×25	3194	3200
GK022	100×25	2517	2523
GK023	200×15	9226	9235
GK024	500×25	9054	9070

Table 3.7: Results for the data set of Glover an Kochenberger

are considerable (about 300 s for 100 variables, 700 s for 250 variables, and 2000 s for 500 variables).

3.1.11 Conclusion

We presented a new dynamic programming based approach to the MKP. We introduce and use sparse data representation, which decreases memory and time requirements. We use the dynamic programming and the LP relaxation information to derive lower and upper bounds allowing to find the optimal values of some or all the variables. The proposed algorithm is a fast and efficient preprocessing procedure allowing to reduce the problem size and in many cases even to find the optimal solution. Even if no variables are fixed, the procedure remains a robust and very fast heuristic method, providing a feasible solution of good quality by successive improvements of the rounded LP solution. We use the LP bounds but another bounding technique as Lagrangean or surrogate relaxation may be directly plugged in our algorithm.

The experimental results are promising and motivate future work on the improvement of the method. After the end of the DP phase, it is possible to make an attempt to reduce the gap between the lower and the upper bounds by tightening the upper bounds of the last variables using for example partial enumeration. Another reduction of the gap may come from improving the lower bound.

3.2 A Hybrid Algorithm for the Unbounded Knapsack Problem

This section presents a new approach for exactly solving the Unbounded Knapsack Problem (UKP) and proposes a new bound that was proved to dominate the previous bounds on a special class of UKP instances. Integrating bounds within the framework of sparse dynamic programming led to the creation of an efficient and robust hybrid algorithm, called `EDUK2`. This algorithm takes advantage of the majority of the known properties of UKP, particularly the diverse dominance relations and the important periodicity property. Extensive computational results show that, in all but a very few cases, `EDUK2` significantly outperforms both MTU2 and EDUK, the currently available UKP solvers, as well the well-known general purpose mathematical programming optimizer CPLEX of ILOG. These experimental results demonstrate that the class of hard UKP instances needs to be redefined, and the authors offer their insights into the creation of such instances.

3.2.1 Introduction

The knapsack problem is one of the most popular combinatorial optimization problems. Its unbounded version, UKP (also called the integer knapsack), is formulated as follows: there is a knapsack of a *capacity* $c > 0$ and n types of items. Each item of type $i \in N = \{1 \dots n\}$ has a *profit*, $p_i > 0$, and a *weight*, $w_i > 0$. The problem, $\mathbf{UKP}^{c}_{w,p}$, is to fill the knapsack in an optimal way, which is done by solving

$$f(\mathrm{w}, \mathrm{p}, c) = \max \{\mathrm{px} \text{ subject to } \mathrm{wx} \leq c, \mathrm{x} \text{ natural integers vector}\} \tag{3.12}$$

where w, p and x denote vectors of size n.

Many of this problem's properties have been discovered over the last three decades: [2, 8, 18, 37, 34, 42], but no existing solver has yet been developed that benefits from all of them. A detailed and comprehensive state-of-the art discussion the interested reader can find in the very recent monograph [39].

In this paper we introduce *a new upper bound* and determine a UKP family for which this bound is the tightest one known. We also design a new algorithm that combines dynamic programming and branch-and-bound methods to solve UKP[1]. To the best of our knowledge this is the first time that such an approach has been used for UKP. Extensive computational experiments demonstrate the effectiveness of embedding a branch-and-bound algorithm into a dynamic programming.[2] These results also shed light on the case of really hard UKP instances.

A hybrid algorithm, combining dynamic programming and branch-and-bound approaches has been proposed in [55] for the 0/1 knapsack problem, and in [56] for the case of subset-sum problem. The adjective "hybrid" was also used for knapsack problem algorithms in [57] (0/1 knapsack problem) and [54] (0/1 multiple knapsack problem), but this is another kind of hybridization.

---

[1] The U_v bound has already been presented in a research report [43] and partial results have been used by others in [39].

[2] `EDUK2` is free open-source software available at:
`http://download.gna.org/pyasukp/` where it is denoted by PYAsUKP.

The paper is organized as follows. Section 3.2.2 briefly summarizes the basic properties of the problem. Section 3.2.3 presents a new upper bound and its corresponding class. Section 3.2.4 is dedicated to the description of EDUK2 a new algorithm that takes advantage of all known dominance relations and successfully combines them with a variety of bounds. In Section 3.2.5 this algorithm is compared with other available solvers. In Section 3.2.6 we conclude.

3.2.2 A summary of known dominance relations and bounds

The dominance relations between items and bounds allow the size of the search space to be significantly reduced. All the dominance relations, enumerated below, could be derived by the following inequalities:

$$\sum_{j \in J} x_j w_j \le \alpha w_i, \text{ and } \sum_{j \in J} x_j p_j \ge \alpha p_i \text{ for some } \mathbf{x} \in Z_+^n \tag{3.13}$$

where $\alpha \in Z_+$, $J \subseteq N$ and $i \notin J$.

1. *Dominances*

   (a) *Collective Dominance* [2, 52]. The i-th item is **collectively dominated** by J, written as $i \ll J$ iff (3.13) hold when $\alpha = 1$. The verification of this dominance is computationally hard, so it can be used in a dynamic programming approach only. To the best of our knowledge EDUK (Efficient Dynamic programming for UKP) [2] is the only one that makes practical use of this property.

   (b) *Threshold Dominance* [2]. The i-th item is **threshold dominated** by J, written as $i \prec\!\!\prec J$ iff (3.13) hold when $\alpha \ge 1$. This is an obvious generalization of the previous dominance by using instead of single item i a compound one, say α times item i. The smallest such α defines the **threshold** of the item i, written t_i, as $t_i = (\alpha - 1)w_i$.

   The lightest item of those with the greatest profit/weight ratio is called *best item*, written as b. One can trivially show that $t_i \le w_b w_i$ or even sharper inequality $t_i \le lcm(w_b, w_i)$ where $lcm(w_b, w_i)$ is the least common multiple of w_i and w_b.

   (c) *Multiple Dominance* [34]. Item i is **multiply dominated** by j, written as $i \ll_m j$, iff for $J = \{j\}, \alpha = 1, x_j = \lfloor \frac{w_i}{w_j} \rfloor$ the inequations (3.13) hold.

   This dominance could be efficiently used in a preprocessing because it can be detected relatively easily.

   (d) *Modular Dominance* [52]. Item i is **modularly dominated** by j, written as $i \ll_\equiv j$ iff for $J = \{b, j\}, \alpha = 1, w_j = w_i + t w_b, t \le 0$, $x_b = -t, x_j = 1$ the inequalities (3.13) hold.

2. *Bounds*

U_3 **[34]** It is assumed here that the first three items are of the largest profit/weight ratio.

$$\begin{aligned} U_3 &= \max\{U^0, \bar{U}^1\} \qquad (3.14) \\ \text{where: } z' &= \left\lfloor \frac{c}{w_1} \right\rfloor p_1 + \left\lfloor \frac{\bar{c}}{w_2} \right\rfloor p_2;\ \bar{c} = c \mod w_1;\ c' = \bar{c} \mod w_2 \\ U^0 &= z' + \left\lfloor \frac{c' p_3}{w_3} \right\rfloor \\ \bar{U}^1 &= z' + \left\lfloor \left(c' + \left\lceil \frac{w_2 - c'}{w_1} \right\rceil w_1 \right) \frac{p_2}{w_2} - \left\lceil \frac{w_2 - c'}{w_1} \right\rceil p_1 \right\rfloor \end{aligned}$$

U_s [8] . $U_s = c + \left\lfloor \frac{c}{w_1} \right\rfloor \alpha$. This bound is stronger than U_3 for the class of strongly correlated UKP (**SC-UKP**) defined as $p_i = w_i + \alpha$ where $\alpha > 0$. Here item 1 is supposed to be the lightest one.

U_v [43] . $U_v = c + \max\left\{ \frac{(p_i - w_i)}{\lfloor \frac{w_i}{w_1} \rfloor}, i \in N \right\} \left\lfloor \frac{c}{w_1} \right\rfloor$. Here again item 1 is supposed to be the lightest one. This bound is stronger than U_3 for a special class of UKP (namely **SAW-UKP** see def. 7 below)

3.2.3 A new general upper bound for UKP

In following paragraphs, we introduce a new upper bound for the UKP and show that it improves U_v and is not comparable to U_3 in the general case. For the special UKP family, the *SAW-UKP*, which includes the **SC-UKP** class (with $\alpha \geq 0$), this new bound is tighter than the previously known bounds.

Without losing generality it is assumed in this section that: 1 is the lightest item within the set of items with $(p_i - w_i) \geq 0$ (i.e. $\forall i > 1,\ w_1 \leq w_i$ or $p_i \leq w_i$) and $p_1 \geq w_1 + 1$. (If all $p_i - w_i \leq 0$ then assume 1 is the item with the best ratio and by changing p to $\psi\mathrm{p}, \psi = \left\lceil \frac{w_1+1}{p_1} \right\rceil$ we will achieve the goal. If such an equivalent transformation is done, the bound should be divided by ψ). It is also assumed that no item is multiply dominated. Let us define the following terms:

for all $i \neq 1, q_i = \frac{p_i - p_1 \left\lfloor \frac{w_i}{w_1} \right\rfloor}{w_i \mod w_1}$, $q^* = \max_{i \neq 1} \{q_i\}$, $\tau^* = \min\{1, q^*\}$,

$\beta(\tau) = \max_{i \in N} \left\{ \frac{(p_i - \tau w_i)}{\lfloor \frac{w_i}{w_1} \rfloor} \right\}$, $\beta^* = \beta(\tau^*)$.

Theorem 12. *$[U_{\tau^*}]$ for all UKP$^{c}_{w,p}$,* $f(\mathrm{w}, \mathrm{p}, c) \leq U_{\tau^*} = \tau^* c + \beta^* \left\lfloor \frac{c}{w_1} \right\rfloor \leq U_v$

Proof. First, for a fixed τ, $0 \leq \tau \leq 1$,

$$\begin{aligned} \max\{\mathrm{px}, \mathrm{wx} \leq c\} &= \max\{\tau \mathrm{wx} + (\mathrm{p} - \tau \mathrm{w})\mathrm{x}, \mathrm{wx} \leq c\} \\ &\leq \tau c + \max\{(\mathrm{p} - \tau \mathrm{w})\mathrm{x}, \mathrm{wx} \leq c\} \qquad (3.15) \end{aligned}$$

Case $\tau^* = q^* \leq \tau \leq 1$: in this case,

$$q^* = \max_{i \neq 1} \left\{ \frac{p_i - p_1 \left\lfloor \frac{w_i}{w_1} \right\rfloor}{w_i \mod w_1} \right\} = \max_{i \neq 1} \left\{ \frac{p_i - p_1 \left\lfloor \frac{w_i}{w_1} \right\rfloor}{w_i - w_1 \left\lfloor \frac{w_i}{w_1} \right\rfloor} \right\} \leq \tau$$

and therefore

$$\text{for all } i, p_i - \tau w_i \leq \left\lfloor \frac{w_i}{w_1} \right\rfloor (p_1 - \tau w_1) \tag{3.16}$$

Relation (3.16) means that in UKP$^{c}_{\mathrm{w},(\mathrm{p}-\tau\mathrm{w})}$ all items i are multiply dominated by the item 1, and also that $\beta(\tau) = p_1 - \tau w_1$. Thus, $\max\{(\mathrm{p} - \tau\mathrm{w})\mathrm{x}, \mathrm{wx} \leq c\} = \beta(\tau) \left\lfloor \frac{c}{w_1} \right\rfloor$. The function $u(\tau) = \tau c + (p_1 - \tau w_1) \left\lfloor \frac{c}{w_1} \right\rfloor$ is an increasing function, and its minimum is reached for $\tau = \tau^*$. This proves both inequalities of the theorem as $U_v = u(1)$ and $U_{\tau^*} = u(\tau^*)$.

Case $q^* > 1 = \tau^*$: in this case,

$$\begin{aligned} \Sigma_{i=1}^{n}(p_i - w_i)x_i &\leq (p_1 - w_1)x_1 + \beta^* \Sigma_{i=2}^{n} \left\lfloor \frac{w_i}{w_1} \right\rfloor x_i \\ &\leq \beta^* \Sigma_{i=1}^{n} \left\lfloor \frac{w_i}{w_1} \right\rfloor x_i \leq \beta^* \left\lfloor \Sigma_{i=1}^{n} \frac{w_i x_i}{w_1} \right\rfloor \leq \beta^* \left\lfloor \frac{c}{w_1} \right\rfloor \end{aligned}$$

and $U_{\tau^*} = U_v = c + \beta^* \left\lfloor \frac{c}{w_1} \right\rfloor$.

□

Remark 1. When $q^* > 1$, then in almost all cases the (classical) upper-bound $U = \left\lfloor \frac{p_b c}{w_b} \right\rfloor$ is better than U_{τ^*}. It can be obtained from (3.15) by taking $\tau = \max \frac{p_i}{w_i} = \frac{p_b}{w_b}$.

We recall the definition of SAW-UKP first given in [43]

Definition 7. All UKP$^{c}_{w,p}$ instances in which $q^* \leq 1$ are called **SAW-UKP**.

Thus a SAW-UKP verifies: $(p_i - w_i) \leq (p_1 - w_1) \left\lfloor \frac{w_i}{w_1} \right\rfloor$.

The following condition is a necessary condition for UKP$^{c}_{w,p}$ to be a SAW-UKP.

Lemma 2. If UKP$^{c}_{w,p}$ is a SAW-UKP, then the item 1 is the best one.

Proof.

UKP$^{c}_{w,p}$ is a SAW-UKP means that $q^* \leq 1$, i.e. for all $i \in N, q_i = \frac{p_i - p_1 \left\lfloor \frac{w_i}{w_1} \right\rfloor}{w_i \mod w_1} \leq 1$. Then we can derive for all $i \in N$:
$\frac{p_i - p_1 \left\lfloor \frac{w_i}{w_1} \right\rfloor}{w_i - w_1 \left\lfloor \frac{w_i}{w_1} \right\rfloor} \leq 1 \Leftrightarrow (p_i - w_i) \leq (p_1 - w_1) \left\lfloor \frac{w_i}{w_1} \right\rfloor$ which implies
$(p_i - w_i) \leq (p_1 - w_1) \frac{w_i}{w_1} \Leftrightarrow \frac{p_i}{w_i} \leq \frac{p_1}{w_1}$

□

Thus, it can now be established that U_v is tighter than U_3 for this family of UKP.

Theorem 13. *If UKP*$_{w,p}^{c}$ *is a SAW-UKP, then* $U_{\tau^*} \leq U_v \leq U_3$

Proof. It is assumed that the first three items are of the largest ratio, and also that $\frac{p_3}{w_3} \geq 1$ (as above, if it is not the case, changing p to $\psi p, \psi = \lceil \frac{w_3+1}{p_3} \rceil$ achieves the goal).

Since $U_3 = \max\{U^0, \bar{U^1}\}$ it is enough to prove $U_v \leq U^0$. Since $w_2 \geq w_1$, $\left\lfloor \frac{c \mod w_1}{w_2} \right\rfloor = 0$. Thus $z' = \left\lfloor \frac{c}{w_1} \right\rfloor p_1$ and $c' = \bar{c} = c \mod w_1$.

$$\begin{aligned} U^0 &= \left\lfloor \frac{c}{w_1} \right\rfloor p_1 + \left\lfloor c' \frac{p_3}{w_3} \right\rfloor = \left\lfloor \frac{c}{w_1} \right\rfloor p_1 + \left\lfloor (c \mod w_1) \frac{p_3}{w_3} \right\rfloor \\ &\geq \left\lfloor \frac{c}{w_1} \right\rfloor p_1 + (c \mod w_1) = \left\lfloor \frac{c}{w_1} \right\rfloor p_1 + c - \left\lfloor \frac{c}{w_1} \right\rfloor w_1 = \left\lfloor \frac{c}{w_1} \right\rfloor (p_1 - w_1) + c \\ &\geq U_v \end{aligned}$$

□

Example 1 (A Saw UKP)**.** n=7; c=2900;N={1;...;7};
p=[300;580;301;601;605;322;310]; w=[120;245;130;260;310;194;190].

We can compute that q= [_; -4.; 0.1; 0.05; 0.0714285; 0.297297; 0.142857] (remember that q_1 is not define). Hence $q^* \approx 0.297$ and *Example 1* is therefore a SAW-UKP. The bounds are: $U_{\tau^*} = 7205 < U_v = 7220 < U_3 = 7246$. The optimal value is 7202.

Remark 2. For a *non*-SAW-UKP, it is possible for U_v to be stronger than U_3 and vice versa: (c.f. examples 2 and 3)

Example 2 (A non-SAW-UKP with $U_3 < U_v$)**.** case where **Remark 1** applies.
n=6; c=1 619 881; p =[1 001; 1 002; 10 025; 10 026; 11 248; 11 249].
w=[1 000; 1 001; 10 023; 10 024; 11 233; 11 234]. Here we obtain:
q $= [_; \mathbf{1.; 0.6521739; 0.6667; 1.0171673; 1.0170940}]$ and $q^* = 1.017$.
The bounds are $U_3 = 1\,622\,044 < 1\,622\,088 = U_v = U_{\tau^*}$. The optimal profit is $1\,622\,032$ obtainable with $x_1 = 8$; $x_2 = 28$; $x_3 = x_4 = x_6 = 0$; $x_5 = 141$.

Example 3 (A non-SAW-UKP with $U_3 > U_v = U_{\tau^*}$)**.** n=2; c=2 900;
p=[297;309]; w=[120;131]. Here, q=[_;1.0909] hence $q^* \approx 1.09$; and the bounds are $U_{\tau^*} = U_v = 7172 < U_3 = 7175$. The optimal profit is 7140 obtainable with $x_1 = 23$; $x_2 = 1$.

3.2.4 The new algorithm `EDUK2`

The algorithm described below is based on a convenient combination of two basic approaches used in UKP solvers, namely dynamic programming **(DP)** and branch and bound **(*B&B*)** methods.

Dynamic programming (DP)
One of the basic recursions used for solving UKP is

$$f(N, y) = \max_{j \in J_y} \{f(N, y - w_j) + p_j\} \text{ for } J_y \subseteq N \quad (3.17)$$

The eligible set J_y is supposed to contain at least one item i s.t. $x_i > 0$ in some optimal solution to UKP$_N^y$. The cardinality of this set is crucially important for the efficiency of any algorithm based on the above formula(3.17). To the best of our knowledge EDUK is the only solver that uses this recursion with obvious efficiency. The main components of its implementation are the computation of formula (3.17) by slice, a sparse representation of the iteration space, and the use of threshold dominance. Slices are defined as intervals of y, and the sparse representation is based on the stepwise form of the function f. In the following presentation, the couples $(y, f(N, y))$ in which the function value changes will be called *optimal states*.

The *periodicity* property has been described by Gilmore and Gomory [22] as the capacity y^*, called the *periodicity level*, such that for each $y > y^*$, there is an optimal solution with $x_b > 0$. It is well known that, for each UKP$_N^\infty$ such a y^* exists, but its value is not easily detectable. So, although the periodicity property can drastically reduce the search space, it can only be detected in a DP framework, using the following formula:
$y^* < y^+ = \min\{y | \forall y' > y - w_{\max}$, there is an optimal solution with $x_b > 0\}$.

Finally, the fact that **DP** algorithms compute optimal solutions for all values of y below the capacity c allows the recursion to be stopped when the capacity:
$\min\{\max\{\frac{c}{2}, w_{\max}\}, y^+\}$ is reached.

Branch-and-bound (B&B)
Unlike DP, *B&B* algorithms compute an optimal solution *only* for a given capacity, and are dependent on the quality of the computed upper bounds. The `MTU2` algorithm proposed by Martello and Toth [34] uses the upper bound U_3 and the now well known variable reduction scheme: let z be the objective function value of a known feasible solution, and let $\mathcal{U}$ be an upper bound of $f(N, c - w_j) + p_j$; if $\mathcal{U} \leq z$, then either z is optimal or x_j can be set to zero.

Hybridization of DP and B&B
There are several complementary ways to integrate the knowledge of bounds into the DP process.

1. The first approach is to use the variable reduction scheme in a pre-processing stage to reduce the set N.

2. The second approach consists of computing, for each *optimal state* $(y, f(N, y))$, an upper bound $U(c - y)$ for a knapsack with $c - y$ capacity. If $U(c - y) + f(N, y) \leq z$, then the state can be discarded, or in other words, can be *"fathomed by bounds in the B&B context"*. In this way, if a sparse representation is chosen, fewer states must be computed. This designs a *DP with bounds algorithm*.

3. The third approach consists of solving an UKP$^{c}_{core}$ using a *B&B* algorithm in which the *core* set is a subset of the items with the best ratios. If $f(core, c) = U(c)$ then the problem is solved. Otherwise, $f(core, c)$ is used during the *DP with bounds algorithm* described above.

The EDUK2 algorithm outline

The algorithm EDUK2 given below is an hybridization of EDUK with an arbitrary (but efficient) *B&B* algorithm, following the above description. The basic steps of EDUK2 are:

step 1 Compute in $O(n)$ time an upper bound U and an initial feasible solution with value z. Discard from N all items multiply dominated by b. They are detected in linear time.

step 2 For the reduced set of items N, apply the variable reduction scheme in $O(|N|)$ times. Then, select a size C core subset containing the items with the best ratios.

step 3 To improve the lower bound, run a *B&B* algorithm on the core, limiting the algorithm to a maximum of B explored nodes.

step 4 Run the *DP with bounds fathoming states* algorithm (the second integration approach described above).

In the current implementation of EDUK2, a *B&B* similar to the one used by Martello and Toth in MTU2 [34] with the ability to choose the computed upper bound (currently U_v, U_τ or U_3), is used in step 3. An enhanced version of EDUK, which eliminates the fathomed states, operates in step 4.

The EDUK2 parameters, B and C, were experimentally tuned and in the current implementation of the algorithm, their values are C= min{n,max{100, n/100}} and $B = 10000$.

3.2.5 Performance evaluation experiments

Computational experiments were run in order to: (i) test the efficiency of the *B&B*/DP pairing and the state discriminating capacity of the new bounds U_{τ^*} and U_v; (ii) exhibit some actual hard instances. Unfortunately, very few real-life instances of **UKP** have been reported in the literature. For this reason we concentrated our efforts on a set of benchmark tests using: (a) random profit and/or weight generation with some correlation formulae; (b) hard data sets that were specially designed for the *B&B* approach [11].

Very few other UKP solvers are available for comparison with EDUK2. Though Babayev *et al.* [3] have proposed an integer equivalent aggregation and consistency approach (CA) that appears to be an improvement over MTU2. However, this approach does not use the threshold dominance and is incomparable to the one suggested here. Caccetta & Kulanoot [8] have recently described two specialized algorithms for solving two particular classes of UKP: CKU1 for Strongly Correlated UKP (SC-UKP) and CKU2 for Subset Sum Problem(SS-UKP). However, these algorithms are not applicable to the general UKP. Thus, we chose to compare EDUK2 with the only two publicly available solvers: EDUK [2], which is considered to be the most efficient

DP algorithm [39], and MTU2, a *B&B* solver [34]. All instances are free from simply dominated pairs $(p_i, w_i), (p_j, w_j)$, s.t. $p_i \geq p_j$ and $w_j \leq w_j$.

We start by a comparison of the behaviors of MTU2, EDUK and EDUK2 on classic data sets, then we focus on comparing EDUK with EDUK2 on new hard instances not solvable by MTU2. In the case of SAW UKP, we study the impact on the resolution time when using the new bound U_{τ^*} instead of U_v. We also compare EDUK2 with the general purpose solver CPLEX.

Classic data sets

A complete study of the classic UKP benchmarks, where the behaviors of EDUK and MTU2 have been compared, can be found in [2]. Most of these UKP appear to be easy solvable by EDUK2, and for this reason we report only the most interesting subset of the data from our computational results.

EDUK2 and EDUK were written in objective caml 3.08. The respective codes were all run on a Pentium 4, 3.4GHZ with 4GB of RAM, and the time limit for each run was set to 300 sec. MTU2 was executed on the same machine and compiled with g77-3.2. The impact of the bounds was tested by simply substituting the bound U_v in EDUK2 with U_3 in a version called edu$_{U_3}$.

Known "hard" instances

First, we focus on the data sets found to be difficult for MTU2 or EDUK [2].
(A) The SS-UKP instances ($w = p$) are known to be difficult for EDUK.
We built such instances by generating 10 instances for each possible combination of $w_{\min} \in \{100, 500, 1000, 5000, 10000\}$, $w_{\max} \in \{0.5\times10^5; 10^5\}$ and $n \in \{1000; 2000; 5000; 10000\}$ with c randomly generated within $[5 \times 10^5, 10^6]$. We obtain in this manner 400 distinct instances. The average cpu time for the different algorithms was:

EDUK2: $0.045s$; edu$_{U_3}$: $0.045s$; EDUK: 0.474; MTU2: $0.136s$.

According to these results, *EDUK2 is 10 (resp. 3) times faster than EDUK (resp. MTU2).*

We also tested the sensitivity of the algorithms with respect to $w_{\min}$, and the results showed that EDUK2 is much less sensitive to $w_{\min}$ than EDUK. On an average the time for EDUK increased about 80 times when $w_{\min}$ passed from 100 to 10000, while for EDUK2the average increase is 40.

	EDUK2	EDUK	MTU2
$w_{\min} = 100$	0.005s.	0.025s.	0.042s.
$w_{\min} = 10000$	0.2s.	1.82s.	0.25s.

(B) A set of instances of a *Special SC-UKP* was built according to the formula

$$w_i = w_{\min} + i - 1 \quad \text{and} \quad p_i = w_i + \alpha \qquad \text{with } w_{\min} \text{ and } \alpha \text{ given.} \tag{3.18}$$

Chung *et al.* [11] have shown that solving this problem is difficult for *B&B* . We set $w_{\min} = 1 + n(n+1)$ and $n \in \{50; 100; 200; 300; 500\}$, and used both a negative and a positive

value for α. For each set, we generated 30 instances with a capacity taken randomly from the interval $[10^6, 10^7]$.

$\alpha > 0$ The average time needed to solve the 150 instances was:

`EDUK2`: 3.32s, $\mathtt{edu}_{U_3}$: 3.37s; `EDUK`: 4.29s.

`MTU2` was able to solve only 9 of the 60 instances with $n \in \{50; 100\}$ and none for $n > 100$. NB: *these problems are SAW-UKP.*

$\alpha < 0$ The average time needed to solve the 150 instances was:

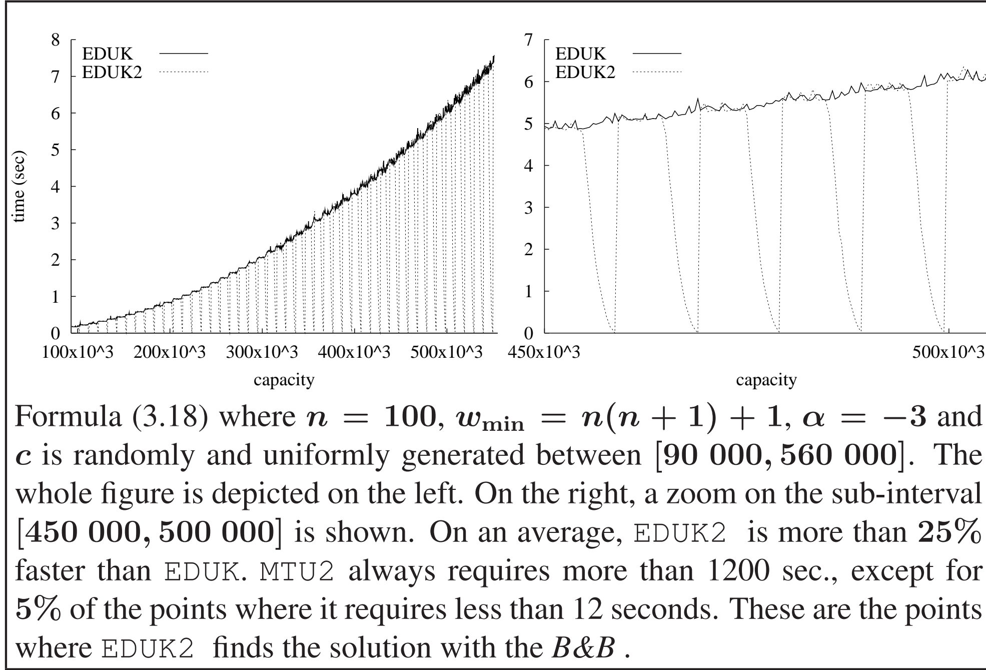
`EDUK2`: 6.01s; $\mathtt{edu}_{U_3}$:5.93s; `EDUK`:8.65s.

`MTU2` was able to solve only 10 of the 60 instances with $n \in \{50; 100\}$ and none for $n > 100$. NB: *these problems are not SAW-UKP.*

From these results, it appears that *`EDUK2` is 1.3 (resp. 1.45) times faster than `EDUK`* when $\alpha > 0$ (resp. <0). As expected, these instances were hard for MTU2.

Sensitivity to variations in the capacity: a comparison with `EDUK`

The *B&B* algorithms are known to be very sensitive to variations in the capacity. DP algorithms, on the other hand, are known to be robust, but with computational time increases linearly with the capacity value. Our computational experiments show that `EDUK2` inherits the good properties of both *B&B* and DP. Data presented in **Fig. 3.1** were generated by formula (3.18) as a *Special SC-UKP*. We observe that `EDUK2`'s overall computational time is upper-bounded by the minimum between the time taken by the pseudo-polynomial DP approach and the time for *B&B*. `EDUK2` has lost the regular behavior typical of `EDUK`, but this is in its favor, since the time ratio $\frac{\mathtt{EDUK}(i)}{\mathtt{EDUK2}\ (i)} \geq 1$ is valid for any instance i, and reaches a value of 2.5 for more than 12% of the c values.



Formula (3.18) where $n = 100$, $w_{\min} = n(n+1)+1$, $\alpha = -3$ and c is randomly and uniformly generated between $[90\ 000, 560\ 000]$. The whole figure is depicted on the left. On the right, a zoom on the sub-interval $[450\ 000, 500\ 000]$ is shown. On an average, `EDUK2` is more than 25% faster than `EDUK`. `MTU2` always requires more than 1200 sec., except for 5% of the points where it requires less than 12 seconds. These are the points where `EDUK2` finds the solution with the *B&B* .

Figure 3.1: Capacity sensitivity of `EDUK2` and `EDUK`

SAW-UKP instances

This class contains 880 SAW-UKP instances according to **definition 7**, generated using the following parameters: $c = \frac{1}{10}\sum w$, $w_{\min} \in \{100; 200; 500; 1000\}$,
$w_{\max} \in \{10000; 100000; 1000000\}$ and $n \in \{1000; 2000; 5000; 10000\}$. For each of the 44 possible parameter combinations[3], we randomly generated 20 instances, for which we obtained the following average times:

`EDUK2`: 0.129s, $\texttt{edu}_{U_3}$: 0.252s; `EDUK`: 0.610s.

We therefore observe that for this family *`EDUK2` is about 5 times faster than `EDUK`*, and using U_v instead of U_3 accelerates `EDUK2` by a factor of 2. A comparison of efficiency between U_v and U_τ is reported in the section 3.2.5.

Due to arithmetic overflow `MTU2` was run only 200 instances with $w_{\max} = 1000$. For 95 of these instances, it reached the time limit of 300 seconds.

EDUK2 versus CPLEX versus EDUK

In this section we compare `EDUK2` and EDUK with one of the most popular general purpose mathematical programming optimizers CPLEX of ILOG[4]. For this purpose we focus on three types of problems, each defined by a pair $(\mathbf{w}, \mathbf{p})$ and a wide set of capacity. Each instance has been solved by `EDUK2` , EDUK and CPLEX, and the respective required times are reported in

[3]The combination $n = w_{\max} = 10000$ is not possible due to simple dominance.

[4]We used version 10.0.1 of CPLEX

Fig.3.2-Fig.3.7. The first two problems were generated by formula (3.18) with parameters as given above the graphics. As discussed in section 3.2.5, they are known to be difficult for *B&B*.

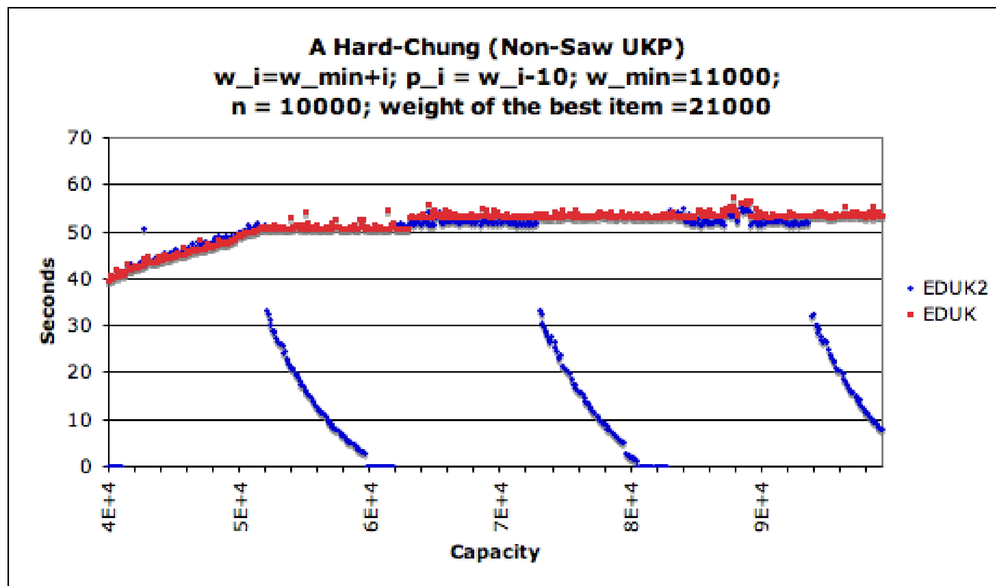


Figure 3.2: `EDUK2` versus EDUK on a set of 540 hard non-saw UKP instances

For the first problem, (**Fig.3.2-Fig.3.3**), the capacity varies from 4×10^4 to 10^5 generating 540 instances. **Fig. 3.2** compares the behavior of `EDUK2` with the one of EDUK. As in **Fig. 3.1**, EDUK behaves regularly, while the shape of `EDUK2`'s curve permits to distinguish three different cases that alternate periodically: i) a high plateau where both algorithms need the same time since the solution was found by dynamic programming; ii) a low plateau where the solution was found by the bound provided in the B&B phase. `EDUK2` computes the results instantaneously being 50 times faster than EDUK. iii) intermediate stage where the solution was found due to B&B/DP hybridization. The weight of the best item (here 21000) is a period of any of these three stages in the behavior of `EDUK2`.

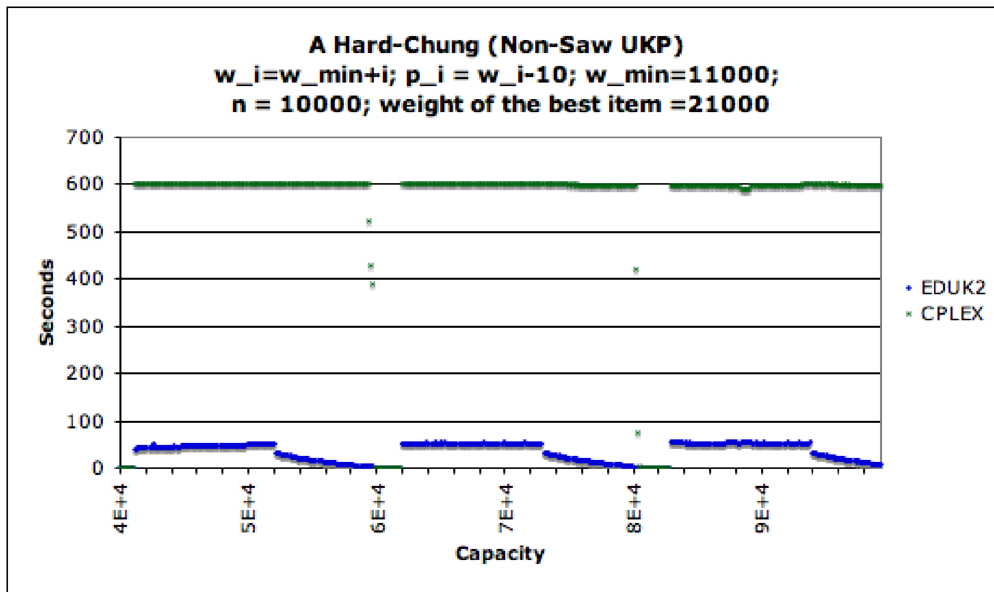


Figure 3.3: EDUK2 versus CPLEX on a set of 540 hard non-saw UKP instances

Next experiment was dedicated to EDUK2 versus CPLEX comparison. Running time for CPLEX was bounded by 600 seconds. **Fig. 3.3** illustrates that for this lapse of time and on the same data set CPLEX succeeds to solve about 12% of the instances. The solved instances have their capacity in a narrow neighborhood of a multiple of the best item weight. This is clearly seen on **Fig. 3.3**. These instances correspond in fact to the low plateau ii) above described. In the dominant case, 88%, EDUK2 is more than 100 times faster than CPLEX.

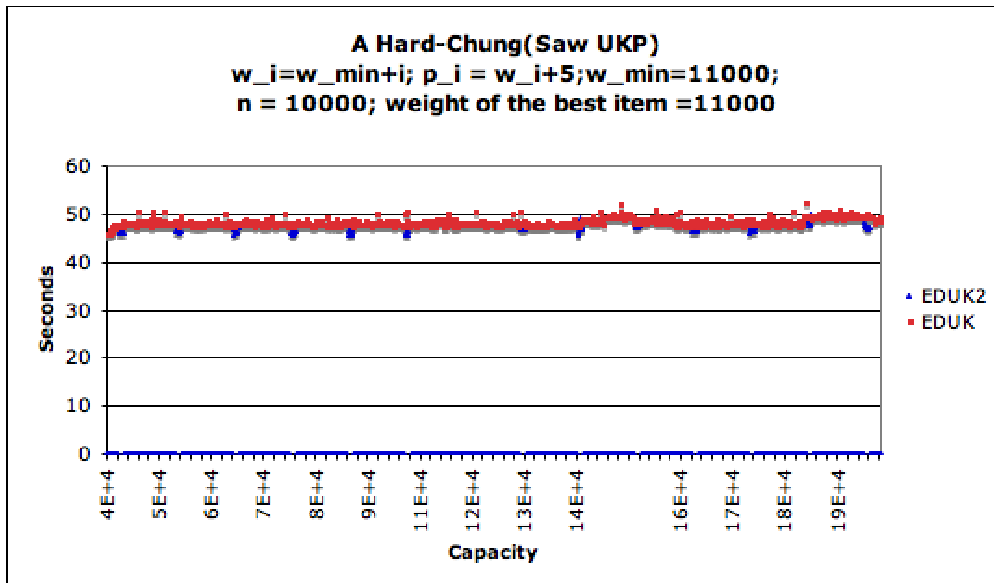


Figure 3.4: EDUK2 versus EDUK on a set of 1350 hard saw UKP instances

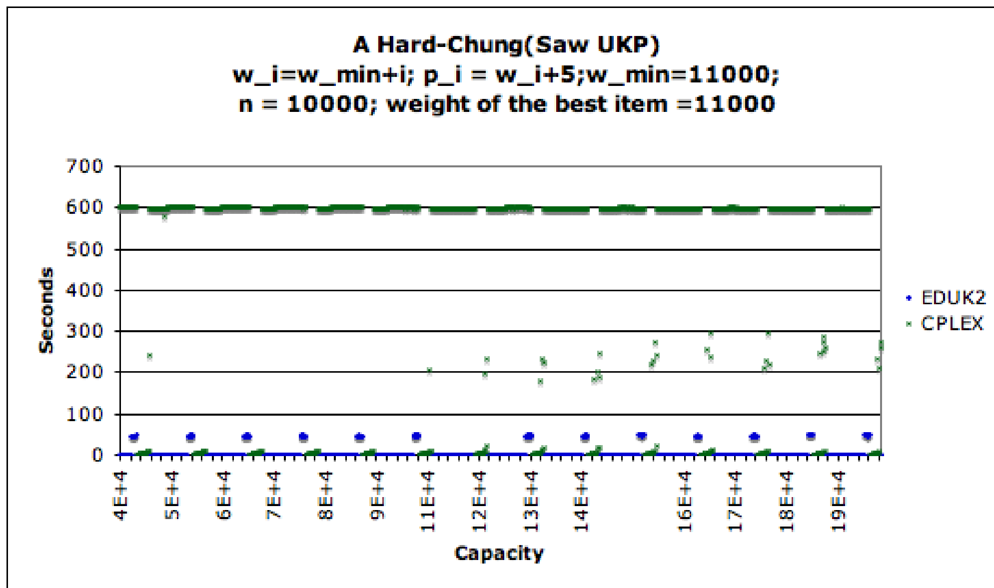


Figure 3.5: EDUK2 versus CPLEX on a set of 1350 hard saw UKP instances

Figures 3.4 and 3.5 illustrate the same comparison in case of SAW UKP. Here the capacity varies from 4×10^4 to 2×10^5 generating 1350 instances . As theoretically expected, due to the new bound, `EDUK2` instantaneously finds the solution (except for few values just below a multiple of the weight of the best item). We observe similar phenomena as before: again `EDUK2` is about 50 times faster than EDUK (with very few exceptions). CPLEX succeeds to solve about 22% of the instances for the given lapse of time. These instances correspond to a multiple of the best item weight. Outside these rare cases `EDUK2`, is more than 100 times faster than CPLEX.

Next experiment focusses on randomly generated instances being non-SAW UKP and without simple dominance. We generated 2700 such instances with parameters as described in figures 3.7 and 3.6 and a capacity varying in the interval $11 \times 10^4 - 43 \times 10^4$. **Fig. 3.6** compares `EDUK2` versus EDUK on this data set. The behavior of both algorithms is very similar to the one observed on **Fig. 3.2**: the running time of `EDUK2` has a typical saw shape with minimums around the multiples of the best item and upper-bounded by the time of EDUK. **Fig. 3.7** illustrates `EDUK2` versus CPLEX behavior. CPLEX succeeds to solve all instances with a capacity less than 21×10^4 and those with a capacity close to a multiple of the best item, but fails for all other instances with a capacity larger than 21×10^4. For all these instances `EDUK2` is as at least 100 times faster than CPLEX.

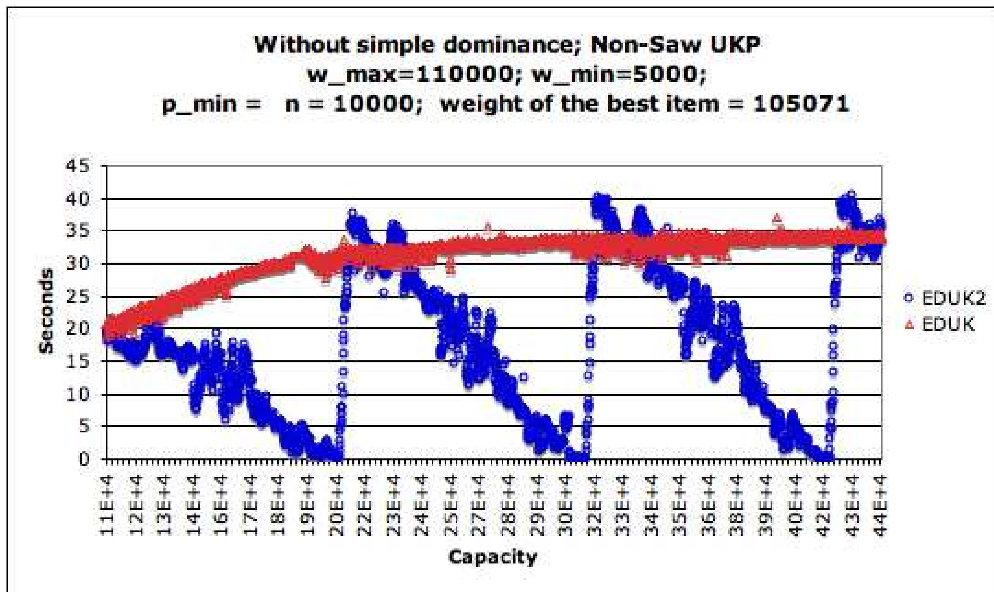


Figure 3.6: EDUK2 versus EDUK on a set of 2700 randomly generated UKP instances

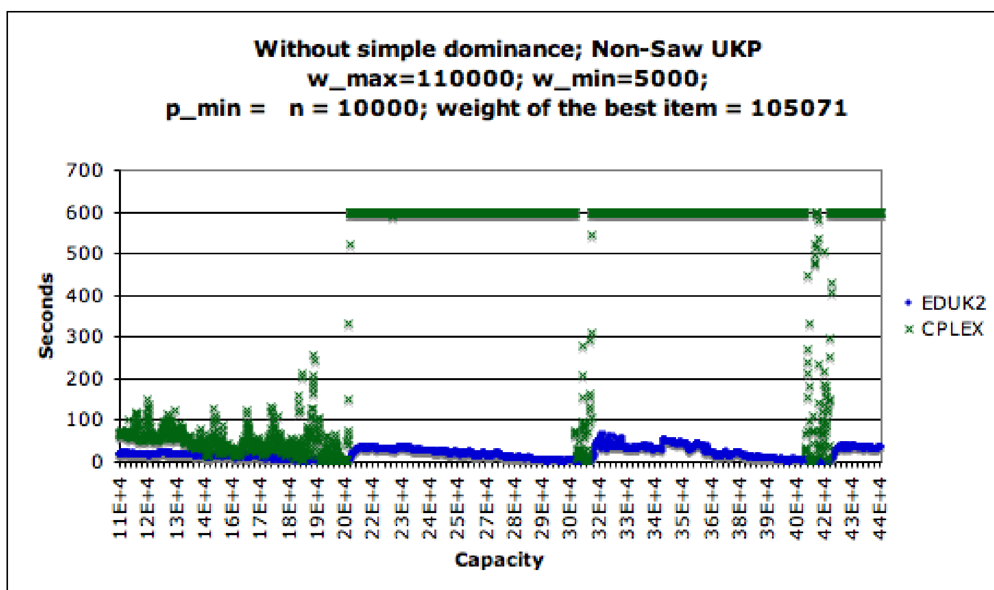


Figure 3.7: EDUK2 versus CPLEX on a set of 2700 randomly generated UKP instances

Do hard UKP instances really exist?

Based on these results, one is inclined to conclude –wrongly– that **UKP** are easy to solve. It is important to remind that, in the above experiments, the considered instances are of moderate size only. A real-life problem of the same size would indeed be easy to solve. However, real

problems may have large coefficients, which makes necessary testing the solvers' behavior on such data sets.

New hard UKP instances

In order to construct difficult instances, we considered data sets with large coefficients and/or large number of items. Because MTU2 cannot be used for such instances because of arithmetic overflow, we restricted our comparisons to EDUK, edu_{U_3} and EDUK2. For such data sets EDUK2 and edu_{U_3} benefit of the num ocaml library, which provides exact unlimited integer arithmetic to compute the bounds. All the runs were done on a Pentium IV Xeon , 2.8GHZ with 3GB of RAM. CPU time was limited to one hour per instance. If this time limit was reached, we reported 3600 sec. in order to compute the average[5]. We use the notation $x\overline{n}$ to denote $x \times 10^{u+1} + n$, where $0 < \lfloor \frac{n}{10^u} \rfloor < 10$ (e.g. $n = 213, 4\overline{n} = 4213$).

In order to measure the improvement of EDUK2 in respect to EDUK and the influence of the new bound, we use the following metrics:

for each of the three algorithms :

- **nmd**: number of non-multiply dominated items (step 1 of EDUK2);
- **ncd**: number of non-collectively dominated items (as computed by EDUK);
- **cpu**: running CPU time in seconds;
- **rp**: denotes the ratio $\frac{y^+}{c}$ where y^+ is the capacity level where the algorithm detects that the periodicity level y^* is reached.

for EDUK2 and edu_{U_3} :

- **vrs**: number of items eliminated in the variable reduction step [6];
- **wdp**: number of instances for which the optimal solution was found without using DP (steps 1 to 3);
- **rst**: ratio of the number of states in the DP phase (step 4 of EDUK2) with respect to the number of states for EDUK.

In the tables below, the reported value in the **nmd**, **ncd**, **cpu** columns is the average for the number of instances; the value in the **wdp** columns refers to the total number of instances; the value in the **vrs**, **rp** and **rst** columns, reports the average for the number of instances for which the algorithm enters the DP phase.

Instances known to be difficult for *B&B*

We generated large data sets using the formula (3.18). It is easy to see that for such a data set, ncd $= \min(n, w_{\min})$. For a given n, the formula determines n pairs (w_i, p_i), and we generated 20 different values for c (Fig. 3.8).

EDUK had some trouble in solving these sets and was unable to solve the 20 problems with $\alpha = -5, n = 10^4$, and $w_{\min} = 11 \times 10^4$ in less than one hour. In one special case, where

[5] The notation $t(k)$ means that the average time is t sec., with k instances reaching the time limit.

[6] The notation $x(y)$ in this column means that for y instances the optimal value was founded in this step and x is the average of the number of reduced variables in the *other* instances.

20 instances per line					EDUK2					edu_{U_3}					EDUK	
α	n	w_{min}	nmd	ncd	cpu	vrs	wdp	rst	rp	cpu	vrs	wdp	rst	rp	cpu	rp
5	5	10	n	n	21.77	0(13)	13	0.29	0.047	37.81	642(3)	3	0.38	0.069	80.06	0.108
		15	n	n	46.57	0(8)	8	0.34	0.099	52.29	83(7)	7	0.56	0.141	111.28	0.188
		50	n	n	154.19	0(2)	2	0.55	0.470	156.63	0(2)	2	0.68	0.555	261.29	0.661
5	10	10	n	n	0.03	0(20)	20	-	-	135.22	2420(3)	3	0.54	0.007	336.70	0.008
		50	n	n	344.12	0(6)	6	0.26	0.037	367.94	0(6)	6	0.41	0.052	915.11	0.079
		110	n	n	771.53	0(2)	2	0.20	0.112	816.90	0(2)	2	0.26	0.139	2808.50	0.300
-5	5	10	n	n	64.82	44(6)	6	0.78	0.091	65.14	44(6)	6	0.78	0.091	113.67	0.108
		15	n	n	104.89	11(2)	2	0.61	0.091	104.62	11(2)	2	0.61	0.091	183.31	0.188
		50	n	n	232.26	0(8)	8	0.86	0.650	231.13	0(8)	8	0.86	0.650	447.40	0.660
-5	10	10	n	n	167.26	1317(4)	4	0.67	0.009	170.34	1317(4)	4	0.67	0.009	317.01	0.009
		50	n	n	508.37	0(6)	6	0.45	0.058	511.36	0(6)	6	0.45	0.058	1539.74	0.079
		110	n	n	1401.(3)	0(4)	4	-	0.124	1394.(3)	0(4)	4	-	0.124	(20)	-

c is randomly generated between $[20\overline{n}; 100\overline{n}]$.

Figure 3.8: Large hard data sets created using formula (3.18). Data from **n** and $\mathbf{w_{min}}$ columns should be multiplied by 10^3 to get the real value.

$\alpha = 5$ and $n = w_{min} = 10000$, the solution was always found immediately in the initial variable reduction step, using the bound U_v. Excluding these two special sets, EDUK2 is on an average from 1.7 to 3.7 times faster than EDUK. Note that for all these instances, the optimal solution was found by EDUK2 and edu_{U_3} either in the variable reduction step, either in the DP phase but never in the *B&B* step. Note that EDUK2 was 1.01 to 1.7 times faster than edu_{U_3} when $\alpha > 0$ (these instances belong to the SAW-UKP family). However, in the case $\alpha < 0$ EDUK2 and edu_{U_3} behave very similarly.

Data sets without simply dominated items

For the data in (Fig. 3.9), w_i were randomly generated between $[w_{min}; w_{max}]$, and p_i values were generated using $p_1 \in [w_1; w_1 + 500]$, $p_i \in [p_{(i-1)} + 1; p_{(i-1)} + 125]$. c was randomly generated between $[w_{max}; 2 \times 10^6]$. Clearly, for these instances, the number of non-collectively

200 instances per line					EDUK2					edu_{U_3}					EDUK	
n	w_{min}	w_{max}	nmd	ncd	cpu	vrs	wdp	rst	rp	cpu	vrs	wdp	rst	rp	cpu	rp
20	5	$1\overline{n}$	20000	19999	317.27	10989	3	0.49	0.703	317.47	10989	3	0.49	0.703	713.23	0.676
50	5	$1\overline{n}$	46569	10052	38.66	44106	6	0.54	0.441	38.61	44106	6	0.54	0.441	208.48	0.485
20	20	$10\overline{n}$	19985	16851	118.65	11121	2	0.25	0.989	120.47	11121	2	0.25	0.989	344.81	0.994
50	20	$10\overline{n}$	50000	49999	1026.(1)	28881	0	0.22	1.00	1015.(1)	28881	0	0.22	1.00	2959.(8)	1.00
20	50	$10\overline{n}$	19999	19924	126.(2)	9955	0	0.23	1.	210.(1)	9997	0	0.23	1	504.	1
50	50	$10\overline{n}$	50000	49999	1553.(1)	22827	0	0.32	1.00	1555.(1)	26981	0	0.32	1.00	3289.(51)	1.00

Figure 3.9: Data sets without simply dominated items. The data from **n** and $\mathbf{w_{min}}$ columns should be multiplied by 10^3 to get the real value.

dominated items determines the efficiency of the algorithms. With this kind of data generation, where $c < 2 \times w_{max}$ and n is large enough, the periodicity property does not help ($rp \approx 1$). EDUK2 outperforms significantly EDUK and slightly edu_{U_3} .

SAW data sets

SAW-UKP instances (according to **definition 7**) were generated with the following parameters: $w_{\max} = 1\overline{n}$, $p_{\max} = 2\overline{n}$ and $c \in [w_{\max}; 10\overline{n}]$. For each pair $(n, w_{\min})$, we generated ***nbi*** *distinct instances* (Fig. 3.10). The tight and computationally cheap upper-bound for these sets

					EDUK2					$\texttt{edu}_{U_3}$					EDUK	
n	**$\mathbf{w_{min}}$**	**nbi**	**nmd**	**ncd**	**cpu**	**vrs**	**wdp**	**rst**	**rp**	**cpu**	**vrs**	**wdp**	**rst**	**rp**	**cpu**	**rp**
10	10	200	9975	1965	8.03	8015	14	0.40	0.597	11.12	5323	2	0.47	0.630	29.06	0.636
50	5	500	49925	5568	70.78	41289(1)	17	0.05	0.51	108.97	25287(1)	11	0.53	0.517	294.30(1)	0.521
50	10	200	49955	8983	71.02	39779(3)	6	0.40	0.49	122.66	26510(3)	3	0.49	0.492	416.88	0.496
100	10	200	99809	6592	264.12	90436	1	0.32	0.510	387.03	65289	1	0.45	0.519	1268.45	0.523

Figure 3.10: SAW data sets. Data from **n** and $\mathbf{w_{min}}$ columns should be multiplied by 10^3.

gives a clear advantage to `EDUK2` compared to `EDUK` and $\texttt{edu}_{U_3}$. This bound has an impact on the number of instances solved in the variable reduction step or by the initial *B&B* (column wdp), the number of reduced variable (column vrs), and the number of states (column rst).

Increasing ratio sets

In order to create difficult instances for DP, we generated items in such a way that the ratio $\frac{p}{w}$ is an increasing function of the weight. It is easy to see that $cd = n$ in this case. w values were uniformly and randomly generated within the interval $[w_{\min}..w_{\max}]$ (without duplicates) and were sorted in an increasing order. Then p was generated using

$$\begin{aligned} p_1 &= p_{\min} + k_1 \text{ and} \\ p_i &= \lfloor w_i \times (0.01 + \frac{p_{i-1}}{w_{i-1}}) \rfloor \quad + \quad k_i \text{ with } k_i \text{ randomly generated } \leq 10 \end{aligned} \tag{3.19}$$

We set $w_{\min} = p_{\min} = n$, $w_{\max} = 10\overline{n}$, and c was randomly generated within $[w_{\max}..1000\overline{n}]$. We did not observe any significant difference between `EDUK2` and $\texttt{edu}_{U_3}$, though both were about 4 times faster than `EDUK`.

500 instances per line				EDUK2					$\texttt{edu}_{U_3}$					EDUK	
n	**$\mathbf{w_{min}}$**	**nmd**	**ncd**	**cpu**	**vrs**	**wdp**	**rst**	**rp**	**cpu**	**vrs**	**wdp**	**rst**	**rp**	**cpu**	**rp**
5	**n**	**n**	**n**	7.93	3101	23	0.40	0.827	7.84	3101	23	0.40	0.827	29.05	0.816
10	**n**	**n**	**n**	36.84	5660(1)	13	0.43	0.745	36.73	5660(1)	13	0.43	0.745	147.76	0.759
20	**n**	**n**	**n**	184.55	12010	3	0.38	0.791	184.18	12010	3	0.38	0.791	735.24	0.783
50	**n**	**n**	**n**	808.26	25499	2	0.46	1	805.24	25499	2	0.46	1	2764.59	1

Figure 3.11: Increasing ratio data sets generated with formula (3.19). Data from **n** column should be multiplied by 10^3.

U_{τ^*} versus U_v for the SAW-UKP family

Theorem 13 states that U_{τ^*} is the best known upper-bound for the SAW-UKP family. In order to illustrate the quality improvement we have run `EDUK2` with U_{τ^*} and U_v respectively on a set of 14 000 SAW-UKP instances. The results are given in **Fig. 3.12**. The quality of a bound U is

computed by the gap: $Q_U = (U - opt)/opt$ where opt is the optimal value of the UKP. On these 14 000 instances, we observe that the average values are $Q_{U_{\tau^*}} = 0.016$, $Q_{U_v} = 0.023$.

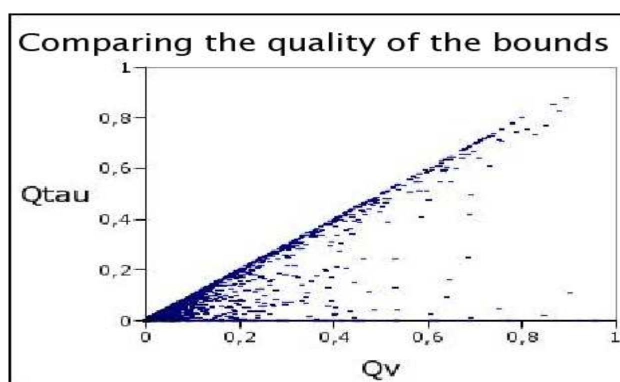
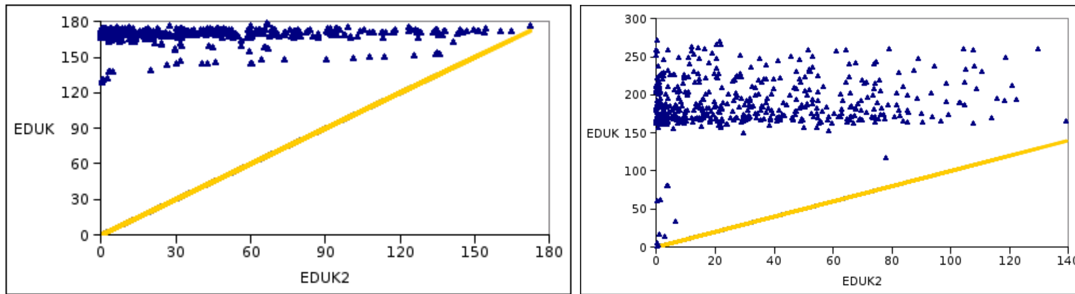


Figure 3.12: U_{τ^*} versus U_v on 14 000 instances of SAW-UKP. To any instance is associated a point with coordinates (Q_v, Q_τ) where Q_v (resp. Q_τ) is the corresponding gap $Q_U = (U - opt)/opt$. In all instances U_{τ^*} yields better bound.

Summary

`EDUK2` consistently and significantly outperformed EDUK on all data sets. On an average, `EDUK2` was between 1.7 and 6 times faster than EDUK ; for many instances, `EDUK2` yielded the solution immediately while EDUK required several minutes (sometimes more than 1 hour). **Fig. 3.13** illustrates the clear superiority of `EDUK2` to the one of EDUK on two large sets of instances. The efficiency of `EDUK2` is obtained by the cumulative effect of the different ways that *B&B* and DP are integrated. Taking into account all the new hard instances (except those generated with formula (3.18)), the reduction variables step reduces the number of items to be considered on an average varying from 55% to 95%. Integrating bounds during the DP phase further reduces the number of states from 46% to 95%. The impact of the new bound U_{τ^*} is important for all SAW-UKP instances and it affects all steps of the algorithm. For the non-SAW `UKP` instance no significant difference was observed between using U_{τ^*}, U_v or U_3.



Running times in seconds of EDUK2 (on the horizontal axis) and of EDUK (on the vertical axis). Each point corresponds to one instance. The line is the equal-time line. Left: data set generated by formula (3.19) with $n = 2 \times 10^4$. Right: SAW data set with $n = 5 \times 10^4$).

Figure 3.13: Plots of two set of instances

The superiority of EDUK2 to the general solver CPLEX is (as expected) apparent. In the dominant case, in all tests presented in section 3.2.5 EDUK2 was more than 100 times faster than CPLEX[7]. Additionally to these tests we found useful to check the performance of EDUK2 in some recent UKP applications. One such application is described in [53] where CPLEX has been used as UKP solver, instead of a special purpose algorithm. We generated the same set of instances as in [53] for $n = 10^6$. EDUK2 computed 5 such instances on an average time of 0.15 seconds, while the respective running time in [53] is announced to be around 30hrs!

There are still hard instances with large values for n and $w_{\min}$, notably those generated with formula (3.18), where $\alpha < 0, w_{\min} = 110000, n = 10000$. They were solved by EDUK2 on an average of 25 to 30 minutes. For all these difficult instances, the number of items that are not collectively dominated is very large. Thus, it appears that for such cases, DP algorithm needs to explore a huge iteration space when *B&B* fails to discover the solution.

3.2.6 Conclusion

We have shown that a hybrid approach combining several known techniques for solving UKP performs significantly better than any one of these techniques used separately. The effectiveness of the approach is demonstrated on a rich set of instances with very large inputs. The combined algorithm inherits the best timing characteristics of the parents (DP with bounds and *B&B*) and performs significantly better on almost all of the instances. We also proposed a new upper bound for the UKP and demonstrated that this bound is the tightest one known for a specific family of UKP. Our EDUK2 algorithm takes advantages of most of the known UKP properties and is able to solve all but the very special hard problems in a very short time. It appears that instances, previously known to be difficult, are now solvable in less than a few minutes.

[7] CPLEX execution time was upper bound by 600 sec.

Bibliography

[1] J. H. Ahrens and G. Finke. Merging and sorting applied to 0-1 knapsack problem. *Operations Research*, 23:1099–1109, 1975.

[2] R. Andonov, V. Poirriez, S. Rajopadhye, Unbounded knapsack problem : dynamic programming revisited, European Journal of Operational Research 123 (2) (2000) 168–181.

[3] D. Babayev, F. Glover, J. Ryan, A new knapsack solution approach by integer equivalent aggregation and consistency determination, INFORMS Journal on Computing 9 (1) (1997) 43–50.

[4] E. Balas. An additive algorithm for solving linear programs with zero-one variables. *Operations Research*, 13:517–546, 1965.

[5] E. Balas. Discrete programming by the filter method. *Operations Research*, 19:915–957, 1967.

[6] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, 1957.

[7] A.V. Cabot. An enumeration algorithm for knapsack problems. *Operations Research*, 18:306–311, 1970.

[8] L. Caccetta, A. Kulanoot, Computational Aspects of Hard Knapsack Problems, Nonlinear Analysis 47 (2001) 5547–5558.

[9] A. K. Chandra, D. S. Hirshberg, and C. K. Wong. Approximate algorithms for some generalized knapsack problems. *Theoretical Computer Science*, 3:293–304, 1976.

[10] H. Crowder, E. L. Johnson, and H. W. Padberg. Solving large scale zero-one linear programming problems. *Operations Research Society*, 31:803–834, 1983.

[11] C.-S. Chung, M. S. Hung, W. O. Rom, A Hard Knapsack Problem, Naval Research Logistics 35 (1988) 85–98.

[12] P. Chu and J. Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics*, 4:63–86, 1998.

[13] D. Fayard and G. Plateau. Reduction algorithm for single and multiple constraints 0-1 linear programming problems. In *Proceedings of Congress Methods of Mathematical Programming*, Zakopane, 1977.

[14] A. Fréville and G. Plateau. Heuristics and reduction methods for multiple constraints 0-1 linear programming problems. *European Journal of Operational Research*, 24:206–215, 1986.

[15] A. Fréville and G. Plateau. An efficient preprocessing procedure for the multidimensional knapsack problem. *Discrete Applied Mathematics*, 49:189–212, 1994.

[16] A. Fréville and G. Plateau. The 0-1 bidimensional knapsack problem: Towards an efficient high-level primitive tool. *Journal of Heuristics*, 2:147–167, 1996.

[17] A. M. Frieze and M. R. Clarke. Approximation algorithms for the m-dimensional 0-1 knapsack problem: Worst-case and probablistic analyses. *European Journal of Operational Research*, 15:100–109, 1984.

[18] R. Garfinkel, G. Nemhauser, Integer Programming, John Wiley and Sons, 1972.

[19] B. Gavish and H. Pirkul. Efficient algorithms for solving zero-one multidimensional knapsack problems to optimality. *Mathematical Programming*, 31:78–105, 1985.

[20] G. V. Gens and E. V. Levner. Computational complexity of approximation algorithms for combinatorial problems. In *Mathematical Foundations of Computer Science*, volume 74 of *Lecture Notes in Computer Science*, pages 292–300, 1979.

[21] A. Geoffrion. An improved implicit enumeration approach for integer programming. *Operations Research*, 17:437–454, 1969.

[22] P. C. Gilmore and R. E. Gomory. The theory and computation of knapsack functions. *Operations Research*, 14:1045–1074, 1966.

[23] F. Glover. A multiphase dual algorithm for the zero-one integer programming problem. *Operations Research*, 13:879–919, 1965.

[24] F. Glover and G. Kochenberger. Critical event tabu search for multidimensional knapsack problems. In I. Osman and J. Kelly, editors, *Metaheuristics: Theory and Applications*, pages 407–427. Kluwer Academic Publishers, 1996.

[25] F. Glover, H. Sherali, and Y. Lee. Generating cuts from surrogate constraint analysis for zero-one and multiple choice programming. *Computational Optimization and Applications*, 8:151–184, 1997.

[26] M. Guignard and K. Spielberg. Logical reduction methods in zero-one programming minimal preferred variables. *Operations Research*, 29:49–74, 1981.

[27] P. Hammer, M. Padberg, and U. Peled. Constraint pairing in integer programming. *INFOR*, 13:68–81, 1975.

[28] S. Hanafi and A. Fréville. An efficient tabu search approach for 0-1 multidimensional knapsack problem. *European Journal of Operational Research*, 106:659–675, 1998.

[29] E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. *Journal of ACM*, 21:277–292, 1974.

[30] T. Ibaraki. Enumerative approaches to combinatorial optimization-part ii. *Annals of Operations Research*, 11:?, 1987.

[31] B. Korte and R Schrader. On the existence of fast approximation schemes. In O. L. Mangasarian, R. R. Meyer, and S. M. Robinson, editors, *Nonlinear Programming 4*, pages 415–437. Academic Press, 1980.

[32] A. S. Manne and H. M. Markowitz. On the solution of discrete programming problems. *Econometrica*, 25:84–100, 1957.

[33] R.E. Marsten and T.L. Morin. A hybrid approach to discrete mathematical programming. *Mathematical Programming*, 14:21–40, 1978.

[34] S. Martello and Toth P. *Knapsack Problems: Algorithms and Computer Implementations*. Series in Discrete Mathematics and Optimization. Wiley Interscience, New York, 1990.

[35] S. Martello, D. Pisinger, and P. Toth. New trends in exact algorithms for the 0-1 knapsack problem. *European Journal of Operational Research*, 123:325–336, 1999.

[36] G. L. Nemhauser, M. W. P. Savelsbergh, and G. C. Sigismondi. MINTO, a Mixed INTeger Optimizer. *Operations Research Letters*, 15:47–58, 1994.

[37] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. Wiley-Interscience series in discrete mathematics and optimization. Wiley, 1988.

[38] M.A. Osorio, F. Glover, and P. Hammer. Cutting and surrogate constraint analysis for improved multidimensional knapsack solutions. Research report, University of Puebla, Mexico, 2000. Presented at IFORS, Mexico, September 2000.

[39] U. Pferschy, H. Kellerer, D. Pisinger, Knapsack Problems, Springer Verlag, 2004, iSBN 3-540-40286-1.

[40] D. Pisinger. A minimal algorithm for the 0-1 knapsack problem. *Operations Research*, 45:758–767, 1997.

[41] G. Plateau and M. Elkihel. A hybrid method for the 0-1 knapsack problem. *Methods of Operations Research*, 49:277–293, 1985.

[42] V. Poirriez, R. Andonov, Unbounded Knapsack Problem: New Results, in: Proceedings of the Workshop Algorithms and Experiments (ALEX98), 1998, pp. 103–111, available at: `http://rtm.science.unitn.it/alex98/proceedings.html`.

[43] V. Poirriez, N. Yanev, R. Andonov, Towards reduction of the class of intractable unbounded knapsak problem, Research Report 1, LAMIH/ROI UMR CNRS 8530,(july 2002).

[44] M. W. P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal of Computing*, 6:445–454, 1994.

[45] S. Senju and Y. Toyoda. An approach to linear programming with 0-1 variables. *Management Science*, 15:196–207, 1968.

[46] W. Shih. A branch and bound method for the multiconstraint knapsack problem. *Journal of the Operational Research Society*, 30:369–378, 1979.

[47] A.L. Soyster, B. Lev, and W. Slivka. Zero-one programming with many variables and few constraints. *European Journal of Operational Research*, 2:195–201, 1978.

[48] P. Toth. Dynamic programming algorithms for the zero-one knapsack problem. *Computing*, 25:29–45, 1980.

[49] M. Vasquez and J. K. Hao. A hybrid approach for the 0-1 multidimensional knapsack problem. In *Proceedings of IJCAI-01, Seventeenth International Joint Conference on Artificial Intelligence*, pages 328–333, 2001.

[50] F. Viader. *Méthodes de Programmation Dynamique et de Recherche Arborescente pour l'Optimisation Combinatoire: Utilisation Conjointe des deux approches et Parallélisation d'Algorithmes*. PhD thesis, Univérsité Paul Sabatier, Toulouse, 1998.

[51] H. M. Weingartner and D. N. Ness. Method for the solution for the multidimensional 0-1 knapsack problem. *Operations Research*, 15:83–103, 1967.

[52] N. Zhu, K. Broughan, On dominated terms in the general knapsack problem, Operations Research Letters 21 (1997) 31–37.

[53] C. Srisuwannapa, P. Charnsethikul, An Exact Algorithm for the Unbounded Knapsack Problem with Minimizing Maximum Processing Time, Journal of Computer Science, 3 (3) : 138-143, 2007.

[54] V. Boyer, M. Elkihel, D. El Baz , Efficient heuristics for the 0/1 multidimensional knapsack, ROADEF 2006, 95-106.

[55] S. Martello, D. Pisinger, P. Toth, Dynamic programming and strong bounds for the 0-1 knapsack problem , Manag. Sci., 45:414-424, 1999.

[56] S. Martello, P. Toth, A mixture of dynamic programming and branch-and-bound for the subset-sum problem, Manag. Sci., 30:765-771.

[57] G. Plateau, M. Elkihel, A hybrid algorithm for the 0-1 knapsack problem, Methods of Oper. Res., 49:277-293, 1985.

Chapter 4

The classification problem

Discriminant analysis involves studying the difference between two or more mutually exclusive groups. In the classification problem of discriminant analysis, the objective is to use measured values on a set of relevant variables or attributes in order to predict the group membership of a new observation. Discriminant analysis has been applied extensively in wide variety of areas, such as finance, marketing, artificial intelligence, medical sciences, biology, and social sciences. Fisher's linear discriminant function and quadratic discriminant function have long been the standard techniques for establishing discriminant rules in classification analysis. Both of these functions, however, are based on the assumption of multivariate normality of the measured variables. In many situations involving real data, these assumptions are seriously violated, for instance, in the case of binary variables and when outliers are present in the data set. Recent studies have indicated that outlier-contaminated data set are the norm rather than the exception in a number of business-related fields with up to 10% of outlier observations. That is why, a number of researchers have introduced and investigated mathematical programming- (MP) based formulations to solve the classification problem, resulting in a number of useful nonparametric techniques which have been shown to perform well under various data conditions. The most common MP approaches suggested in the literature are the MSD (minimize the sum of the deviations) and the MMD (minimize the maximum deviation). Mixed-integer programming (MIP) formulations have been suggested which directly minimize the number of misclassified observations, in the training sample. Of course, the MIP formulations can require extensive computational resources which may be prohibitive for large data sets. What is proposed below is a step towards overcoming this obstacle. More precisely, we study the two-group classification problem which involves classifying an observation into one of two groups based on its attributes. The classification rule is a hyperplane which misclassifies the fewest number of observations in the training sample. Exact and heuristic algorithms for solving the problem are presented. Computational results confirm the efficiency of this approach.

Introduction

The classification problem, also called a discriminant problem, involves classifying an observation into one of a number of mutually exclusive groups, according to its attributes. This is one of the most fundamental problems of scientific inquiry with applications in biology, artificial intelligence, social and administrative sciences.

Observations are characterized by their attributes, so they can be represented by vectors, each component of which is a value of some attribute. In the two-group linear discriminant analysis a hyperplane with a normal vector w and a bias w_0 is determined, so that an observation X is assigned to the first group if $Xw \leq w_0$ and to the second group otherwise. This hyperplane, also called a linear discriminant function, is determined from a training sample – a set of observations, whose group membership is a priori known, so that it best separates the two groups according to some criterion. Different approaches to the classification problem are developed depending on this criterion.

In this paper we present an efficient algorithm for constructing liner discriminant function which minimizes the number of misclassified observations from the training sample. This problem is known to be intractable, which necessitates the developing of efficient special purpose exact or heuristic algorithms.

We use the following notations:

m – the number of attributes;
n – the number of observations (points) in the training sample;
$X_i = (x_{i1}, \ldots, x_{im})$ – the i-th observation, $i \in \overline{1, n}$;
G_1 – the index set for the points of the first group;
G_2 – the index set for the points of the second group.

The rest of the paper is organized as follows. In Section 1 we briefly discuss and compare the most popular approaches to the classification problem. In Section 2 we transform the problem to more convenient equivalent form by duality considerations. Using the new formulation we develop an exact branch-and-bound algorithm for solving the problem in Section 3. Section 4 presents a family of heuristics and a framework of local search algorithms for approximate solutions. Section 5 summarizes the results of computational experiment and analyzes the efficiency of the proposed algorithms. In Section 6 we conclude.

4.1 Overview of approaches to the classification problem

There are two popular ways to tackle the classification problem – by statistical techniques and by mathematical programming (MP) approaches.

Statistical approaches are the classical tool in discriminant analysis. In general, they operate with the assumption that each population has known distribution. Classification rules, produced by statistical approaches, minimize the probability of misclassification. When the population distributions are multivariate normal with mean vectors μ_i, a common variance-covariance matrix Σ, and relative frequencies π_i, $i = 1, 2$, the classification rule states:

Assign an observation X to G_1 if

$$X^T\Sigma^{-1}(\mu_1 - \mu_2) > \frac{1}{2}(\mu_1 - \mu_2)^T\Sigma^{-1}(\mu_1 + \mu_2) + \ln\frac{\pi_2}{\pi_1}$$

and to G_2 otherwise.

This is the most widely known Fisher's linear discriminant function [5]. When μ_1, μ_2, and Σ are replaced by their maximum likelihood estimators, a sample-based classification rule is obtained.

MP approaches are developed as an alternative to the statistical methods. As an example of linear programming (LP) approaches we shall demonstrate the MMD (minimize maximum distance) model [6]:

$$\text{Maximize } d$$

subject to:

$$\sum_{j=1}^{m} x_{ij}w_j + d \leq w_0 \text{ for } i \in G_1$$

$$\sum_{j=1}^{m} x_{ij}w_j - d \geq w_0 \text{ for } i \in G_2$$

$$w_j, d \text{ – unrestricted}$$

This model determines a hyperplane that maximizes the minimum distance, d, between an observation and the hyperplane.

After their introduction by Freed and Glover [6, 7], various LP formulations have been proposed [8, 9, 14] and shown to outperform statistical approaches, when the assumptions for group distributions are violated [3, 11, 13]. As well as their advantages, LP formulations have several defects [14]. One of them is that they may produce the trivial solution $w_j = 0$ for all j, i. e. meaningless classification rule. To overcome this defect, a normalization constraint $bw = c$ is added to the model, where b is a nonzero vector and c is a nonzero constant. The addition of this constraint prevents the zero solution, but causes some side effects as eliminating possible solutions, for which $bw = 0$. Sensitivity to the extreme values is also a problem of LP formulations. If an observation from one group is far within the region of the other group, LP models give unsatisfactory results. This fault can be overcome by using mixed integer programming (MIP) formulations. For example:

$$\text{Minimize } \sum_{i=1}^{n} z_i$$

subject to:

$$\sum_{j=1}^{m} x_{ij}w_j - Mz_i \leq w_0 \text{ for } i \in G_1$$

$$\sum_{j=1}^{m} x_{ij}w_j + Mz_i \geq w_0 \text{ for } i \in G_2$$

$$w_j \text{ – unrestricted; } z_i \in \{0, 1\}$$

where M is sufficiently large number. The binary variables z_i are 0 if the i-th observation is correctly classified and 1 otherwise. MIP formulations directly minimize the number of misclassified observations. Computational time for solving these formulations by general purpose methods is significant because of the large number of binary variables and also because LP relaxations of MIP models produce bad lower bounds for large M. Bajgier and Hill [3] stated that "MIP approaches is unlikely to be practical for discriminant problems with more than 50 cases unless very efficient, special-purpose algorithms (either exact or heuristic) are developed". Such algorithms are proposed in [1, 4, 12]. In the next sections we also propose algorithms that minimize the number of misclassifications.

4.2 Problem formulation

Consider n points $X_i = (x_{i1}, \ldots, x_{im})$, $i \in \overline{1, n}$ from two groups – G_1 and G_2. The group membership of each point is known. Our objective is to remove minimal number of points, so that the rest of them become linearly separable, i. e. the system

$$\sum_{j=1}^{m} x_{ij} w_j \leq w_0 \text{ for } i \in G_1$$

$$\sum_{j=1}^{m} x_{ij} w_j \geq w_0 \text{ for } i \in G_2$$

has a nonzero solution after removing these points. It is proved in [2] that the above problem is NP-hard and moreover, very hard to approximate – not approximable within $2^{\log^{1-\epsilon} n}$ for any $\epsilon > 0$, unless $NP \subseteq QP$.

Let A be a matrix whose columns are

$$A_i = \begin{cases} (-1, X_i)^T & \text{for i } \in G_1 \\ (1, -X_i)^T & \text{for } i \in G_2 \end{cases}, \; i \in \overline{1, n}$$

and let $w = (w_0, w_1 \ldots, w_m)^T$. We prevent zero solution by normalization $bw > 0$ for an arbitrary nonzero vector b. So we obtain the following problem:

(P) *F*ind a set $I \subseteq \overline{1, n}$ of minimal cardinality, such that the system

$$\begin{array}{rcl} A^T w & \leq & 0 \\ bw & > & 0 \end{array} \tag{4.1}$$

becomes feasible after removing constraints with numbers in I.

Consider the system

$$\begin{aligned} Ax &= b \\ x &\geq 0 \end{aligned} \tag{4.2}$$

According to the Farkas theorem, (1) is feasible if and only if (2) is infeasible. Let $I \subseteq \overline{1,n}$ be an index set. Denote by (2_I) the system (2) with removed (zeroed) variables x_i for $i \in I$. Then the initial problem **P** is equivalent to (has the same solution as) the following problem:
(R) Find a set $I \subseteq \overline{1,n}$ of minimal cardinality, so that the system (2_I) is infeasible.

Further on we consider the problem **R** instead of the original problem **P**. Before proceeding with solving this problem, let us make several remarks.

1. Note that for practical problems the number of points n is much grater than the dimension m of the observation space. Working with the new system with only $m+1$ constraints instead of the initial system with n constraints is much easier.

2. In our formulation points on the discriminant hyperplane are considered as properly classified. Such separation is called unstrict. When strict separation is needed, we consider the system
$$\sum_{j=1}^{m} x_{ij} w_j \leq w_0^1 \text{ for } i \in G_1$$
$$\sum_{j=1}^{m} x_{ij} w_j \geq w_0^2 \text{ for } i \in G_2$$
with a normalization constraint
$$w_0^2 - w_0^1 > 0$$
and construct a problem, similar to **R** in the same way.

3. From any optimal solution I^* of **R**, the separating hyperplane of the points, whose indices are out of I^*, could be easily found. It is possible to solve some of the linear models with the remaining points, already linearly separable, to obtain some secondary goal.

4. Finally, let us specify some notations. As usual, a basis of (2) is an index set $B \subseteq \overline{1,n}$ with $m+1$ elements, such that the corresponding matrix B, consisting of columns A_j for $j \in B$ is non-singular. We shall use basic representations of (2) of the form
$$\sum_{i=1}^{n} y_i x_i = \overline{b} \tag{3}$$
where $y_i = B^{-1}A_i$ and $\overline{b} = B^{-1}b$. When $\overline{b} \geq 0$, the relevant basis is called feasible. We shall denote the j-th element of B by $B(j)$. The solution of (2) $x_{B(j)} = \overline{b}_j$ for $j \in \overline{0,m}$ and $x_i = 0$ for $i \notin B$ corresponds to the feasible basis B. Basic representations of (2) are a key feature of our algorithms.

4.3 Exact algorithm

In this section we propose a branch-and-bound algorithm for solving the problem **R** to optimality. The main features of each branch-and-bound algorithm are the branching strategy and the lower bounds generated at each node. They are described in subsections 1 and 2 respectively. Other problem specific details and an overall description of the algorithm are given in subsection 3.

4.3.1 Branching procedure

The branching strategy is based on the following theorem.

Theorem 14. *Let I^* be an optimal solution of* **R**. *Then:*
(i) for each $k \in I^*$, there exists a feasible basis B of (2), such that $k \in B$;
(ii) for each feasible basis B of (2), there exists $k \in I^*$, such that $k \in B$.

Proof. (i) Let $k \in I^*$ and $I = I^* \setminus \{k\}$. The system (2_I) is feasible, because I^* is an optimal solution of **R**. Let B be a feasible basis of (2_I). Then (2_I) has the following basic representation

$$\sum_{i\in B} y_i x_i + \sum_{i\in\overline{1,n}\setminus(I\cup B)} y_i x_i = \bar{b}.$$

Suppose that $k \notin B$, then

$$\sum_{i\in B} y_i x_i + \sum_{i\in\overline{1,n}\setminus(I\cup B\cup\{k\})} y_i x_i = \bar{b}$$

is a basic representation of (2_{I^*}), i. e. (2_{I^*}) is feasible, which contradicts to the optimality of I^*.

(ii) Let B be a feasible basis of (2). Suppose that for each $k \in I^*$ $k \notin B$. Then (2) has a basic representation

$$\sum_{i\in B} y_i x_i + \sum_{i\in I^*} y_i x_i + \sum_{i\in\overline{1,n}\setminus(I^*\cup B)} y_i x_i = \bar{b}.$$

Hence, (2_{I^*}) has a basic representation

$$\sum_{i\in B} y_i x_i + \sum_{i\in\overline{1,n}\setminus(I^*\cup B)} y_i x_i = \bar{b},$$

i. e. (2_{I^*}) is feasible. This contradiction proves (ii). □

Theorem 1 shows that each feasible basis of the system (2) contains a variable which is removed in the optimal solution of our problem. So at each node the candidates for removing may be only the $m+1$ basic variables instead of all n variables. Formally, let I be an index set. By $\mathbf{R}_I$ we denote the problem **R**, where the system (2) is replaced by (2_I). The branching tree is defined in the following way:

1. **R** is the root of the tree.

2. Suppose R_I is in a certain node of the tree. Then:

   (a) if the system (2_I) is infeasible, this node is a leaf of the tree;

   (b) if the system (2_I) is feasible and B is its feasible basis, then the successors of this node are the problems $\mathrm{R}_{I\cup\{B(j)\}}$ for $j \in \overline{0,m}$.

It follows from Theorem 1 that if I^* is an optimal solution of **R** and $|I^*| = s$, then R_{I^*} is a leaf on the s-th level of the tree. So, the branching at any node will be performed on the basic variables of the corresponding problem.

Note that the same problem may appear several times in the tree. For example, let k and l be in a basis of (2), k be in a basis of $(2_{\{l\}})$, and l be in a basis of $(2_{\{k\}})$. Then the problem $\mathrm{R}_{\{k,l\}}$ will appear twice at the second level of the tree. To avoid this, we proceed as follows. Let B be the feasible basis of problem R_I, on which we perform branching. Then branching on $B(0), \ldots, B(j-1)$ is not performed in the whole subtree with root $\mathrm{R}_{I\cup\{B(j)\}}$, even if they are basic in some node of this subtree. It is easy to see that in this way we avoid repeating problems without loosing possible solutions. When we pass from a basis of a certain node to a basis of its successor, it is useful to leave as many of the old basic variables as possible in the new basis. This will increase the number of repeating problems and hence, will reduce the nodes of the tree.

4.3.2 Lower bounds

To obtain lower bounds of the solutions of the problems R_I, we use Theorem 2, which immediately follows from Theorem 1.

Theorem 15. *Let $B_1, \ldots, B_l$ be mutually exclusive feasible bases of the system* (2) *and I^* be an optimal solution of the problem* **R**. *Then $|I^*| \geq l$.*

We find a sequence of mutually exclusive feasible bases of (2) using the following "greedy" strategy:

1. $l = 0$.

2. Find a feasible basis of (2). If there is not such a basis, i. e. (2) is infeasible, stop. Else $l = l + 1$, B_l =this feasible basis.

3. Remove all basic variables from (2). Goto 2.

To find mutually exclusive bases of a certain problem, bases of its predcessor, already found, can be used in the following way. Let $B_1, \ldots, B_l$ be mutually exclusive feasible bases for the problem R_I and let us try to find such a sequence for the problem $\mathrm{R}_{I\cup\{i\}}$. If $i \notin B_1 \cup \ldots \cup B_l$ then the new problem has the same sequence of mutually exclusive bases. In the case when i belongs to some of these bases, without loss of generality, consider that $i \in B_1$.

```
dsf(problem R);
{
  /*try to fathom node*/
  if (R.lb>=record) return;
  /*branch on the basic vars*/
  for(j=0;j<=m;j++)
  {
    /*branch only on free vars*/
    if (R.status[R.basis[j]]!=NONE) continue;
    /*generate new problem*/
    newR=R;
    newR.status[R.basis[j]]=OUT;
    remove basis[j] from newR by pivoting on row j;
    perform dual simplex method over newR;
    if (newR is infeasible)
    {
      /*better solution is found*/
      modify record and save the new incumbent;
      return;
    }
    compute newR.lb using the information for bases of R;
    dsf(newR);
    R.status[R.basis[j]]=BANNED;
  }
}
```

Figure 4.1: Branch-and-bound algorithm

Then $B_2, \dots, B_l$ remain feasible bases for the new problem. We try to find more bases by applying the above "greedy" technique to the system $(2_{B_2 \cup \dots \cup B_l \cup \{i\}})$. In this way generating the lower bounds at each node of the tree is not an expensive procedure if the data are organized in appropriate way.

Finally, note that if $B_1, \dots, B_l$ are mutually exclusive bases for the problem $\mathbf{R}_I$ then a lower bound for this problem is $|I| + l$, which together with the previous considerations ensures the nondecreasing property of the bounds along any path from the root.

4.3.3 Algorithm Description

We perform a depth-first branch-and-bound search using the described branching procedure. Lower bounds generated at each node are used to fathom in the usual way. A brief description of the algorithm is given on Figure 1. It is implemented by recursive function `dsf` (Depth-First Search) which gets an argument of type `problem`. This is a structure containing information

for the problem at the current node and some of its fields are:

`lb` – the lower bound for the current problem;
`basis` – an array containing the indices of basic variables for the basic representation of the current problem;
`status` – an array containing information for the status of all variables. The status of each variable may be: `NONE` – meaning that the variable is free to branch on it; `OUT` – meaning that this variable has been removed at lower level of the tree and `BANNED` – meaning that branching on this variable has already been performed;
`y` and `b` – arrays containing the basic representation of the current problem.

The `problem` structure also contains an information about the mutually exclusive bases used to obtain the lower bound.

For the efficient performance of the branch-and-bound algorithm it is important to have a good initial incumbent and value of `record` respectively. In the next section we propose heuristic algorithms used to obtain the initial solution. They may also be used as an independent tool for approximate solving of the discriminant problem.

4.4 Upper bounds and heuristic algorithms

We propose methods that produce feasible solutions of the problem **R** close to the optimal solution. These methods are based on the following result.

Theorem 16. *Let* (3) *be an arbitrary basic representation of the system* (2) *and* $\boldsymbol{\lambda}$ *be a vector, such that* $\langle \boldsymbol{\lambda}, \overline{\boldsymbol{b}} \rangle > 0$. *Then the set* $\boldsymbol{T} = \{i \in \overline{1,n}, \langle \boldsymbol{\lambda}, \boldsymbol{y_i} \rangle > 0\}$ *is a feasible solution of* **R**.

Proof. Multiplying (3) by $\boldsymbol{\lambda}$ we obtain

$$\sum_{i=1}^{n} \langle \boldsymbol{\lambda}, \boldsymbol{y_i} \rangle \boldsymbol{x_i} = \langle \boldsymbol{\lambda}, \overline{\boldsymbol{b}} \rangle.$$

After removing the variables with indices in $\boldsymbol{T}$, the last equality becomes:

$$\sum_{\substack{i \in \overline{1,n} \\ \langle \lambda, y_i \rangle \leq 0}} \langle \boldsymbol{\lambda}, \boldsymbol{y_i} \rangle \boldsymbol{x_i} = \langle \boldsymbol{\lambda}, \overline{\boldsymbol{b}} \rangle.$$

The left-hand side of the last equality is nonpositive for any values of the variables $\boldsymbol{x_i} \geq \boldsymbol{0}$ while the right-hand side is positive by assumption. Hence, the system $(\boldsymbol{2_T})$ is infeasible, i. e. $\boldsymbol{T}$ is a feasible solution of **R**. □

If we set all but $\boldsymbol{j}$-th component of $\boldsymbol{\lambda}$ equal to zero, we obtain that the sets

$$\boldsymbol{T_j} = \begin{cases} \{i \in \overline{1,n}, y_{ji} > 0\} & \text{if } \overline{b}_j > 0 \\ \{i \in \overline{1,n}, y_{ji} < 0\} & \text{if } \overline{b}_j < 0 \end{cases}$$

```
U1(problem R)
{
  U={1,...,n};
  for(j=0;j<=m;j++)
  {
    if (R.b[j]==0) continue;
    T={};
    if (R.b[j]>0)
      for(i=1;i<=n;i++) {if (R.y[j][i]>0) T=T+{i};}
    else
      for(i=1;i<=n;i++) if (R.y[j][i]<0) T=T+{i};
   if (|T|<|U|) U=T;
  }
  return U;
}
```

Figure 4.2: Heuristic U_1

are feasible solutions of **R**. Taking the minimal of these sets we obtain the simplest heuristic, U_1:

U_1 is the set of minimal cardinality among the sets T_j for $j \in \overline{0, m}$ and $\bar{b}_j \neq 0$.

The algorithm for finding the set U_1 is shown at Figure 2. Geometrically, the j-th row of B^{-1} determines a hyperplane which misclassifies all points i with $y_{ji} > 0$ (or by reversing the signs of the normal vector those with $y_{ji} < 0$). The heuristic U_1 suggests to take the hyperplane with the smallest misclassification rate.

Obviously, we can try to improve by looking for a linear combination of hyperplanes which is a kind of a local search procedure. As usual its complexity and quality increases with the volume of the neighborhood area. The formal description of the idea is given below.

Consider k rows from (3) with numbers $j \in J$, $|J| = k$:

$$\sum_{j=1}^{n} c_i x_i = c_0$$

where c_i are the vectors with components y_{ji}, $j \in J$, and c_0 is the vector with components $\bar{b}_j$, $j \in J$. We choose J so that $c_0 \neq 0$. Note that there are at least k linearly independent among the vectors c_i, $i \in I$ (for example, such are the unit vectors e_j, $j \in \overline{1, k}$). Let

$$T(\lambda) = \{i \in \overline{1, n}, \langle \lambda, c_i \rangle > 0\}.$$

By Theorem 3, if $\langle \lambda, c_0 \rangle > 0$ then $T(\lambda)$ is a feasible solution of **R**. Consider the following problem:

[(S_k)] Find a vector λ, such that $\langle \lambda, c_0 \rangle > 0$ and the set $T(\lambda)$ is minimal.

Obviously, any optimal solution of S_k is an upper bound for the optimal solution of **R** and as it will be shown later, the sharpness of the bounds increases with k. To solve this problem, let

$$D(\lambda) = \{\mu : \langle \mu, c_0 \rangle > 0; \langle \lambda, c_i \rangle \langle \mu, c_i \rangle \geq 0, i \in \overline{1, n}\}.$$

Lemma 3. If λ is a solution of S_k then each vector from $D(\lambda)$ is also a solution of S_k.

Proof. Let λ be a solution of S_k and $\mu \in D(\lambda)$. If $\langle \lambda, c_i \rangle \leq 0$ then $\langle \mu, c_i \rangle \leq 0$, i. e. if $i \notin T(\lambda)$ then $i \notin T(\mu)$. Then $T(\mu) \subseteq T(\lambda)$. But $\langle \mu, c_0 \rangle > 0$, hence μ is a solution of S_k. □

For the sets $I \subseteq \overline{1, n}$, $|I| = k - 1$, for which the vectors c_i, $i \in I$ are linearly independent, we define rays d_I as follows:

$$\langle d_I, c_i \rangle = 0, \; i \in I \tag{4}$$

$$\langle d_I, c_0 \rangle > 0 \tag{5}$$

It is clear that for each set I there is at most one ray satisfying conditions (4) and (5).

Lemma 4. For each vector λ, such that $\langle \lambda, c_0 \rangle > 0$, there exists a set I such that $d_I \in D(\lambda)$.

Proof. Consider the cone

$$D'(\lambda) = \{\mu : \langle \lambda, c_i \rangle \langle \mu, c_i \rangle \geq 0, i \in \overline{1, n}\}.$$

Let $d_1, \ldots, d_s$ be all of the extreme rays of $D'(\lambda)$. Since $\lambda \in D'(\lambda)$, we have

$$\lambda = \sum_{i=1}^{s} \alpha_i d_i,$$

where $\alpha_i \geq 0$, $i \in \overline{1, s}$. Hence,

$$\langle \lambda, c_0 \rangle = \sum_{i=1}^{s} \alpha_i \langle c_0, d_i \rangle > 0.$$

Then $\langle c_0, d_i \rangle > 0$ for at least one $i \in \overline{1, s}$. Let d be one of the rays $d_1, \ldots, d_s$, such that $\langle c_0, d \rangle > 0$. Then $d \in D(\lambda)$. d is a section of $k - 1$ hyperplanes

$$\{\mu : \langle \lambda, c_i \rangle \langle \mu, c_i \rangle = 0\}.$$

Let I be the set of the numbers of these hyperplanes. Since d satisfies (4) and (5), $d \equiv d_I$. □

As an immediate corollary from Lemma 1 and Lemma 2 appears the following result.

Theorem 17. *Some of the rays d_I, defined by* (4) *and* (5) *is an optimal solution of the problem* S_k.

The rays d_I are easily found by (4) and (5). By checking all of them we can find a solution of S_k.

Thus we obtained a method for finding upper bounds U_k, $k \in \overline{1, m+1}$ by considering k rows of (3) and solving the problems S_k. It is easy to see that $|U_1| \geq \cdots \geq |U_{m+1}|$, but the computational effort for finding U_k is considerable for big k. These heuristics are useful only for small k. We shall prove that as it is to be expected, U_{m+1} produces the optimal solution of **R**. For that purpose, note that the problem S_{m+1} can be defined as follows: Find the minimal among the sets $T(\lambda) = \{i \in \overline{1, n}, \langle \lambda, y_i \rangle > 0\}$ where $\langle \lambda, \bar{b} \rangle > 0$. and the initial problem **P** can be written down in the following way: Find the minimal among the sets $I(w) = \{i \in \overline{1, n}, \langle w, A_i \rangle > 0\}$ where $\langle w, b \rangle > 0$.

Lemma 5. The problems S_{m+1} and **P** are equivalent (have the same solutions).

Proof. Let $T(\lambda^0)$ be an optimal solution of S_{m+1} and $w^0 = (B^{-1})^T \lambda^0$. Then:

$$\langle w^0, b \rangle = \langle (B^{-1})^T \lambda^0, b \rangle = \langle \lambda^0, B^{-1} b \rangle = \langle \lambda^0, \bar{b} \rangle > 0,$$

$$\langle w^0, A_i \rangle = \langle (B^{-1})^T \lambda^0, A_i \rangle = \langle \lambda^0, B^{-1} A_i \rangle = \langle \lambda^0, y_i \rangle, \ i \in \overline{1, n}.$$

Hence, $I(w^0)$ is a feasible solution of **P** and $I(w^0) \equiv T(\lambda^0)$. Suppose that $I(w^0)$ is not an optimal solution of **P**. Then there exists w^1, such that $\langle w^1, b \rangle > 0$ and $|I(w^1)| < |I(w^0)|$. Let $\lambda^1 = B^T w^1$. Then:

$$\langle \lambda^1, \bar{b} \rangle = \langle B^T w^1, B^{-1} b \rangle = \langle w^1, B B^{-1} b \rangle = \langle w^1, b \rangle > 0,$$

$$\langle \lambda^1, y_i \rangle = \langle B^T w^1, B^{-1} A_i \rangle = \langle w^1, B B^{-1} A_i \rangle = \langle w^1, A_i \rangle > 0, \ i \in \overline{1, n}.$$

Hence, $T(\lambda^1)$ is a feasible solution of S_{m+1} and $T(\lambda^1) \equiv I(w^1)$. But $|T(\lambda^1)| = |I(w^1)| < |I(w^0)| = |T(\lambda^0)|$, which contradicts the optimality of $T(\lambda^0)$. Therefore, $I(w^0) \equiv T(\lambda^0)$ is an optimal solution of **P**.

It can be proved that an optimal solution of **P** is also an optimal solution of S_{m+1} in the same way. □

From the last lemma and the equivalence between the problems **P** and **R**, it follows that:

Theorem 18. *U_{m+1} is an optimal solution of the problem* **R**.

The heuristics U_k depend on the basis where they are computed, i. e. $U_k = U_k(B)$. The next theorem shows the strong influence of the basis upon these heuristics.

Theorem 19. *There exists a basis B of the system* (2)*, such that $U_1(B)$ is an optimal solution of the problem* **R**.

Proof. Let (3) be an arbitrary basic representation of (2) and λ be a solution of the problem S_{m+1}. Due to Theorem 5, $T(\lambda) \equiv U_{m+1}$ is an optimal solution of **R**. According to Theorem 4, there exists a set I, such that $|I| = m$; the vectors y_i, $i \in I$, are linearly independent; $\langle \lambda, y_i \rangle = 0$, $i \in I$; and $\langle \lambda, \bar{b} \rangle > 0$. We construct a matrix M with the following columns:

$M_1, \dots, M_m$ are the vectors y_i, $i \in I$, and $M_0 = y_s$, where we choose s so that the matrix M is non-singular. By multiplying (3) from left by M^{-1}, we obtain a new basic representation

$$\sum_{i=1}^{n} y_i' x_i = \overline{b}'$$

where $y_i' = M^{-1} y_i$, $\overline{b}' = M^{-1}\overline{b}$. The basis corresponding to this representation is $B = \{s\} \cup I$. We shall prove that B is the required basis. Let $\lambda' = M^T \lambda$. Then:

$$\langle \lambda', \overline{b}' \rangle = \langle M^T\lambda, M^{-1}\overline{b} \rangle = \langle \lambda, MM^{-1}\overline{b} \rangle = \langle \lambda, \overline{b} \rangle > 0,$$

$$\langle \lambda', y_i' \rangle = \langle M^T\lambda, M^{-1}y_i \rangle = \langle \lambda, MM^{-1}y_i \rangle = \langle \lambda, y_i \rangle,\ i \in \overline{1,n}.$$

Hence, $T(\lambda') \equiv T(\lambda)$ is the optimal solution of **R**. y_i', $i \in I$ are unit vectors and moreover, $\langle \lambda', y_i' \rangle = \langle \lambda, y_i \rangle = 0$, $i \in I$. Hence, $\lambda_j' = 0$, $j \in \overline{1,m}$. Since $\langle \lambda', \overline{b} \rangle > 0$, $\lambda_0' \neq 0$ and has the same sign as $\overline{b}_0'$. Then

$$T(\lambda') = \{i \in \overline{1,n}, \langle \lambda', y_i' \rangle > 0\} = \{i \in \overline{1,n}, \lambda_0' y_{0i}' > 0\} = T_0$$

is an optimal solution of **R**. U_1 was defined to be the minimal among the sets $T_0, \dots, T_m$, hence $U_1 \equiv T_0$. The last means that the heuristic U_1, applied for the new basis B, produces the optimal solution of the problem **R**. □

Using the last result, a local-search algorithm suggests itself. Let $f(B) = |U_1(B)|$ and a neighborhood of B be the set $N(B)$ of all bases that differ from B only by one element (neighbors of B in the usual sense). We minimize locally the function f by using the neighborhood system N. The local search procedure we use is shown on Figure 3. This routine is called several times with different starting points – the mutually exclusive bases of (2), found to derive a lower bound of **R**. The local-search heuristic is used to determine an initial lower bound for the branch-and-bound algorithm.

4.5 Computational experiment

In this section we describe the computational testing done to evaluate the performance of the proposed branch-and-bound (BB) algorithm and the local search (LS) heuristic. The discriminant power of the MIP approaches to the classification problem was investigated in detail in [3, 12, 15]. That is why we focus our study only on computational efficiency of our algorithms. We use a part of Joachimsthaler and Stam [11] simulation data experimental design also used in [12] and [1]. This approach allows comparing the computational behavior of the algorithms on easily generated benchmarks which are statistically identical samples rather than on identical instances.

We generate three-variate normal distributions with different means and dispersion matrices. Different settings of these parameters form three cells. Table 1 shows the population parameters for each cell. I is the unit matrix and $e = (1, 1, 1)^T$.

```
localsearch(problem R)
{
  while (1)
  {
    U=U1(R);
    for(j=0;j<=m;j++)
      for(i=1; i<=n; i++)
      {
        if (R.y[j][i]==0) continue;
        newR=R;
        make a simplex pivot with pivoting element newR.y[j][i];
        if (|U1(newR)|<|U|) break;
      }
    if (|U1(newR)|<|U|) R=newR;
    else return U;
  }
}
```

Figure 4.3: Local search heuristic

Cell	Dispersion matrices		Mean vectors	
	$\boldsymbol{G_1}$	$\boldsymbol{G_2}$	$\boldsymbol{G_1}$	$\boldsymbol{G_2}$
1	$\boldsymbol{I}$	$\boldsymbol{I}$	0	$\boldsymbol{.5e}$
2	$\boldsymbol{I}$	$\boldsymbol{2I}$	0	$\boldsymbol{.6e}$
3	$\boldsymbol{I}$	$\boldsymbol{4I}$	0	$\boldsymbol{.8e}$

Table 4.1: Parameter settings

Cell	BB	LS	PMM	BPMM
1	.2640	.2648	.258	.266
2	.2529	.2533	.255	.262
3	.2194	.2197	.220	.226

Table 4.2: Average misclassification rates

Cell	0	1	2
1	93	6	1
2	96	4	0
3	97	3	0

Table 4.3: Number of misclassifications of LS over BB

For each cell, 100 random samples were generated. Each sample contained 100 observations – 50 observations from G_1 and 50 observations from G_2. We compare our results with the results of Koehler and Erenguc [12] for their exact algorithm PMM and heuristic BPMM. Table 2 shows the average misclassification rates for each cell of the study. The surprisingly good performance of LS is seen in this table.

The closeness of LS to the optimal solution becomes more obvious from Table 3. It shows the number of misclassifications produced by LS over those of BB. As it is seen in this table, among all of the 300 generated samples, only in 14 LS differs from the optimum. In no case LS misclassified more than two observations over BB.

Tables 4 through 8 summarize the statistics for the required computational effort. Table 4 shows the number of the nodes of the tree generated by BB. Table 5 shows the number of linear programs (LPs) solved by BB (by LP we mean finding of a feasible basis of a system (2_I) for some I). Note that in a certain node of the tree, more than one LP may be solved because of the searching of lower bounds. Tables 6 and 7 show the number of pivots required by BB and LS respectively. The number of pivots required by BB includes the number of pivots required by LS because LS was used to obtain an initial upper bound for BB.

As it is seen in these tables, the computational effort required by BB sharply decreases when the group overlap is smaller. For example, when the average misclassification rate decreases only by 0.0335 (cells 2–3), the average number of pivots is approximately 2.5 times lower. The

Cell	Average	St. deviation	Minimum	Maximum
1	8,532	10,774	174	62,298
2	7,072	9,134	81	44,538
3	2,405	2,880	47	19,816

Table 4.4: Number of nodes of the tree generated by BB

Cell	Average	St. deviation	Minimum	Maximum
1	19,734	25,280	417	143,972
2	16,041	21,052	242	99,744
3	5,327	6,565	115	45,679

Table 4.5: Number of LPs solved by BB

Cell	Average	St. deviation	Minimum	Maximum
1	137,377	159,238	13,581	897,485
2	101,778	109,990	12,453	507,973
3	39,849	33,016	10,313	236,037

Table 4.6: Number of pivots performed by BB

computational effort required by LS is relatively constant and significantly lower.

For comparison, Table 8 shows some average results from the study of Koehler and Erenguc for their exact algorithm. Linear programs solved by their algorithm have a matrix of the same size as ours but they also have an objective function and a simple lower bound for each variable. A minor pivot is one in which a nonbasic variable switches its boundary value. A major pivot is one where a nonbasic variable is pivoted into basis. Major pivots are computationally more expensive than the minor ones. As there are no simple lower bounds for the variables in our models, we perform only major pivots.

4.6 Conclusions

In this paper we proposed exact and heuristic algorithms for solving the two-group classification problem, formulated into its most difficult form, where the goal is a hyperplane directly minimizing the number of misclassified points. The algorithms are based on some specific properties of the vertices of a polyhedral set neatly connected with the model, which is far from the commonly used mixed-integer model. The results of the conducted computational experiment characterize the proposed branch-and-bound algorithm as an efficient algorithm for minimizing the number of the misclassifications in the training sample especially when the group overlap is relatively small. The local search heuristic produces solutions, very close to the optimum and requires

Cell	Average	St. deviation	Minimum	Maximum
1	13,538	1,703	8,712	17,400
2	12,834	1,712	8,705	17,667
3	11,283	1,383	8,353	14,287

Table 4.7: Number of pivots performed by LS

Cell	LPs	Minor pivots	Major pivots
1	99,417	229,612	62,720
2	100,284	233,216	64,437
3	61,821	161,663	53,485

Table 4.8: Some average results from the study of Koehler and Erenguc

significantly lower computational effort, so it is an valuable alternative to the exact methods. The algorithms may be a subject to further refinement. We did not investigate, for example, the effects of changing the order of processing the nodes of the tree, the order of choosing basic variables etc. Some more sophisticated (but more expensive) ways of finding mutually exclusive bases for generating lower bounds than the simple “greedy” technique may also be tested. The approach we developed for the local search algorithm may also be used by other heuristics. It was already successfully applied in [10] in the context of a tabu search algorithm.

Bibliography

[1] Abad, P., and Banks, W. New LP based heuristics for the classification problem. European Journal of Operational research, 1993, 67, 88-100.

[2] Amaldi, E., and Kann, V. On the approximability of removing the smallest number of relations from linear systems to achieve feasibility. 1994, Technical report ORWP-6-94, Department of Mathematics, Swiss Federal Institute of Technology, Lausanne, and Technical Report TRITA-NA-9402, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm.

[3] Bajgier, S., and Hill, A. An experimental comparison of statistical and linear programming approaches to the discriminant problem. Decision Sciences, 1982, 13, 604-618.

[4] Banks, W., and Abad, P. An efficient optimal solution algorithm for the classification problem. Decision sciences, 1991, 22, 1008-1023.

[5] Fisher, R. The utilization of multiple measurements in taxonomic problems. Annals of Eugenics, 1936, 7, 179-188.

[6] Freed, N., and Glover, F. A linear programming approach to the discriminant problem. Decision Sciences, 1981, 12, 68-74.

[7] Freed, N., and Glover, F. Simple but powerful goal programming models for discriminant problems. European Journal of Operational research, 1981, 7, 44-60.

[8] Freed, N., and Glover, F. Evaluating alternative linear programming models to solve the two-group discriminant problem. Decision sciences, 1986, 17, 151-162.

[9] Glover, F., Keene, S., and Duea, B. A new class of models for the discriminant problem. Decision sciences, 1988, 19, 269-280.

[10] Hanafi S., Yanev N., and Freville A. A Tabu Approach for the Classification Problem, Joint International Meeting EURO XV – INFORMS XXXIV, Barcelona, Spain, July 14-17, 1997, also available as RR-97-3, LIMAV, University of Valenciennes, March, 1997

[11] Joachimsthaler, E., and Stam, A. Four approaches to the classification problem in discriminant analysis: an experimental study. Decision sciences, 1988, 19, 322-333.

[12] Koehler, G., and Erenguc, S. Minimizing misclassifications in linear discriminant analysis. Decision Sciences, 1990, 21, 63-85.

[13] Markowski, C., and Markowski, E. An experimental comparison of several approaches to the discriminant problem with both qualitative and quantitative variables. European Journal of Operational research, 1987, 28, 74-78.

[14] Ragsdale, C., and Stam, A. Mathematical programming formulations for the discriminant problem: An old dog does new tricks. Decision sciences, 1991, 22, 296-307.

[15] Stam, A. and Joachimsthaler, E. A comparison of a robust mixed-integer approach to existing methods for establishing classification rules for the discriminant problem. European Journal of Operational research, 1990, 46, 113-122.

appendix A

1 Recent Advances in Solving the Protein Threading Problem[†]

R. Andonov[†], G. Collet[†], J-F. Gibrat[‡], A. Marin[‡], V. Poirriez[§], N. Yanev[⋆]

[†] IRISA, Campus de Beaulieu, 35042 Rennes, France,

[‡] INRA, Unité Mathématique Informatique et Génome UR1077, F-78352 Jouy-en-Josas,

[§] LAMIH, UMR CNRS 8530, Université de Valenciennes,59313 Valenciennes, France,

[⋆] University of Sofia, 5 J. Bouchier Str., 1126 Sofia, Bulgaria

1.1 INTRODUCTION

Genome sequencing projects generate an exponentially increasing amount of raw genomic data. For a number of organisms whose genome is sequenced, very little is experimentally known, to the point that, for some of them, the first experimental evidence gathered is precisely their DNA sequence. In the absence, or extreme paucity, of experimental evidences, bioinformatic methods play a central role to exploit the raw data. The bioinformatic process that extracts biological knowledge from raw data is known as annotation.

Annotation is composed of two phases:

1. a static phase whose purpose is to describe the basic “objects” that are found in the genome: the genes and their products the proteins.

2. a dynamic phase that seeks to describe the processes, i.e., the complex ways genes and proteins interact to create functional networks that underly the biological properties of the organism.

The first phase is the cornerstone of the annotation process. The first step consists in finding the precise location of genes on the chromosome. Then, for those genes that encode proteins, the next step is to predict the associated molecular, cellular and phenotypic functions. This is often referred to as *in silico* functional annotation.

[†]**Partially supported by ANR grant Calcul Intensif projet PROTEUS (ANR-06-CIS6-008) and by Hubert Curien French-Bulgarian partnership “RILA 2006” N^0 15071XF**

Different methods exist for predicting protein functions, the most important of which are based on properties of homologous proteins.

Homology is a key concept in biology. It refers to the fact that two proteins are related by descent from a common ancestor. Homologous proteins have the following properties:

- they may have sequences that, despite the accumulated mutations, still resemble the ancestor sequence;
- their three-dimensional structures are similar to the structure of the ancestor;
- they may have conserved the ancestor function, or at least a related function.

Therefore the principle of *in silico* functional analysis, based on homology searches, is to infer a homology relationship between a protein whose function is known and the new protein under study then to transfer the function of the former to the latter.

The inference of the homology relationship is based on the previously listed properties of homologous proteins. The first methods developed used the first property, the conservation of the sequences, and were based on sequence comparisons using alignment tools such as PSI-BLAST [1].

These methods are still the workhorses of *in silico* functional annotation: they are fast and endowed with a very good statistical criterion allowing to judge when two proteins are homologous. Unfortunately they also have a drawback. They are very inefficient when the proteins under study happen to be remote homologs, i.e., when their common ancestor is very ancient. In such a case the sequences may have undergone many mutations and they are no longer sufficiently similar for the proteins to be recognized as homologous.

For instance, when analyzing prokaryote genomes, these techniques cannot provide any information about the function of a noticeable fraction of the genome proteins (between 25% and 50% according to the organism studied). Such proteins are known as "orphan" proteins. One also speaks of orphan families when several homologous proteins are found in newly sequenced genomes that cannot be linked to any protein with a known function.

To overcome this problem new methods have been developed that are based on the second property: the good conservation of the 3D structure of homologous proteins. These methods are known as threading methods, or more formally, as fold[1] recognition methods.

The rational behind these methods is threefold:

1. As mentioned above, 3D structures of homologous proteins are much better conserved than the corresponding amino acid sequences. Numerous cases of proteins with similar folds and the same function are known, though having less than 20% sequence identity [2].

[1] in this context fold refers to the protein 3D structure

2. There is a limited, relatively small, number of protein structural families. Exact figures are still a matter of debate and vary from 1 000 [3] to at most a few thousands [4]. According to the statistics of the Protein Data Bank (PDB)[2] there are about 700 (CATH definition [5]) or 1 000 (SCOP definition [6]) different 3D structure families that have been experimentally determined so far.

3. Different types of amino acids have different preferences for occupying a particular structural environment (being in an α-helix, in a β-sheet, being buried or exposed). These preferences are the basis for the empirically calculated score functions that measure the fitness of a given alignment between a sequence and a 3D structure.

Based on these facts, threading methods consist in aligning a query protein sequence with a set of 3D protein structures to check whether the sequence might be compatible with one of the structures. These methods consist of the following components:

- a database of representative 3D structural templates;
- an objective function (score function) that measures the fitness of the sequence for the 3D structure;
- an algorithm for finding the optimal alignment of a sequence onto a structural template (with respect to the objective function);
- a statistical analysis of the raw scores allowing the detection of the significant sequence-structure alignments.

To develop an effective threading method all these components must be properly addressed. A description of the implementation of these different components in the FROST (Fold Recognition Oriented Search Tool) method [7] is detailed in the next section. Let us note that, from a computer scientist's viewpoint, the third component above is the most challenging part of the treading method development. It has been shown that, in the most general case, when variable length alignment gaps are allowed and pairwise amino acid interactions are considered in the score function, the problem of aligning a sequence onto a 3D structure is NP-hard [8]. Until recently, it was the main obstacle to the development of efficient threading methods. During the last few years, much progress has been accomplished toward a solution of this problem for most real life instances [9, 10, 11, 12, 13, 14].

Despite these improvements, threading methods, like a number of other bioinformatic applications, have high computational requirements. For example, in order to analyze the orphan proteins that are found in prokaryote genomes, a back of the envelop computation shows that one needs to align 500 000[3] protein sequences with

[2]http://www.rcsb.pdb/

[3]This figure corresponds to the number of sequenced genomes (500) times the average number of proteins per genome (3 000) times the mean fraction of orphan proteins ($\frac{1}{3}$)

at least 1 000 3D structures. This represents 500 millions alignments. Solving such quantity of alignments is, of course, not easily tractable on a single computer. Only a cluster of computers, or even a grid, can manage such amount of computations. Fortunately, as we will show hereafter, it is relatively straightforward to distribute these computations on a cluster of processors or over a grid of computers.

Grids are emerging as a powerful tool to improve bioinformatic applications effectiveness, particularly for protein threading. For example, the encyclopedia of life project [15] integrates 123D+ threading package in its distributed pipeline. All the pipeline processes, from DNA sequence to protein structure modeling, are parallelized by a grid application execution environment called APST (for Application-level scheduling Parameter Sweep Template). Another distributed pipeline for protein structure prediction is proposed by T. Steinke and al. [16]. Their pipeline consists in three steps : a pre-processing phase by sequence alignments, a protein threading phase and a final 3D refinement. Their threading algorithm solves the alignment problem by a parallel implementation of a Branch-&-Bound optimizer using the score function of Xu and al. [17]. With a cluster of 16 nodes, they divided by 2 the computation-time of aligning 572 sequences with about 37 500 structures from the PDB.

To maintain a structural annotation database up to date (project e-protein[4]), McGuffin and colleagues describe a fold recognition method distributed on a grid with the JYDE (Job Yield Distribution Environment) system which is a meta-scheduler for clusters of computers. To annotate the human genome, they use their mGen THREADER software integrated with JYDE on three different grid systems. On these three independent clusters of 148, 243 and 192 CPUs (515 CPUs), the human genome annotations can be updated in about 24 hours.

The rest of this chapter is organized as follows. In section 1.2 we present basic features of the FROST method. Section 1.3 further details the mathematical techniques used to tackle the difficult problem of aligning a sequence onto a 3D structure. Section 1.4 introduces the different operations required in FROST to make the entire procedure modular and describes how the modules can be distributed and executed in parallel on a cluster of computers. Computational benchmarks of the parallelized version of FROST are presented in section 1.5. In section 1.6 we discuss future research directions.

1.2 FROST: A FOLD RECOGNITION METHOD

1.2.1 Definition of protein cores

Threading methods require a database of representative 3D structures. The Protein Data Bank (PDB) that gathers all publicly available 3D structures contains about 40 000 structures. However this database is extremely redundant. Analyses of the

[4]http://www.e-protein.org/

PDB show that it contains at most about 1 000 different folds [6]. In theory only these folds need to be taken into consideration. In practice, to obtain a denser coverage of the 3D structure space, the PDB proteins are clustered into groups having more than 30% sequence identity and the best specimen of each group (in terms of quality of the 3D structure : high resolution, small R-factor, no, or few, missing residues) is selected. The final database contains about 4 500 3D structures.

For the purpose of fold recognition, the whole 3D structure is not required, only those parts of the structure which are characteristic of the structural family need to be considered. This leads to the notion of structural family core. The core is defined as those parts that are conserved in all the 3D structures of the family and are thus distinctive of the corresponding fold.

There are two practical reasons for using cores:

1. aligning a sequence onto portions of the 3D structure that are not conserved is likely to introduce a noise that would make the detection process more difficult;

2. by definition, no insertion or deletion is permitted within core elements since, otherwise, they would not be conserved parts of the family structures.

In protein families often one observes that the conserved framework of the 3D structure consists of the periodic secondary structures, α helices and β strands, the loops at the surface of the protein are variables. Accordingly, in FROST the core of the protein structures is defined as consisting of the helices and strands.

Hereafter we will refer to cores instead of 3D structures or folds.

1.2.2 Score function

To evaluate the fitness of a sequence for a particular core we need an objective (or score) function. There are two categories of score functions: “local” and “non local”. The former ones are, in essence, similar to the score functions used in sequence alignment methods. The later consider pairs of residues in the core and are specific of threading methods.

In threading methods, a schematic description of the core structure is used instead of a full atomic representation. Each residue in the core is represented by a single *site*. In FROST it is the Cα of the residue in the structure. Each site is characterized by its *state* which is a simplified representation of its environment in the core. A state is defined by the type of secondary structure (α helix, β strand or coil) in which the corresponding residue is found and by its solvent accessibility (buried if less than 25% of the residue surface in the core is accessible to the solvent, exposed if more than 60% is accessible and intermediate otherwise). This defines 9 states, for instance the site is located in a helix and exposed, or in a strand and buried, etc.

In FROST we use a canonical expression for the score function. Altschul [18] has shown that the most general form of a score for comparing sequences is a log-likelihood:

$$score(r_i, r_j) = \log \frac{P(r_i r_j|E)}{P(r_i)P(r_j)}$$

The score of replacing amino acid r_i by amino acid r_j is the log of the ratio of two probabilities:

1. the probability that the two amino acids are related by evolution, i.e., they are aligned in the sequence because they evolved from the same ancestral amino acid;
2. the probability that the two amino acids are aligned just by chance.

If the two amino acids, on average, in a number of protein families, are observed more often aligned than expected by chance, i.e., if the numerator probability is greater than the product of the denominator probabilities then the ratio is greater than 1 and the score is positive. On the contrary if the two amino acids are observed to be less often aligned than expected by chance the score is negative.

These considerations led to the development of empirical substitution matrices (for instance the PAM [19] or BLOSUM matrices [20]) that gathers the scores for replacing a given amino acid by another one during a given period of evolution. Finding the optimal alignment score for two sequences amounts to maximizing the probability that these two sequences have evolved from a common ancestor as opposed to being random sequences (assuming that the alignment positions are independent).

Very similar matrices can be developed for threading methods, except that we now have at our disposal an extra piece of information: the three-dimensional structure of one of the sequences. Therefore we can define a set of nine state-dependent substitution matrices as:

$$score(R_i, r_j)_{S_k} = \log \frac{P(R_i r_j|E)_{S_k}}{P(R_i)_{S_k} P(r_j)} \tag{1.1}$$

where $P(R_i)_{S_i}$ is the probability of observing amino acid R_i in state S_k, $P(r_j)$ is the background probability of amino acid r_j in the sequence database and $P(R_i r_j|E)_{S_k}$ is the probability of observing amino acids R_i and r_j aligned in sites with state S_k in protein families. Note that throughout this section uppercases are used for residues that belong to the core and lower case for residues that belong to the sequence that is aligned onto the core.

This expression represents the score for replacing amino acid R_i by amino acid r_j *in a particular state* (see Figure 1.1). In addition, since we know the 3D structure, it is possible to use gap penalties that prevent insertion/deletion in core elements. This provides a score function that is local, i.e., a score depends on a single site in a particular sequence. However, with this kind of score, we do not use the real 3D structure but only some of its properties that are embodied in the state (type of secondary structure and solvent accessibility).

In order to, explicitly, take into account the 3D structure we must generalize these state-dependent substitution matrices. This is done by considering pairs of residues that are *in contact* in the core. In FROST residues are defined to be in contact in a

three-dimensional structure if there exists at least one pair of atoms, one atom from each residue side chain, for which the distance is less than a given cut-off value. The corresponding score function is defined as:

$$score(R_iR_j, r_kr_l)_{S_nS_m} = \log \frac{P(R_iR_jr_kr_l|E)_{S_nS_m}}{P(R_iR_j)_{S_nS_m}P(r_k, r_l)} \tag{1.2}$$

where $P(R_iR_j)_{S_nS_m}$ is the probability of observing the pair of amino acids R_i and R_j at sites that are in contact in protein 3D structures and are characterized, respectively, by states S_n and S_m. $P(r_k, r_l)$ is the background probability for the amino acid pair r_kr_l in the sequence database. $P(R_iR_jr_kr_l|E)_{S_nS_m}$ is the probability to observe the amino acid pair R_iR_j aligned with the amino acid pair r_kr_l in the structural context described by states S_nS_m in protein families.

This expression represents the score for replacing the pair of amino acids R_iR_j by the pair r_kr_l in sites that are characterized by states S_n and S_m and are in contact in protein cores (see Figure 1.1). There are 89 such matrices. This type of score function is non-local since it takes into account two sites in the sequence. As we will describe in the next section the fact that the score function is local or non-local has a profound influence on the type of algorithm that needs to be used for aligning the sequence onto the core.

1.2.3 Sequence-core alignment algorithms

For local score functions there exists very efficient algorithms to align sequences with cores. It is sufficient to borrow the algorithms used for sequence alignments and to make the slight modifications that are required to adapt them to our problem. These algorithms are all based on some forms of dynamic programming [21, 22] and thus are of $O(N^2)$, N being the size of the sequences. Besides, if the computational requirements are of prime importance, we also have available fast and accurate heuristics (such as BLAST and its variants [1] or FASTA [23]). As shown on Figure 1.1 the knowledge of the 3D structure of one of the sequence, permits the use of substitution matrices that are proper to the state of each site in the core. Secondary structure specific gap penalties can also be used, i.e., gap penalties that make insertions/deletions more difficult in helices or strands. In addition these techniques readily enable the use of sequence profiles instead of simple sequences, a procedure that is known to improve the sensitivity of sequence comparison methods [24].

On the contrary, non-local score functions do not permit the use of algorithms based on dynamic programming. Indeed, all dynamic programming techniques are based on a recursive procedure whereby an optimal solution for a given problem is built from previously found subproblem optimal solutions. For instance, for sequence alignments, the optimal score for aligning two substrings $s[1..i]$ and $t[1..j]$ is obtained from the optimal solutions previously found for aligning substrings $s[1..i-1]$ with $t[1..j-1]$, $s[1..i-1]$ with $t[1..j]$ and $s[1..i]$ with $t[1..j-1]$ by the following

H	*H*	*H*	*C*	*C*	*E*	*E*	*E*	*E*	–	–	–	*C*	*C*	*C*
e	*e*	*e*	*b*	*b*	*e*	*b*	*b*	*b*	–	–	–	*b*	*e*	*e*
He	*He*	*He*	*Cb*	*Cb*	*Ee*	*Eb*	*Eb*	*Eb*	–	–	–	*Cb*	*Ce*	*Ce*
M	*F*	*T*	*V*	*N*	*V*	*H*	*I*	*D*	–	–	–	*R*	*L*	*Y*
m	**w**	**t**	–	–	v	**h**	**v**	**e**	**h**	**g**	**v**	**r**	**v**	**y**
	•						•							

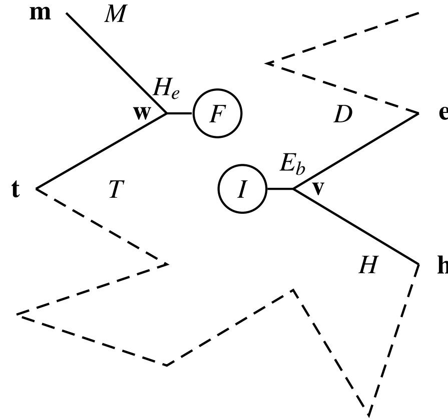


Fig. 1.1 **Upper part**: 1D alignment of two sequences the query sequence (5th row) is shown in bold lowercase letters, the core sequence (4th row) in slanted uppercase letters. The first row is the observed secondary structure: helix (H), strand (E) or coil (C). The second row is the solvant accessibility: exposed (e) or buried (b). The third row is the corresponding state. Deletion are indicated by dashes. In the core we focus on the 2nd and 8th sites, labelled with black circles. The state of the 8th site is Eb, that is, an exposed strand. To score this position in the core we must use $score(I,v)_{Eb}$ the score of replacing an isoleucine by a valine in an exposed strand environment ($R_i = I$, $r_j = v$ and $S_k = Eb$ in the corresponding equation). Note also that since we are in a strand a specific gap penalty must be used. **Lower part**: 3D alignment of the same two sequences. In the 3D structure the two above sites are in contact. To score this interaction we must use $score(FI,wv)_{HeEb}$ the score of replacing the pair FI by the pair wv in an exposed helical - buried strand environment ($R_i = F$, $R_j = I$, $r_k = w$, $r_l = v$, $S_n = He$ and $S_m = Eb$ in the corresponding equation). Here, since we are in core elements, no insertion/deletion is allowed.

recurrence expression:

$$A[i,j] = \max \begin{cases} A[i-1,j] + gp \\ A[i-1,j-1] + c(s[i],t[j]) \\ A[i,j-1] + gp \end{cases}$$

where $A[k,l]$ is the optimal score for aligning substring $s[1..k]$ with substring $t[1..l]$, gp is the cost of a gap and $c(s[i],t[j])$ is the cost for aligning the i-th letter of string s with the j-th letter of string t.

Non-local score functions ruin this recursive procedure since, now, the score for aligning two sequences does not exclusively depends on the optimal score of previous subsequences but also upon interactions with distant residues.

As a consequence, the first threading methods proposed relied on various heuristics to align sequences onto cores, for instance Madej et al. [25] used a stochastic technique in the form of a Gibbs Monte Carlo.

Lathrop [8] showed that, in the most general case, the problem of aligning a sequence onto a core with a non-local score function is NP-hard. A few years later, Akutsu and Miyano [26], showed that it is MAX-SNP-hard, meaning that there is no arbitrary close polynomial approximation algorithm, unless P = NP.

Lathrop and Smith [9] were the first to propose an algorithm, based on a branch & bound technique, that provided, for small instances, an exact solution to the problem. Uberbacher and colleagues [17], a couple of years later, described another algorithm based on a divide & conquer approach. These two algorithms were, apparently, rather slow and only able to cope with the easiest problems. They were not implemented in an actual threading method, to the best of our knowledge.

At the turn of the century, new methods based on advanced mathematical programming methods, Mixed Integer Programming (MIP), were developed [27, 28, 10, 11, 14] that were able to tackle the most difficult instances of the problem in a reasonable amount of time. Two protein threading packages are currently available that implement exact methods based on the latter approach: RAPTOR[5] [12] and FROST[6] [7]. In section 1.3 we will describe in more details the FROST implementation of the MIP models. Other interesting integer programming approaches for solving combinatorial optimization problems that originate in molecular biology are discussed in recent surveys [29, 30].

1.2.4 Significance of scores

Equipped with the above techniques we are able to get an optimal score for aligning any sequence onto a database of cores. We are now faced with the problem of the significance of this score. Let us assume that we have aligned a particular sequence with a core and got a score of 60. What does this score of 60 mean? Is it representative of a sequence that is compatible with the core? In other words, if we align a number of randomly chosen sequences with this core what kind of score distribution are we going to obtain? If, for a noticeable fraction of those alignments, one gets scores greater than or equal to 60 it is likely that the initial score is not very significant (unless of course all the chosen sequences are related to the core).

Similar questions arise when one compares two sequences. Statistical analyses have been carried out to study this problem and it has been shown [31] that the distribution of scores for ungapped local alignments of random sequences follows an extreme value distribution. The parameters of this distribution can be analytically

[5] http://www.bioinformaticssolutions.com/
[6] http://genome.jouy.inra.fr/frost/

calculated from the features of the problem : type of substitution matrix used, size of the aligned sequences, background frequencies of the amino acids, etc. When gaped alignments are considered it is no longer possible to perform analytical calculations but computer experiments have shown that the shape of the empirical distribution is still an extreme value distribution whose parameters can be readily determined from a set of sequence comparison scores.

Such analytical calculation cannot be done for a sequence-core alignment. In fact we do not even know the shape of the score distribution for aligning randomly chosen sequences onto cores although some preliminary work seems to indicate that it could also be an extreme value distribution [32].

In FROST, to solve this problem, we adopt a pragmatic, but rather costly, approach. For each core, we randomly extract from the database five sets of 200 sequences unrelated to the core. Each set contains sequences whose size corresponds to a percentage of the core size, i.e., 30% shorter, 15% shorter, same size as the core, 15% longer and 30% longer. The assumption behind this procedure is that when a sequence is compatible with a core, its length must be similar to the core length (± 30%). [7] We align the sequences of each set with the core. This provides empirical distributions of scores for aligning sequences with different lengths onto the core. For each distribution we determine the median and the third quartile and we compute a normalized score as :

$$S_n = \frac{S - q_2}{q_3 - q_2}$$

where S_n is the normalized score, S is the score of the query sequence, q_2 and q_3 are, respectively, the median and third quartile of the empirical distribution.

This normalized score allows us to compare the alignments of the query sequence onto different cores. The larger the normalized score the more probable the existence of a relationship between the sequence and the core. Indeed, a large normalized score indicates that the query sequence is not likely to belong to the population of unrelated sequences from which the score distribution was computed. Unfortunately, since we do not know the shape nor the parameters of the distributions, we cannot compute a precise probability for the sequence to belong to this population of unrelated sequences. We use empirical results obtained on a test database to estimate when a normalized score is significant at a 99% level of confidence [33, 7] (see next section).

When we need to align a new query sequence whose length is not exactly one of the above lengths that were used to pre-calculate the score distributions, we linearly interpolate the values of the median and the third quartile from those of the two nearest distributions. For instance if the size of new query sequence is 20% larger than the size of the core, the corresponding median and third quartile values are given by:

[7] This is the assumption in case of a global alignment. In section 1.6 we will consider more general types of alignments: semi-global and local, for which this assumption does not hold.

$$q_n^{20} = q_n^{15} + \frac{20-15}{30-15}(q_n^{30} - q_n^{15})$$

where q_n^L represents the median ($n = 2$) or the third quartile ($n = 3$) of the score distribution when sequences of length L are aligned onto the core.

1.2.5 Integrating all the components: the FROST method

FROST is intended to assess the reliability of fold assignments to a given protein sequence (hereafter called a query sequence or query for short) [33, 7]. To perform this task, FROST used a series of filters, each one possessing a specific scoring function that measures the fitness of the query sequence for template cores. The version we describe, here, possesses two filters.

The first filter is based on a fitness function whose parameters involve only a local description of proteins and corresponds to Eq. (1.1). This filter belongs to the category of profile-profile alignment methods and is called 1D filter. The algorithm used to find the optimal alignment score is based on dynamic programming techniques.

The second filter employs the non local score function (1.2). Because it makes use of spatial information, it is called a 3D filter in the following. As explained in section 1.2.3, this type of score function requires dedicated algorithms for aligning the query sequence onto the cores. The algorithm used in FROST, based on a MIP model, is further described in the next section.

FROST functions as a sieve. The 1D filter is fast, owing to its dynamic programming algorithm of quadratic complexity. It is used to compare the query sequence with all the database cores and rank them in a list according to the normalized scores. Only the first N cores from this list are, then, passed to the 3D filter and aligned with the query sequence.

Each of the N above cores is now characterized by two normalized scores, one for the 1D filter and one for the 3D filter. These scores can be plotted on a 2 dimensional diagram. As shown on Figure 1.2 this allows us to define the area on the diagram, delimited by line equations connecting the scores, that empirically provides a 99% confidence threshold.

Several score functions, other than the ones described in section 1.2.2, can be developed. The only point that matters is whether these functions are local or non-local. The same sieve principle as the one described for the two above score functions is still applicable. The difference is that now the N resulting cores are characterized by a number of scores greater than two. This makes the visual inspection as explained above difficult and one must rely, for instance, on a Support Vector Machine (SVM) algorithm to find the hyperplanes that separate positive from negative cases.

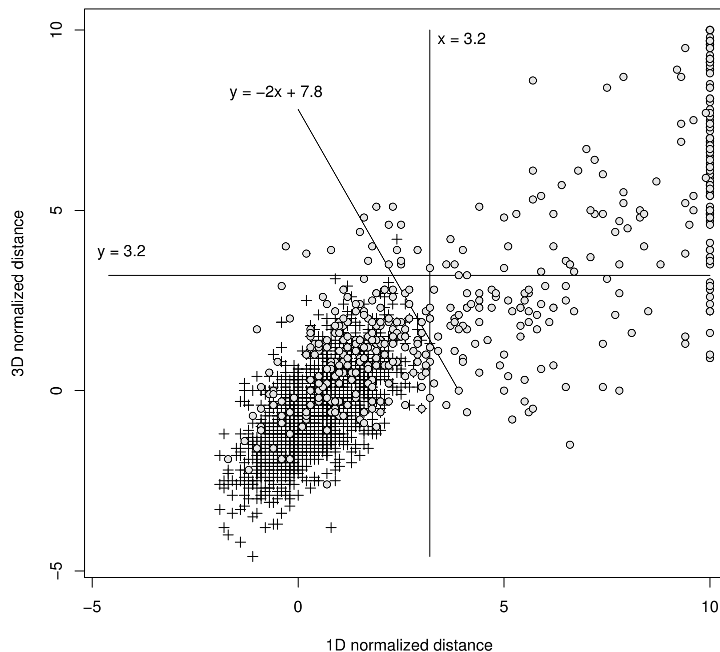


Fig. 1.2 Plot of the 1D score (along the x-axis) and 3D score (along the y-axis) for different (Q,C) pairs (where Q is a query sequence and C is a core). Grey open circles represent (Q,C) pairs that are related, black crosses (Q,C) pairs that are not related, that is, respectively, the query sequence is known to have the same 3D structure as the core and the query sequence is known to have a different 3D structure from the core. The area beyond the lines indicated on the plot contains only 1% black crosses, which are thus false positives. For this example the recall is 60% [7].

1.3 FROST: A COMPUTER SCIENCE VISION

1.3.1 Formal definition

In this section we give a more formal definition of protein threading problem (PTP) and simultaneously introduce some existing terminology. Our definition is very close to the one given in [9, 34]. It follows a few basic assumptions widely adopted by the protein threading community [11, 35, 9, 34, 12, 17]. Consequently, the algorithms presented in the next sections can be easily plugged in most of the existing fold recognition methods based on threading.

Query sequence A query sequence is a string of length N over the 20-letter amino acid alphabet. This is the amino acid sequence of a protein of unknown structure which must be aligned with core templates from the database.

Core template All current threading methods replace the 3D coordinates of the known structure by an abstract template description in terms of core blocks or segments, neighbor relationships, distances, environments, as explained in section 1.2.2. This avoids the computational cost of atomic-level mechanics in favor of a more abstract, discrete representation of alignments between sequences and cores.

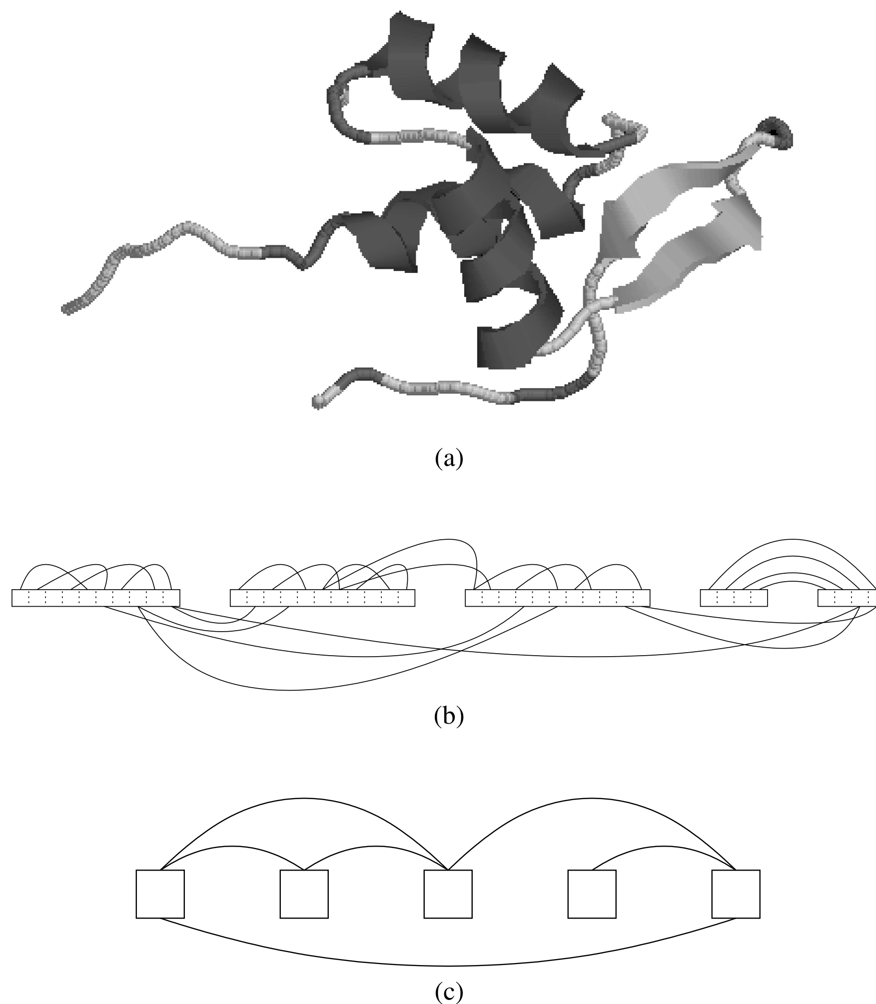
We consider that a core template is an ordered set of m segments or blocks. Segment i has a fixed length of l_i amino acids. Adjacent segments are connected by variable length regions, called loops (see Fig. 1.3(a)). Segments usually correspond to the most conserved parts of secondary structure elements (α-helices and β-strands). They trace the path of the conserved fold. Loops are not considered as part of the conserved fold and consequently, the pairwise interactions between amino acids belonging to loops are ignored. It is generally believed that the contribution of such interactions is relatively insignificant. The pairwise interactions between amino acids belonging to segments are represented by the so-called contact map graph (see Fig. 1.3(b)). Different definitions for residues in contact in the core can be used, for instance in [12] they assume that two amino acids interact if the distance between their C_β atoms is within p Å and they are at least q positions apart along the template sequence (with $p = 7$ and $q = 4$). There is an interaction between two segments, i and j, if there is at least one pairwise interaction between amino acids belonging to i and amino acids belonging to j. Let $L \subseteq \{(i, j) \mid 1 \leq i < j \leq m\}$ be the set of segment interactions. The graph with vertices $\{1, \ldots, m\}$ and edges L is called generalized contact map graph (see Fig. 1.3(c)).

Alignments Let us note, first, that in this section we adopt an inverse perspective and describe the alignment of a sequence onto a core as positioning the segments along the sequence. The problem remains exactly the same but it is easier to describe this way. Such an alignment is called feasible if the segments preserve their original order and do not overlap (see Fig 1.4(a)). An alignment is completely determined by the starting positions of all segments along the sequence. In fact, rather than absolute positions, it is more convenient to use relative positions. If segment i starts at the kth query sequence character, its relative position is $r_i = k - \sum_{j=1}^{i-1} l_j$. In this way the possible (relative) positions of each segment vary between 1 and $n = N + 1 - \sum_{i=1}^{m} l_i$ (see Fig. 1.4(b)). The set of feasible alignments is

$$\mathcal{T} = \{(r_1, \ldots, r_m) \mid 1 \leq r_1 \leq \cdots \leq r_m \leq n\}. \tag{1.3}$$

The number of possible alignments (the search space size of PTP) is $|\mathcal{T}| = \binom{m+n-1}{m}$, which is a huge number even for small instances (for example, if $m = 20$ and $n = 100$ then $|\mathcal{T}| \approx 2.5 \times 10^{22}$).

Most of the alignment methods impose an additional feasibility condition, upper and lower bounds on the lengths of query zones not covered by segments (loops).



(a)

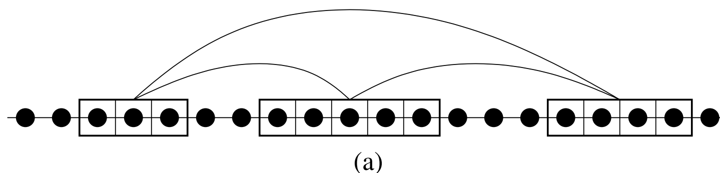
(b)

(c)

Fig. 1.3 (a) 3D structure backbone showing α-helices, β-strands and loops. (b) The corresponding contact map graph. (c) The corresponding generalized contact map graph.

This condition can be easily incorporated by a slight modification in the definition of relative segment position.

In the above definition, gaps are not allowed within segments. They are confined to loops. As explained above, the biological justification is that segments are conserved so that the probability of insertion or deletion within them is very small.



(a)

abs. position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
rel. position block 1	1	2	3	4	5	6	7	8	9											
rel. position block 2				1	2	3	4	5	6	7	8	9								
rel. position block 3									1	2	3	4	5	6	7	8	9			

(b)

Fig. 1.4 (a) Example of alignment of query sequence of length 20 and template containing 3 segments of lengths 3, 5 and 4. (b) Correspondence between absolute and relative block positions.

1.3.2 Network flow formulation

This section follows the formulation proposed in [27, 10]. In order to develop appropriate mathematical models, PTP is restated as a network optimization problem. Let $G(V,A)$ be a digraph with vertex set V and arc set A. The vertex set V is organized in columns, corresponding to segments from the aligned core. In each column, each vertex correspond to a relative position of the corresponding segment along the sequence. Then $V = \{(i,j) \mid i = 1,...,m,\ j = 1,...,n\}$ with m the number of segments and n the number of relative positions (see Fig. 1.5 where $m = 6$ and $n = 3$). A cost C_{ij} is associated to each vertex (i,j) as defined by the scoring function (1.1). The arc set is divided into two subsets : A' is a subset containing arcs between adjacent segments and A'' contains arcs between remote segments. Thus $A = A' \cup A''$ with

$$A' = \{((i,j),(i+1,l)) \mid i = 1,...,m-1,\ 1 \le j \le l \le n\}$$
$$A'' = \{((i,j),(k,l)) \mid (i,k) \in L,\ 1 \le j \le l \le n\}$$

To each arc $((i,j),(k,l))$ is associated a cost D_{ijkl} as defined by the scoring function (1.2). The arcs from A' will be referred as x-arcs and the arcs from A'' as z-arcs.

By adding two extra vertices S and T and the corresponding arcs $(S,(1,k))$, $k = 1,...,n$ and $((m,l),T)$, $l = 1,...,n$, (considered as x-arcs) one can see the one-to-one correspondence between the set of the feasible threadings and the set of the S-T path on x-arcs in G. We say that a S-T path *activates* its vertices and x-arcs. A z-arc is *activated* by a S-T path if both ends are on the path. We call the subgraph induced by the x-arcs of an S-T path and the activated z-arcs *augmented path*. Then PTP is equivalent to finding the shortest augmented path in G. Fig. 1.5 illustrates this correspondence.

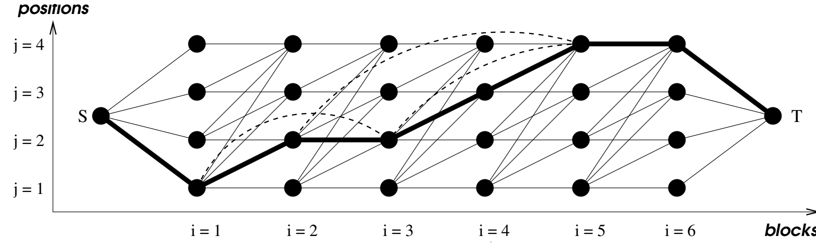


Fig. 1.5 Example of alignment graph. The path in thick lines corresponds to the threading in which the positions of the blocks are 1,2,2,3,4,4. Dashed line arcs belongs to A'' where the set of segment interactions is $L = \{(1,3),(2,5),(3,5)\}$.

1.3.3 Integer programming formulation

Let y_{ij} be binary variables associated with vertices in the previous network. Then y_{ij} is one if segment i is at position j and zero otherwise (vertex (i,j) is activated or not). Let Y be the polytope defined by the following constraints :

$$\sum_{j=1}^{n} y_{ij} = 1 \qquad i = 1,\dots,m \tag{1.4}$$

$$\sum_{l=1}^{j} y_{il} - \sum_{l=1}^{j} y_{i+1,l} \geq 0 \qquad i = 1,\dots,m-1,\ j = 1,\dots,n-1 \tag{1.5}$$

$$y_{ij} \in \{0,1\} \qquad i = 1,\dots,m,\ j = 1,\dots,n \tag{1.6}$$

Constraint (1.4) ensures that each block is assigned to exactly one position. Constraint (1.5) describes a non-decreasing path in the alignment graph. These constraints are illustrated in Fig1.6.

In order to take into account the interaction costs, we introduce a second set of variables $z_{ijkl} \geq 0$, with $(i,k) \in L$ and $1 \leq j \leq l \leq n$. These variables correspond to x-arcs and z-arcs in the network flow formulation. For the sake of readability, we will use the notation z_a for z_{ijkl} with $a \in A$ the arc set. The variable z_{ijkl} is set to one if the corresponding arc is activated. Then, we define the following constraints :

$$y_{ij} = \sum_{l=j}^{n} z_{ijkl} \qquad (i,k) \in L,\ j = 1,...,n \tag{1.7}$$

$$y_{kl} = \sum_{j=1}^{l} z_{ijkl} \qquad (i,k) \in L,\ l = 1,...,n \tag{1.8}$$

$$z_a \geq 0 \qquad a \in A \tag{1.9}$$

These constraints ensure that setting variables y_{ij} and y_{kl} to one (the path passes through these two points), activates the arc z_{ijkl}. Finding the shortest augmented path

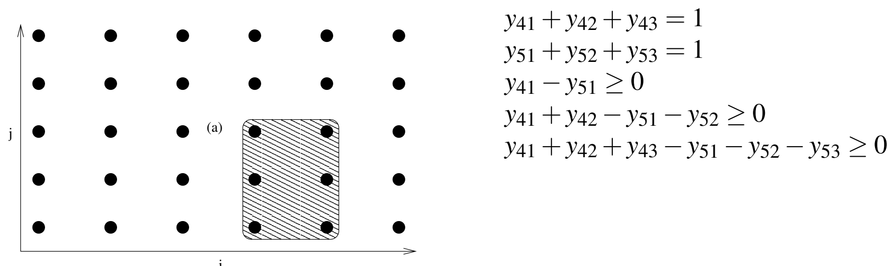


Fig. 1.6 The effect of constraints (1.4) and (1.5) on zone (a). Exactly one vertex is activated in column four and in column five. Activating a vertex at position $(4, j)$ guarantees that no vertex is activated in column five below j. If a vertex is activated in $(5, j)$, then a vertex must be activated in column four below j.

in graph G (i.e. solving PTP) is then equivalent to minimize the following function subject to the previous constraints :

$$\sum_{i=1}^{m} \sum_{j=1}^{n} C_{ij} y_{ij} + \sum_{a \in A} D_a z_a \tag{1.10}$$

This model, introduced in [11], is known as MYZ model. It significantly outperforms the MIP model used in the RAPTOR package [12] for all large instances (see [11] for more details). Both models (MYZ and RAPTOR) are solved using a linear programming relaxation (LP). The advantage of these models is that their LP relaxations give the optimal solution for most of the real-life instances. They have significantly beter performance than the branch & bounds approach proposed in [9]. Their drawback is their huge size (both number of variables and number of constraints) which makes even solving the LP relaxation slow. In the next section we present more efficient approaches for solving these models. They are based on Lagrangian relaxation.

1.3.4 Lagrangian approaches

Consider an integer program

$$z_{IP} = \min\{cx : x \in S\}, \text{where } S = \{x \in Z_+^n : Ax \geq b\} \tag{1.11}$$

Relaxation and duality are the two main ways of determining z_{IP} and upper bounds for z_{IP}. The linear programming relaxation is obtained by changing the constraint $x \in Z_+^n$ in the definition of S by $x \geq 0$. The Lagrangian relaxation is very convenient for

problems where the constraints can be partitioned into a set of "simple" ones and a set of "complicated" ones. Let us assume for example that the complicated constraints are given by $A^1x \geq b^1$, where A^1 is $m \times n$ matrix, while the simple constraints are given by $A^2x \geq b^2$. Then for any $\lambda \in R^m_+$ the problem

$$z_{LR}(\lambda) = \min_{x \in Q}\{cx + \lambda(b^1 - A^1x)\}$$

where $Q = \{x \in Z^n_+ : A^2x \geq b^2\}$ is the Lagrangian relaxation of (1.11), i.e. $z_{LR}(\lambda) \leq z_{IP}$ for each $\lambda \geq 0$. The best bound can be obtained by solving the Lagrangian dual $z_{LD} = \max_{\lambda \geq 0} z_{LR}(\lambda)$. It is well known that the relation $z_{IP} \geq z_{LD} \geq z_{LP}$ holds.

1.3.5 Lagrangian relaxation

We show now how to apply Lagrangian relaxation (LR) taking Eq. (1.8) as a complicated constraint. Recall that this constraint insures that the y-variables and the z-variables select the same position of segment k. By relaxing such a constraint, we relax the right end of a z-arcs. This means that an arc can be activated even though its right end is not on the path, as it is illustrated in Fig1.7(a). For a fixed λ, the relaxed augmented path problem obtained in this way can be solved in a polynomial time using a dynamic programming (see [36]).

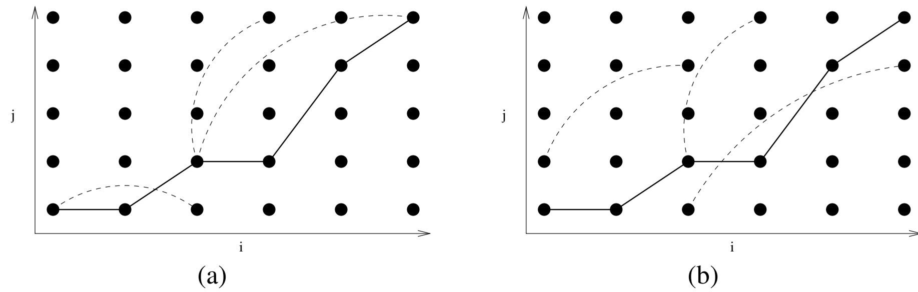


Fig. 1.7 Example of a threading instance with $m = 6$ blocks and $n = 5$ free positions. The set of segment interactions is $L = \{(1,3),(3,4),(3,6)\}$. (a) The Lagrangian relaxation sets the right end of any arc free. The solution for the relaxed problem could not satisfy the original constraints. (b) The Lagrangian relaxation sets both right and left ends of arcs free.

In order to find the Lagrangian dual z_{LD} one has to look for the maximum of a concave piecewise linear function. This appeals for using the so called sub-gradient optimization technique. For the function $z_{LR}(\lambda)$, the vector $s^t = b^1 - A^1x^t$, where x^t is an optimal solution to $\min_x\{cx + \lambda^t(b^1 - A^1x)\}$, is a sub-gradient at λ^t. The follow-

ing sub-gradient algorithm is an analog of the steepest ascent method for maximizing a function:

- (Initialization): Choose a starting point λ^0, Θ_0 and ρ. Set $t = 0$ and find a sub-gradient s^t.
- While $s^t \neq 0$ and $t < t_{\max}$ do $\{ \lambda^{t+1} = \lambda^t + \Theta_t s^t ; \Theta_{t+1} = \rho\Theta_t$, $t \leftarrow t+1$; find $s^t \}$

This algorithm stops either when $s^t = 0$, (in which case λ^t is an optimal solution) or after a fixed number of iterations $t_{\max}$. The parameter $0 < \rho < 1$ determines the decrease of the sub-gradient step.

Note that for each λ the solution defined by the y-variables is feasible for the original problem. In this way at each iteration of the sub-gradient optimization we have a heuristic solution. At the end of the optimization we have both lower and upper bounds on the optimal objective value.

Symmetrically, we can relax the left end of each link or even relax the left end of one part of the links and the right end of the rest (see figure 1.7(b)). This approach is used in [14]. The same paper describes a branch-and-bound algorithm using this Lagrangian relaxation instead of the LP relaxation. This is the default algorithm in the FROST package.

Another relaxation, called *cost-splitting* (CS), is presented in [37]. The results presented in this paper clearly show that CS slightly outperforms LR, and both (LR and CS) relaxations are significantly faster than LP (see Fig. 1.8). The interested reader can find further details concerning these approaches in [36].

1.4 DIVIDING FROST INTO MODULES FOR DISTRIBUTION OVER A CLUSTER

The following two sections are based on the results presented in [38].

1.4.1 Amount of computation to be done

In section 1.2.5 we described the FROST functioning. From a computational viewpoint, this procedure can be divided into 2 phases: the first one is the computation of score distributions (hereafter called phase D) and the second one is the alignment of the sequence of interest with the dataset of templates (hereafter called phase E for evaluation) making use of the previously calculated distributions. These two phases are repeated for each filter (1D and 3D). We denote by `Ali1D(Q,C)` the process of aligning a query sequence (Q) with a core (C) in the 1D filter and by `Ali3D(Q,C)` the more computer intensive alignment process of the 3D filter. Although we have a very efficient implementation of the corresponding algorithm based on a Lagrangian relaxation technique, computing the score distributions for all the templates takes more than a month when performed sequentially.

The whole procedure requires the following computations:

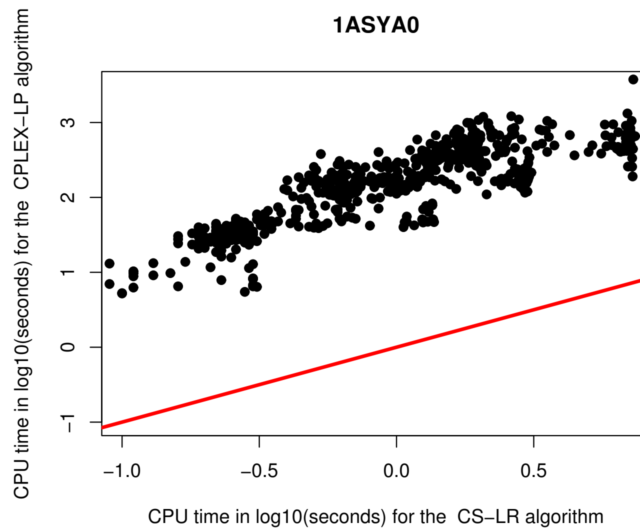


Fig. 1.8 Cost-Splitting Relaxation versus LP Relaxation. Plot of times in seconds with the CS algorithm on the x-axis and the LP algorithm from [11] on the y-axis. Both algorithms compute approximate solutions for 962 threading instances associated with the template 1ASYA0 from the FROST core database. The line $y = x$ is shown on the plot. A significant performance gap is observed between the algorithms. For example point $(x, y) = (0.5, 3)$ corresponds to a case where CS is $10^{2.5}$ times faster than LP relaxation. These results were obtained on an Intel(R) Xeon(TM) CPU 2.4 GHz, 2 GB RAM, RedHat 9 Linux. The MIP models were solved using CPLEX 7.1 solver (see [37] for more details).

1. Phase D: align non homologous sequences in order to obtain the scores distributions for all templates and all filters. Since five distributions are associated to any template, and there are about 200 sequences for each distribution, this procedure needs solving about 1,200,000 quadratic problems `Ali1D` and the same amount of NP-complete problems `Ali3D`.

2. Phase E: align the query with the dataset of templates which requires solving several hundreds of quadratic problems `Ali1D` and N NP-complete problems `Ali3D` (where N is usually ten).

Figure 1.9 shows the distribution of the alignment problems needed to be solved during phase D and gives an idea of the amount of computation required by the 3D filter. The number of the problems is about 1,200,000 while the size of the largest instance is 6.6 10^{77}.

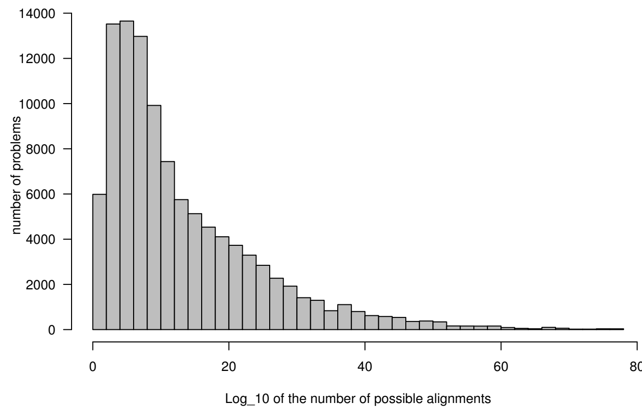


Fig. 1.9 Populations of the 3D problems solved during phase D as a $\log_{10}$ function of the size of the search space (number of possible alignments).

Figure 1.10 shows the plot of the mean CPU time required to solve the 3D problems involved in phase D as a function of the number of possible alignments[8].

The purpose of the procedure proposed in the next section is to distribute all these tasks.

Note that phase D needs to be repeated each time the fitness functions or the library of templates change, which is a frequent case when the program is used in a development phase.

1.4.2 Distribution of the computations: dividing FROST into modules

The first improvement in the distributed version (DFROST) compared to the original FROST consists in clearly identifying the different stages and operations in order to make the entire procedure modular. The process of computing the scores distributions is dissociated from the alignment of the query versus the set of templates. We therefore split the two phases (D and E) which used to be interwoven in the original implementation. Such a decomposition presents several advantages. Some of them are:

- Phase D is completely independent from the query, it can be performed as *a preprocessing stage* when it is convenient for the program designer.

[8]The mean CPU time here concerns macro-tasks each one containing ten (gran3D=10) instances Ali3D of the same size (see section 1.4.3)

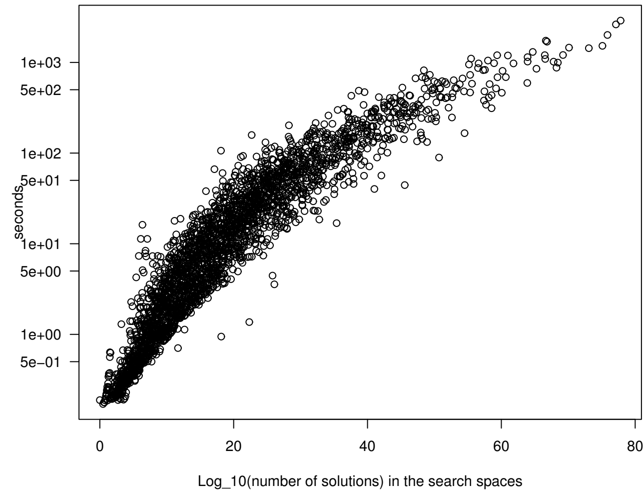


Fig. 1.10 Mean CPU time required to solve the 3D problems in phase D as a function of their size.

- The utilization of the program is *simplified*. Note that only the program designer is supposed to execute phase D, while phase E is executed by an "ordinary" user. From a user's standpoint DFROST is *significantly faster* than FROST, since only phase E is executed at his request (phase D being performed as a preprocessing step).

- The program designer can *easily carry out different operations* needed for further developments of the algorithm or for database updating such as: adding new filters, changing the fitness functions, adding a new template to the library, etc.

- This organization of DFROST in modules is very *suitable for its decomposition in independent tasks* that can be solved in parallel.

The latter point is discussed in details in the next section.

1.4.3 Parallel Algorithm

We distinguish two kinds of atomic independent tasks in DFROST: the first is related to solving an instance of a problem of type `Ali1D`, while the second is associated with solving an instance of an `Ali3D` problem[9].

Hence phase *D* consists in solving 1,200,000 independent tasks of type `Ali1D` and `Ali3D`, while phase *E* consists in solving several hundreds of independent tasks `Ali1D` and ten independent tasks `Ali3D`. The final decision requires sorting and analysis of the *N* best solutions of type `Ali1D` and the *N* best solutions of type `Ali3D`.

There is a couple of important observations to keep in mind in order to obtain an efficient parallel implementation for DFROST. The first is that the exact number of tasks is not known in advance. Second, which is even more important, the tasks are irregular (especially tasks of type `Ali3D`) with unpredictable and largely varying execution time. In addition, small tasks need to be aggregated in macro-tasks in order to reduce data broadcasting overhead. Since the complexity of the two types of tasks is different, the granularity for macro-tasks `Ali1D` should be different from the granularity for macro-tasks `Ali3D`.

The parallel algorithm that we propose is based on *centralized dynamic load balancing*: macro-tasks are dispatched from a centralized location (pool) in a dynamic way. The work pool is managed by a "master" who gives work *on demand* to idle "slaves". Each slave executes the macro-tasks assigned to it by solving sequentially the corresponding subproblems (either `Ali1D` or `Ali3D`). Note that dynamic load balancing is the only reasonable task-allocation method when dealing with irregular tasks for which the amount of work is not known prior to execution.

In phase *E* the pool contains initially several hundreds of tasks of type `Ali1D`. The master increases the work granularity by grouping `gran1D` of them in macro-tasks. These macro-tasks are distributed on demand to the slaves that solve the corresponding problems. The solutions computed in this way are sent back to the master and sorted by it locally. The templates associated to the *N* best scores yield *N* problems of type `Ali3D`. The master groups them in batches of size `gran3D` and transmits them to the slaves where the associated problems are solved. The granularity `gran1D` is bigger than the `gran3D` granularity. Finally the slaves send back to the master the computed solutions.

The strategy in phase *D* is simpler. The master only aggregates tasks in macro-tasks of size either `gran1D` or `gran3D`, sends them on demand to idle slaves (where the corresponding problems are sequentially solved), and finally gathers the distributions that have been computed. The master processes the library of templates in a sequential manner. First, it aims at distributing all tasks for a given template to the slaves. However, when the list of tasks for a given template becomes empty, but the granularity level is not attained, the master proceeds to distribute tasks from the next template. This strategy allows to reduce globally the idle time of the processors.

[9] In reality this problem can be further decomposed in subtasks. Although non independent, these subtasks can be executed in parallel as show in [11, 10]. This parallelization could be easily integrated in DFROST if necessary.

1.5 COMPUTATIONAL EXPERIMENTS

1.5.1 Running times

The numerical results presented in this section (see Table 1.1) were obtained on a cluster of 12 Intel(R) Xeon(TM) CPU 2.4 GHz, 2 Gb Ram, RedHat 9 Linux, connected by a 1 Gb Ethernet network. The behavior of DFROST was tested by entirely computing the phase D of the package, i.e. all the distributions for 1125 templates for both filters.

	Number of tasks	Wall clock time	Total sequential time	Speed-up
3D filter	1,104,074	3d 3h 20m	37d 5h 11m	11.9
1D filter	1,107,973	31m	4h 13m	8.2
Both filters	2,202,047	3d 3h 51m	37d 9h 24m	11.8

Table 1.1 Comparison of the total time (in days, hours, minutes) taken by a number of 1D and 3D tasks with the corresponding wall clock time after parallelizing the program

In the case of 3D filter, solving 1,104,074 alignments in parallel as shown on table 1.1 is very efficient. Comparison of the total sequential running times with the wall clock time of the master shows that we obtain a speed-up of about 12, i.e., an efficiency close to one. In the case of 1D filter, for solving 1,107,973 alignments, the speed up is lower but then the total sequential time is much shorter than for solving 3D tasks.

These significant results, obtained on such a large data set, justify the work done to distribute FROST and prove the efficiency of the proposed parallel algorithm.

Details from this execution are presented in table 1.2. The value of the parameters `gran1D` and `gran3D` were experimentally fixed to 1000 and 10 respectively.

We can calculate an upper limit for the number of processors beyond which it is not any more possible to benefit from adding more processors. The maximum time for an alignment is 797.4 seconds 1.3, this time is the lower limit of the wall clock time for the complete computation of the distributions for `Ali3d`. The total CPU time necessary to calculate all `Ali3D` alignments is 3,215,460 seconds. Thus, adding more than *4032 processors* (3215460/797.4) will not further accelerate the global process. This gives a theoretical upper limit. The assumption behind this procedure is that difficult computations are submitted first. This strategy was not implemented in the results presented in [38] since it requires a criterion for a preliminary running time task estimation. Our observation on the code behavior when computing all distributions confirm that a meaningful criterion is the solutions search space (see figure 1.10). Another criterion could be the observed in the past running time for a task.

1.5.2 Statistical analysis of the results

Using this parallel algorithm we were able to compute all distributions for the entire FROST templates library. This was never done with the sequential code, because of large templates like `1BGLA0` with sequences as long as 528 amino acids, leading to a number of possible alignments as large as 6.647E+77. Statistics concerning the running time distribution are presented on Figure 1.11.

On average, the running time distribution *of all* `Ali1D` tasks, is characterized by the following data:

minimum	1st quartile	mean	3rd quartile	maximum
0 s	0.03 s	0.58 s	2.32 s	797.4 s

Note that these times correspond to one alignment.

We observed that for 188 templates the computation of the distributions requires more than one hour CPU time. Statistical details concerning the running time of the four most time consuming templates are presented in table 1.3. Remember, that a PTP instance (i.e. when the query and the 3D structure are fixed) is considered as an atomic independent task in the current parallel strategy. Yet, as shown in [11, 10], such an instance could be further decomposed in subtasks that could be executed in parallel. We studied the need for implementing this parallelization in the package FROST. However, taking in account that: i) the number of independent tasks when computing distributions is very high; ii) the data from tables 1.2 and 1.3, as well as their statistical recapitulations in figure 1.11, clearly showing that really hard PTP instances are rather rare; iii) the speedup reported in section 1.5 is very satisfactory, we decided, for the time being, to stay with the current parallel strategy.

1.6 FUTURE RESEARCH DIRECTIONS

It is well known that large fractions of the proteins have a modular organization as shown on Figure 1.12. Such proteins are called *multi-domain proteins*. These modules can be detected at the level of the amino acid sequence as similar subsequences that are found in different protein sequences. In the 3D structure of the whole proteins these modules correspond, usually to one, sometimes to several, substructures called structural domains[10] [40] (see the right hand side of Figure 1.12).

Several cases can occur when studying such multi-domain proteins. Let us illustrate this point with the PEP-utilizers domain presented on Figure 1.12.

If one wishes to analyze the PEP-utilizers module family one needs to compare the corresponding sequences over their complete lengths. Using global alignment of the sequences (i.e. gaps before the beginning of a sequence and after its end are penalized) will not give a satisfactory result. If the goal of the study is to search for

[10]in the literature the terms domain and module are often used somewhat interchangeably. In this paper we restrict the use of module to subsequences and domain to 3D substructures

the PEP-utilizers module in a set of sequences (such as those shown in Fig. 1.12), one must use a semi-global alignment where the gaps before the beginning, and after the end of a sequence are set to zero. This allows the shorter sequence of the PEP-utilizers module to "slide" along the longer sequences until it finds the best match.

The most general case occurs when, comparing two sequences, for instance the second and the fifth in Fig. 1.12, one is trying to analyze what is common between these sequences. This corresponds to carrying out a local alignment, that is, finding subsequences in both sequences that have the maximum score when aligned (for a given score function).

The local alignment is the most general alignment technique. Accordingly, this is the convenient alignment when comparing an unknown sequence with a database of sequences, since it is unknown beforehand what the similarity is between the query and the database sequences.

Due to the strong analogy that exists between sequence-sequence alignment methods and sequence-structure alignment methods, the above considerations are also valid, *mutatis mutandis*, for protein threading methods.

In section 1.2.4 we mentioned that FROST permits *only* global alignment of a sequence with a core. Even more, to the best of our knowledge, no current protein threading approach exists, that uses non-local score functions for providing an exact solution, and that is able to carry out semi-global and local alignments. Some ideas to tackle this problem have been presented by G. Collet and al. in [42, 43] where mathematical formulations, based on MIP models for semi-global and local sequence/structure alignment, are discussed. The latest one is also called *flexible alignment* since it allows omissions of blocks during the alignment process (see Fig. 1.13).

Semi-global and flexible alignments raise a number of new questions. Performing such alignments necessitates the alignment of cores with potentially very long sequences (the largest proteins known are up to 10 000 residues long). The process of computing distributions (see 1.2.4) needs to be significantly modified in the context of arbitrarily long sequences. In addition, these types of alignment will drastically increase the solution space and the corresponding running time. In order to manage such an increase of the computational requirements the future semi-global and flexible alignment algorithms will need more and more parallel and distributed computing.

1.7 CONCLUSION

Fold recognition (protein threading) is rather typical of problems that occur in bioinformatics. It requires knowledge from different disciplines: biology for the definition of cores, physical-chemistry for the development of score functions, computer science for the conception of efficient alignment algorithms and statistics for explaining the significance of the alignment score.

Sequence comparison methods play an outstanding role for exploiting protein sequence data, in particular for *in silico* functional analysis. These methods are

versatile and extremely efficient as long as close homologs are considered. Fold recognition techniques are intended to replace them when the much more difficult case of remote homologs needs to be tackled. Unfortunately, fold recognition techniques are computer intensive and, for the moment, are less universal. In particular the problem of fold recognition has received a satisfactory solution only for the case of global alignments whereas, due to the protein modularity properties, semi-global and local alignments are urgently needed. Fold recognition methods are also plagued by the lack of a statistical theory permitting to assess the significance of alignment scores. Our goal, in the near future, is to set fold recognition methods on an equal footing with sequence alignment methods in terms of available types of alignment and assessment of the alignment score significance.

Of course, due to the inescapable NP-hard property of fold recognition alignment algorithms, these methods will always be more demanding in terms of computer resources than sequence alignments, although we are able to achieve pruning peak rate as high as 10^{74} per second for global alignments. However, as shown in this paper, it is possible to harness the power of grid computing to perform the heavy calculations that will be needed to analyze the 500 currently sequenced microbial genomes and the further thousand that are to be released next year.

REFERENCES

1. SF Altschul, TL Madden, AA Schaffer, J Zhang, Z Zhang, W Miller, and DJ Lipman. Gapped blast and psi-blast: a new generation of protein database search-programs. *Nucleic Acids Res*, 25:3389–402, 1997.

2. S.E. Brenner, C. Chothia, and T.J. Hubbard. Assessing sequence comparison methods with reliable structurally identified distant evolutionary relationships. *Proc Natl Acad Sci U S A*, 95:6073–6078, 1998.

3. C. Chothia. One thousand families for the molecular biologist. *Nature Biotechnology*, 22:1317–1321, 2004.

4. CA Orengo, DT Jones, and JM Thornton. Protein superfamilies and domain superfolds. *Nature*, 372:631–4, 1994.

5. F.M. Pearl, C.F. Bennett, J.E. Bray, A.P. Harrison, N. Martin, A. Shepherd, I. Sillitoe, J. Thornton, and C.A. Orengo. The cath database: an extended protein family resource for structural and functional genomics. *Nucleic Acids Research*, 31(1):452–455, 2003.

6. A. Andreeva, D. Howorth, S.E. Brenner, T.J.P. Hubbard, C. Chothia, and A.G. Murzin. Scop database in 2004: refinements integrate structure and sequence family data. *Nucleic Acids Research*, 32:226–229, 2004.

7. A. Marin, J.Pothier, K. Zimmermann, and J-F. Gibrat. Frost: A filter based fold recognition method. *Proteins*, 49(4):493–509, 2002.

8. R.H. Lathrop. The protein threading problem with sequence amino acid interaction preferences is NP-complete. *Protein Engineering*, 255:1059–1068, 1994.

9. R.H. Lathrop and T.F. Smith. Global optimum protein threading with gapped alignment and empirical pair potentials. *Journal of Molecular Biology*, 255:641–665, 1996.

10. N. Yanev and R. Andonov. Parallel divide&conquer approach for the protein threading problem. *Concurrency and Computation: Practice and Experience*, 16:961–974, 2004.

11. R. Andonov, S. Balev, and N. Yanev. Protein threading problem: From mathematical models to parallel implementations. *INFORMS Journal on Computing*, 16(4):393–405, 2004. Special Issue on Computational Molecular Biology/Bioinformatics, Eds. H. Greenberg, D. Gusfield, Y. Xu, W. Hart, M. Vingro.

12. J. Xu, M. Li, G. Lin, D. Kim, and Y. Xu. Raptor: optimal protein threading by linear programming. *Journal of Bioinformatics and Computational Biology*, 1(1):95–118, 2003.

13. Y. Xu and D. Xu. Protein threading using prospect: design and evaluation. *Proteins*, 40(3):343–354, 2000.

14. Stefan Balev. Solving the protein threading problem by lagrangian relaxation. In Jonassen and J. Kim, editors, *4th International Workshop on Algorithms in Bioinformatics, Bergen, Norway. Volume 3240 of LNCS/LNBI WABI 2004*, pages 182–193, 2004.

15. W.W. Li, R.W. Byrnes, J. Hayes, V.M. Reyes, A. Birnbaum, A. Shahab, C. Mosley, D. Pekurovsky, G.B. Quinn, I.N. Shindyalov, H. Casanova, L. Ang, F. Berman, M.A. Miller, and P.E. Bourne. The encyclopedia of life project: Grid software and deployment. *Special Issue on Grid Systems for Life Sciences. New Generation Computing*, 2003.

16. Thomas Steinke. Alignment & threading on massively parallel computers. Technical report, Berlin Center for Genome Based Bioinformatics, 2003.

17. Y. Xu, D. Xu, and E.C. Uberbacher. An efficient computational method for globally optimal threading. *Journal of Computational Biology*, 5:597–614, 1998.

18. SF Altschul. Amino acid substitution matrices from an information theoretic perspective. *J Mol Biol*, 219:555–65, 1991.

19. MO Dayhoff, RM Schwartz, and BC Orcutt. *Atlas of protein sequence and structure*, volume 5, chapter A model of evolutionary change in proteins, pages 345–352. National Biomedical Research Foundation, Washington DC, 1978.

20. S Henikoff and JG Henikoff. Amino acid substitution matrices from protein blocks. *Proc Natl Acad Sci U S A*, 89:10915–9, 1992.

21. SB Needleman and CD Wunsch. A general method applicable to the search for similarities in the aminoacid sequence of two proteins. *J Mol Biol*, 48, 1970.

22. TF Smith and MS Waterman. Identification of common molecular subsequences. *J Mol Biol*, 147:195–7, 1981.

23. WR Pearson. Flexible sequence similarity searching with the fasta3 program package. *Methods Mol Biol*, 132:185–219, 2000.

24. SE Brenner, C Chothia, and TJ Hubbard. Assessing sequence comparison methods with reliable structurallyidentified distant evolutionary relationships. *Proc Natl Acad Sci U S A*, 95:6073–8, 1998.

25. T Madej, JF Gibrat, and SH Bryant. Threading a database of protein cores. *Proteins*, 23:356–69, 1995.

26. T. Akutsu and S. Miyano. On the approximation of protein threading. *Theoretical Computer Science*, 210:261–275, 1999.

27. Nicola Yanev and Rumen Andonov. Solving the protein threading problem in parallel. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 157.1. IEEE Computer Society, 2003.

28. J. Xu, M. Li, G. Lin, D. Kim, and Y. Xu. Protein structure prediction by linear programming. In *Proceedings of The 7th Pacific Symposium on Biocomputing (PSB)*, pages 264–275, 2003.

29. G. Lancia. Integer programming models for computational biology problems. *J. Comput. Sci. & Technol.*, 19(1):60–77, 2004.

30. J. Blazewicz, P. Lukasiak, and M. Milostan. Some operations research methods for analyzing protein sequences and structures. *4OR A Quarterly Journal of Operations Research*, 4(2):91–123, 2006.

31. S Karlin and SF Altschul. Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proc Natl Acad Sci U S A*, 87:2264–8, 1990.

32. LA Mirny, AV Finkelstein, and EI Shakhnovich. Statistical significance of protein structure prediction by threading. *Proc Natl Acad Sci U S A*, 97:9978–83, 2000.

33. K. Zimmermann A. Marin, J.Pothier and J-F. Gibrat. *Protein structure prediction: bioinformatic approach*, chapter Protein threading statistics: an attempt to assess the significance of a fold assignment to a sequence. International University line, 2002.

34. J. Setubal and J. Meidanis. *Introduction to computational molecular biology*. PWS publishing company, 1997.

35. R.H. Lathrop, R.G. Rogers Jr., J. Bienkowska, B.K.M. Bryant, L.J. Buturovic, C. Gaitatzes, R. Nambudripad, J.V. White, and T.F. Smith. *Computational Methods in Molecular Biology*, chapter 12, pages 227–283. Elsevier Science, 1998.

36. N. Yanev, P. Veber, R. Andonov, and S. Balev. Lagrangian approaches for a class of matching problems in computational biology. Rapport de recherche RR-5973, INRIA, August 2006. to appear in Computers and Mathematics with Applications, special issue on Computational Biology, Edt.Roberto Tadei.

37. P. Veber, N. Yanev, R. Andonov, and V. Poirriez. Optimal protein threading by cost-splitting. In *WABI'05 (5th Workshop on Algorithms in Bioinformatics)*, volume 3692 of *Lecture Notes in Computer Science*, pages 365–375. Springer, 2005.

38. V. Poirriez, R. Andonov, A. Marin, and J-F. Gibrat. Frost: Revisited and distributed. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 7*, page 200.1, Washington, DC, USA, 2005. IEEE Computer Society.

39. Ross Ihaka and Robert Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

40. A.M.Lesk and G.D. Rose. Folding units in globular proteins. *PNAS*, 78:4304–4308, 1981.

41. A. Bateman, L. Coin, R. Durbin, R.D. Finn, V. Hollich, Jones Griffiths, A. Khanna, M. Marshall, S. Moxon, E.L. Sonnhammer, D.J. Studholme, C. Yeats, and S.R. Eddy. The Pfam protein families database. *Nucleic Acids Res*, 32:D138–41, 2004.

42. G. Collet, A. Marin, N. Yanev, R. Andonov, and J-F. Gibrat. Implementing a semi-global alignment algorithm for protein threading methods that use non-local score functions. In *Poster of the ROADEF conference*, 2006. in French.

43. G. Collet, N. Yanev, A. Marin, R. Andonov, and J-F. Gibrat. A flexible model for protein fold recognition. In A. Denise, P. Durrens, S. Robin, E. Rocha, A. de Daruvar, and A. Groppi, editors, *Septièmes Journes Ouvertes de Biologie, Informatique et Mathématiques (JOBIM)*, pages 215–216, 2006.

Template	DFROST	CPU tot	Cpu av	NAli
1BGLA0	15455	107569	113	945
1ALO_0	9565	96579	97	995
1CXSA0	5988	55808	58	960
1DIK_0	4506	46855	47	977
1BGW_0	4152	45286	45	1000
1CLC_0	3580	37973	39	969
1AA6_0	3357	35819	38	926
1DJXB0	3025	31276	31	1000
1DAR_0	2705	28671	28	1000
1AOZA0	2477	25156	26	935
1AK5_0	2072	22326	22	979
1AUIA0	2016	22010	22	1000
1AOFB0	2065	21619	21	1000
1BHGA0	1904	20740	21	980
1AORA0	1920	20059	20	995
1AYL_0	1807	18961	19	973
1EUT_0	1753	18883	18	995
1CTN_0	1535	16670	16	1000
1ECL_0	1439	15589	16	953
1ATIA0	1492	15463	15	980
1CIY_0	1441	15044	15	1000
1BYB_0	1307	13892	14	990
1COY_0	1204	13150	13	957
1DLC_0	1104	11825	13	907
1BDP_0	1173	12814	12	995
1AOP_0	1134	12323	12	1000
1AG8A0	1120	12153	12	990
1BMFC0	1094	11338	11	1000
1ECFB0	1052	11254	11	990
1DERA0	1047	11109	11	1000
1ALKA0	1022	10937	11	965
1DPE_0	988	10626	11	957
1DDT_0	973	10349	10	1000
1AC5_0	907	9877	9	1000
1CAE_0	913	9870	9	990
1BMFD0	914	9467	9	998
1DPGA0	875	9092	9	1000
1ASYA0	1102	8634	9	952
1LYLA0	782	8335	8	990
1BIF_0	657	7129	7	948
1AD3A0	629	6669	6	1000
1DNPA0	776	6580	6	960

Table 1.2 An extract from the execution times (in seconds) when computing the 3D score distributions. The templates for which the distributions are calculated are listed in the first column. The second column gives the parallel time (the execution time for the master) on a cluster of 12 processors. The third column shows the CPU sequential time (obtained by adding the CPU times from the slaves). The fourth column reports the average CPU time per alignment and the last column shows the actual number of sequences that have been threaded to calculate the distributions. The value of the granularity was fixed to 10.

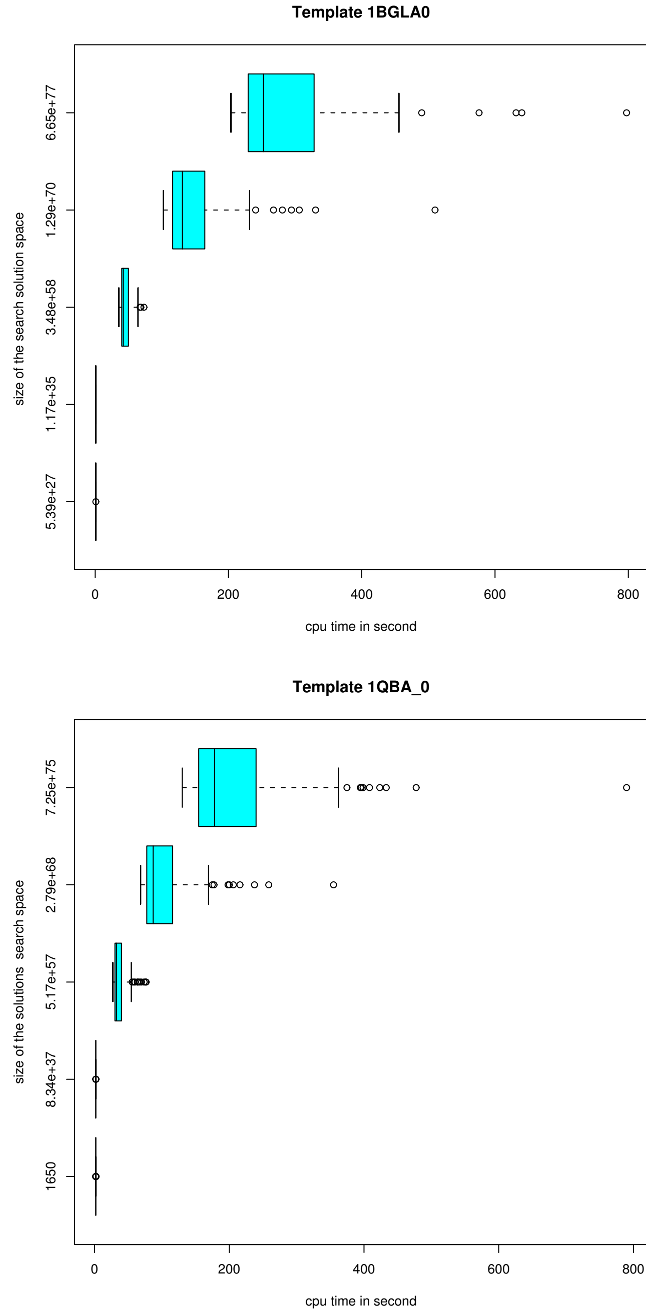


Fig. 1.11 Two templates with heavy distribution computations. 1BGLA0 and 1QBA_0, are selected from table 1.3 and the corresponding box-plots of the distributions running time are plotted using the statistical package **R** [39]. The left and right ends of a box correspond to the lower and upper quartiles and the middle line corresponds to the median of the distribution. Vertical lines, usually called “whiskers”, go left and right from the box to the extreme of the data (here defined as 1.5 times the inter-quartile range). Outliers are plotted individually. Note that the distribution is not symmetric and exhibits a heavy tail for longer CPU times.

	Nb Sol	NAli	Min	Q_1	Med	Mean	Q_3	Max
1BGLA0	$5.4\ 10^{27}$	55	0.95	0.96	0.98	0.97	0.98	1.02
	$1.2\ 10^{35}$	56	0.95	0.96	0.97	0.97	0.98	1.01
	$3.5\ 10^{58}$	192	35.6	39.9	42.2	45.2	50.0	73.2
	$1.3\ 10^{70}$	199	102.4	116.3	131.0	145.7	164.6	510.0
	$6.6\ 10^{77}$	150	203.8	229.7	252.6	291.7	327.5	797.4
1QBA_0	$1.6\ 10^{3}$	58	1.82	1.83	1.83	1.84	1.84	1.89
	$8.3\ 10^{37}$	57	1.82	1.83	1.83	1.84	1.84	1.89
	$5.2\ 10^{57}$	197	27.1	30.2	32.5	36.3	39.8	76.6
	$2.8\ 10^{68}$	200	68.4	77.5	86.9	101.4	116.0	354.8
	$7.2\ 10^{75}$	200	130.1	154.7	178.3	207.0	239.8	789.8
1ALO_0	$3.1\ 10^{33}$	57	0.85	0.87	0.87	0.87	0.88	0.89
	$6.0\ 10^{33}$	57	0.85	0.86	0.87	0.87	0.87	0.89
	$2.5\ 10^{57}$	190	25.8	29.3	36.1	40.8	46.7	135.2
	$1.6\ 10^{69}$	200	67.4	86.3	113.2	123.2	134.8	397.6
	$1.3\ 10^{77}$	200	139.9	175.7	231.0	262.2	303.4	735.0
1YGE_0	$3.4\ 10^{23}$	61	0.39	0.40	0.41	0.41	0.41	0.43
	$2.8\ 10^{45}$	59	0.40	0.41	0.41	0.41	0.42	0.42
	$2.1\ 10^{55}$	192	34.8	39.9	43.1	47.5	48.9	139.8
	$6.5\ 10^{61}$	173	71.2	80.5	89.5	102.0	115.9	365.1
	$4.4\ 10^{66}$	199	120.2	138.5	158.3	178.2	208.9	443.7

Table 1.3 **Sequential times in seconds for computing the 3D score distributions of four templates selected for their "difficulty" (search space size). For a given template the 5 rows represent alignment of sets of non related sequences having length respectively equal to: -30%, -15%, 0%, +15%, +30% of the template length. Nb Sol is the number of possible alignments that can be generated with the sequences and the template. This gives an indication of the difficulty of the problem to solve. NAli is the number of alignments (sequences) in the corresponding set. The last six columns report diverse running time characteristics obtained when aligning the set of sequences with the corresponding 3D structure: Min is the minimum value, Q_1 is the time at the 1st quartile position, Med. is the time at the median position, Mean is the average time, Q_3 is the time at the 3rd quartile position and Max is the maximum value.**

Fig. 1.12 Left panel: schematic representation of protein sequences with different modules (data from the PFAM database [41]). In the figure we focus on the three modules of the second sequence. These modules are also found in other sequences. Upper right panel: the structure of this sequence (a pyruvate phosphate dikinase) has been solved (PDB code 1dik) and the modules have been drawn in similar shades of gray in the 3D structure; lower right panel: zoom on the 3D structure of the second module. This module has 102 residues.

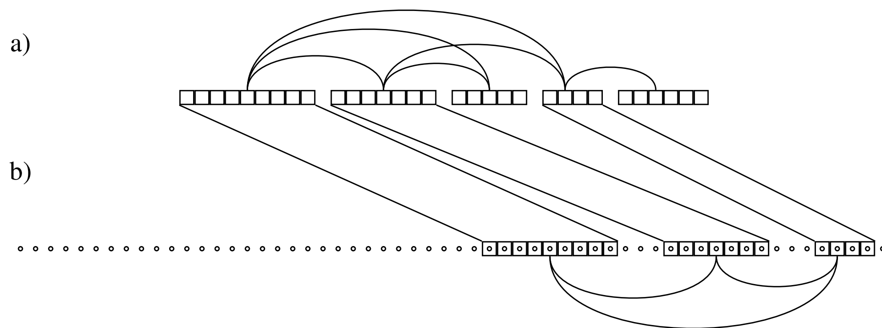


Fig. 1.13 Local alignment. a) A template containing five blocks. b) A sequence of 58 amino acids. On its right-hand site this sequence contains a structural domain which exhibits a good similarity to the template when only three blocks are aligned. To obtain this optimal alignment (i.e. giving the best score), two blocks have to be omitted.