University of California
Los Angeles

Optimal Multi-Way Number Partitioning

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Ethan L. Schreiber

2014

Abstract of the Dissertation

Optimal Multi-Way Number Partitioning

by

Ethan L. Schreiber

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2014

Professor Richard E. Korf, Chair

The NP-hard number-partitioning problem is to separate a multiset S of n positive integers into k subsets, such that the largest sum of the integers assigned to any subset is minimized. The classic application is scheduling a set of n jobs with different run times onto k identical machines such that the makespan, the time to complete the schedule, is minimized. The two-way number-partitioning decision problem is one of the original 21 problems Richard Karp proved NP-complete. It is also one of Garey and Johnson's six fundamental NP-complete problems, and the only one involving numbers.

This thesis explores algorithms for solving multi-way number-partitioning problems optimally. We explore previously existing algorithms as well as our own algorithms: sequential number partitioning (SNP), a branch-and-bound algorithm; binary-search improved bin completion (BSIBC), a bin-packing algorithm; cached iterative weakening (CIW), an iterative weakening algorithm; and a variant of CIW, low cardinality search (LCS). We show experimentally that for high precision random problem instances, SNP, CIW and LCS are all state of the art algorithms depending on the values of n and k. All three outperform previous algorithms by multiple orders of magnitude in terms of run time.

The dissertation of Ethan L. Schreiber is approved.

D. Stott Parker, Jr.

John W. Mamer

Adnan Darwiche

Richard E. Korf, Committee Chair

University of California, Los Angeles

2014

To my mother Linda,
who sparked my curiosity,
helped me explore it, and
taught me to never give up.

Table of Contents

List of Figures

List of Tables

Acknowledgments

Thank you to Richard Korf, my advisor, mentor and role model. He has supported me during my darkest hours when I thought I did not belong as a PhD student. He has guided me towards the topics discussed in this dissertation and helped me to develop the algorithms I present. More than anything, his door is always open and he has always had my best interests at heart. He is an amazing educator, focused on making everything as clear as possible, whether it is one-on-one, running a class or writing and editing a paper. He is a principled researcher, always pushing the boundaries of what is possible while adhering to the strictest ethics. I have become a better teacher, scholar and man while at UCLA, a great deal of this credit goes to Rich.

Thank you to Adnan Darwiche, my original advisor and member of my doctoral committee. When I applied for graduate school, I believed I wanted to work on Bayesian networks, and so my statement of purpose made it clear I wanted to work with Adnan. I was accepted to exactly one PhD program, and I suspect that Adnan had a lot to do with this acceptance, for which I will be forever indebted. Before I chose UCLA, the University of Toronto had put me on their wait list. The response deadline for UCLA was approaching so I emailed a professor at Toronto to ask if they had made a decision yet. He told me that they could not give me an answer yet but UCLA would be a good choice because, "Adnan is a great researcher and a very very nice guy, so it would be a great choice to work with him." While Bayesian network research did not work out for me, I agree with that professor's assesment.

Thank you to Stott Parker and John Mamer, the remaining members of my doctoral committee. While my interaction with these two professors was more scarce than Rich or Adnan, they have both provided support for me. Stott took the time to spend some time to get to know me in order to write a letter of recommendation for a scholarship. He also provided excellent feedback for my doctoral prospectus. When I was focusing on operations research, John took the time to sit with me and help point me in the right direction. The help was invaluable. I would also like to thank Dick Muntz, Judea Pearl, Carlo Zaniolo,

Adam Meyerson, Glenn Reinman, Tyson Condie, Glenn Reinman and David Smallberg. I have had the privilege of learning from, and teaching with these excellent UCLA professors.

Thank you to Gleb Belov who gave sent me the code for his branch-and-cut-and-price bin-packing solver. Beyond just giving me the code, he spent time helping me to get it working and also fixing bugs when I found them. His work was invaluable for my 2013 IJCAI paper. Thank you also to Ariel Felner, Rob Holte, Nathan Sturtevant and Jonathan Schaefer. Their work in heuristic search as well as feedback for my conference papers has been an important part of my graduate education.

My journey on the path to the PhD has crossed the paths of many other great professors. My computer science education started at Vassar College where I was put into a nurturing environment that cultivated my love of computer science. My undergraduate advisor Brad Richards was instrumental in me choosing computer science. His door was always open and he was willing to answer all of my questions. Chris Welty taught my first artificial intelligence class and was my first research mentor. Plus, he suffered through yearly Mets games with me after graduation for many years. Tom Ellman was very supportive when I first started considering graduate school. He helped me to understand what I would need to do to become a successful scholar. Paul Johnson was my favorite professor who did not teach computer science. You have had a profound effect on how I think about the world.

My masters advisor at Brown University, Thomas L. Dean was another wonderful mentor. His passion for his work on biologically plausible models of the brain was inspiring. He gave me the first insight into what a scholar is supposed to be. He demonstrated daily that science is about the love of understanding and not the pursuit of awards and acknowledgement. I also worked with Thomas Griffiths, an amazing researcher with whom I published my first academic paper. I do not understand how one man can be so prolific. I am in awe of the quality and quantity of the work he produces, I was lucky to play a small part.

As well as the support of professors, I have had tremendous support from friends and academic colleagues. Thank you to Joseph Barker, my office mate for many years at UCLA. In many ways, our paths through UCLA have been parallel and we have experienced the highs

and lows together. Joseph has been a tremendous help with research ideas and programming problems. Furthermore, he has been a wonderful friend, synonymous with my time at UCLA. Thank you also to my other UCLA office mates, colleagues and friends Alex Dow, Eric Huang, Teresa Breyer, Cesar Romero, Alex Fukunaga, Arthur Choi, Keith Cascio, Knot Pipatsrisawat, Michael Shindler, David Jurgens, Cataldo Schietroma, Tiansheng Yao, Suming Chen, Karthika Mohan, Neil Conos, Brandon Rothrock, Alex Shkapsky and Barzan Mozafari. You have all helped me along the path towards my degree. Thank you also to my friends and peers from Brown University, John Optimal Donaldson, Theresa Vu, Brendan Dickinson, Jay McCarthy, Jean Tsong, Matt Lease and Shrivaths R Iyenger. The beginning of my graduate education was among the happiest times of my life because of you.

Beyond my computer science education, I have been blessed with many wonderful friends who have supported, loved and inspired me. Thank you to my two best friends and brothers, Nicholas Garwolinski and Matthew D. Parker. To my oldest and closest friends Aron Trocchia, Jesse Ball, Sung Kim, Dana Fleur, Michelle Suh, Ben Sherman, Jesse Sokolovsky, Noriko Nagamoto, Jamie Sue Clark, Vanesa Sanchez and Amy Huang. You are all the family that I was not born with but have gained throughout life. You all have made me a better person.

To my closest family, Linda Schreiber, Kenneth Schreiber, Erica Schreiber, Elliott Lipitz, Elaine Lipitz, Alice Lindholm and Ron Lindholm. You have all been there with unquestionable love and support. You have inspired me to push myself and been there to make sure I actually did. You have given me everything, taught me everything, and most importantly given me the safe space that has allowed me to strive to push myself. I most certainly could not have completed this degree without you. Thank you to Kathie MacKenzie, who has also joined my family since I have moved to Los Angeles.

And finally, to Talia, my partner in everything, thank you. You have endured living with me throughout most of the days of my PhD. I fell in love with you on our third date during a moment we will forever remember. With time, our love has steadily grown. You are the strongest and most important support in my life, I am lucky to have found you.

Vita

1995-1999	B.A. (Computer Science), Vassar College
1999-2002	Software Developer, eMeta Corporation
2002-2003	Senior Software Developer, Condé Nast
2003-2005	Senior Consultant, Boyle Software
2005-2006	ScM (Computer Science), Brown University
2006-2007	Independent Consultant, Jane St. Capital
2008-2011, 2012-2013	Teaching Assistant, Department of Computer Science, University of California, Los Angeles
2011-2012, 2013-2014	Head Teaching Assistant, Department of Computer Science, University of California, Los Angeles

Publications

Ethan Schreiber and Thomas L. Griffiths. Subjective randomness and natural scene statistics. In Proceedings of the 29th Annual Conference of the Cognitive Science Society (COGSCI-07), pages 1449-1454, Nashville, Tennessee, USA, 2007.

Anne S. Hsu, Thomas L. Griffiths, and Ethan Schreiber. Subjective randomness and natural scene statistics. Psychonomic Bulletin & Review 17, no. 5, pages 624-629, 2010.

Ethan L. Schreiber and Richard E. Korf. Using Partitions and Superstrings for Lossless Compression of Pattern Databases. (Student Abstract) In Proceedings of the Twenty-Fifth Annual Conference on Artificial Intelligence (AAAI-11), San Francisco, CA, USA, 2011.

Ethan L. Schreiber and Richard E. Korf. Locality-preserving pattern databases. Technical Report 120010, Department of Computer Science, University of California, Los Angeles, 2012.

Richard E. Korf and Ethan L. Schreiber. Optimally scheduling small numbers of identical parallel machines. In Proceedings of the 23nd International Conference on Automated Planning and Scheduling (ICAPS-13), Rome, Italy, 2013.

Ethan L. Schreiber and Richard E. Korf. Improved bin completion for optimal bin packing and number partitioning. In Proceedings of the Twenty-Third international joint conference on Artificial Intelligence (IJCAI-13), pages. 651-658, Beijing, China, 2013.

Richard E. Korf, Ethan L. Schreiber, and Michael D. Moffitt. Optimal Sequential Multi-Way Number Partitioning. In Proceedings of the 13th International Symposium on Artificial Intelligence and Mathematics (ISAIM-14), Fort Lauderdale, Florida, USA, 2014.

Ethan L. Schreiber and Richard E. Korf. Cached Iterative Weakening for Optimal Multi-Way Number Partitioning. In Proceedings of the Twenty-Eighth Annual Conference on Artificial Intelligence (AAAI-14), pages. Quebec City, Canada, 2014.

CHAPTER 1

Introduction

1.0.1 Multi-Way Number Partitioning

Given an input multiset $S = \{s_1, s_2, ..., s_n\}$ of n positive integers, a number of subsets k, and a positive integer C, the multi-way number-partitioning decision problem is: Can the integers of S be separated into k subsets such that all subset sums are less than or equal to C? More formally, the requirements for all k subsets is that they:

$$\begin{aligned} \text{Have sums less than or equal to } C: &\quad \forall_i \ , \ \text{sum}(S_i) \leq C \\ \text{Are mutually exclusive:} &\quad \forall_{i,j} \ , \ S_i \cap S_j = \emptyset \\ \text{Are collectively exhaustive:} &\quad S_1 \cup S_2 \cup ... \cup S_k = S \end{aligned}$$

There is also an optimization version which either minimizes or maximizes one of three natural objective functions:

1. Minimize the largest subset sum.
2. Maximize the smallest subset sum.
3. Minimize the difference between the largest and smallest subset sums.

This thesis explores algorithms to solve the optimization version with objective function 1: Separate S into k mutually exclusive and collectively exhaustive subsets such that the largest subset sum is minimized. We choose objective function 1 as it corresponds to minimizing the total time required for a simple scheduling problem. (See 1.3.1.)

1.0.1.1 Perfect Partitions

Consider the input set $S = \{1, 2, 3, 4, 5, 6\}$ with $\text{sum}(S) = 21$ to be partitioned into $k = 3$ subsets. A perfect partition evenly distributes the total sum 21 among the three subsets with $\frac{\text{sum}(S)}{k} = \frac{21}{3} = 7$ in each of the three subsets. In this case, a perfect partition P exists: $P = \langle\{1, 6\}, \{2, 5\}, \{3, 4\}\rangle$. No partition with lower cost is possible since decreasing the sum in any of the subsets would increase the sum in another. If $\text{sum}(S)$ is not divisible by k, the ceiling is used since fractional subset sums are not possible. Formally, the cost of a perfect partition C_P^* is defined as:

$$C_P^* = \left\lceil \frac{\text{sum}(S)}{k} \right\rceil$$

All perfect partitions are optimal with no better partition being possible, though not all optimal partitions are perfect.

Example 1.0.1 - Multi-Way Partition

Consider the input set S=$\{8, 6, 5, 3, 2, 2, 1\}$ to be partitioned into $k = 3$ subsets. Here are two complete partitions A and B:

Partition A	S_1	S_2	S_3
Subsets:	{2, 8}	{1, 2, 5}	{3, 6}
Sums:	10	8	9
Cost:	10		

Partition B	S_1	S_2	S_3
Subsets:	{1, 8}	{2, 2, 5}	{3, 6}
Sums:	9	9	9
Cost:	9		

Partition A is complete since the integers of the subsets S_1, S_2 and S_3 are mutually exclusive and collectively exhaustive. (The integer 2 appears in both S_1 and S_2, but this is OK since it appears twice in the input multiset S.) Its cost is the value of the largest subset sum, $\text{sum}(S_1)$=10.

Partition B is also complete with cost $\text{sum}(S_1) = \text{sum}(S_2) = \text{sum}(S_3) = 9$. It is also a perfect partition and hence optimal since $C_P^* = \left\lceil \frac{\text{sum}(S)}{k} \right\rceil = \left\lceil \frac{27}{3} \right\rceil = 9$.

1.1 Related Problems

While multi-way number partitioning is the central topic of this thesis, two-way number partitioning, subset sum, and bin packing are closely related problems defined here. Algorithms for solving these problems will be employed in the algorithms used to solve multi-way number partitioning.

1.1.1 Two-Way Number Partitioning and Subset Sum

Given an input multiset $S = \{s_1, s_2, ..., s_n\}$ of n positive integers and a positive integer C, the two-way number-partitioning decision problem is to answer the question: Can the integers of S be separated into two mutually exclusive and collective exhaustive subsets such that the larger subset sum is less than or equal to C?

There is also an optimization version: Separate S into two mutually exclusive and collectively exhaustive subsets such that the larger subset sum is minimized. For the two-way problem, the three objective functions listed in section 1.0.1 are equivalent.

Given S and a target value T, the subset-sum problem is: Does there exist a subset of the integers of S with sum equal to T? Finding a perfect partition for the two-way number-partitioning problem is equivalent to the subset-sum problem with $T = \left\lceil \frac{\text{sum}(S)}{2} \right\rceil$.

Algorithms for solving two-way number partitioning and subset sum are used for multi-way number partitioning. These algorithms will be discussed in detail in chapter 2.

1.1.2 Bin Packing

Given an input multiset $S = \{s_1, s_2, ..., s_n\}$ of n positive integers representing item weights, a number of bins k, and a positive integer capacity C, the bin-packing problem is: Can the integers of S be separated into k bins of capacity C such that sum of the integers in each bin does not exceed C? There is also an optimization version: Assign the integers of S into the minimal number of bins such that the sum of the elements in each bin is less than or

equal to C. Bin packing uses the terminology "bins" and "capacity" which are analogs to the number-partitioning terms "subsets" and "subset sums". To be consistent, the rest of this thesis uses the terminology "subsets" and "subset sums" for both bin packing and number partitioning.

Bin packing is closely related to number partitioning. Both problems are given an input multiset of positive integers to be separated into mutually exclusive and collectively exhaustive subsets. Number partitioning fixes the number of subsets and minimizes their sums, while bin packing fixes the maximum subset sum and minimizes the number of subsets.

In fact, the decision versions of number partitioning and bin packing are identical, it is the optimization problems which differ. Because of the similarity between the two problems, bin-packing algorithms can be used to solved number-partitioning problems and number-partitioning algorithms can be used to solve bin-packing problems. Chapter 4 will discuss using bin-packing algorithms to solve number-partitioning problems.

1.2 Background

In 1971, Stephen Cook introduced the concept of NP-complete problems in his seminal paper, "The complexity of theorem-proving procedures" [Coo71]. An NP-complete problem has two main properties:

1. It must be in the class nondeterministic polynomial time (NP). A problem in NP must be a decision problem, meaning that the solution is either yes or no. Furthermore, if the answer is yes, the proof must be verifiable on a deterministic Turing machine [Tur36] within time polynomial in the size of the description of the problem.

2. Each of the other problems in NP can be reduced to it in polynomial time. Problem X' can be reduced to problem X if it is possible to construct a polynomial time algorithm that converts a problem instance of X' into a problem instance of X allowing an algorithm for X to solve the X' problem instance.

If a problem is not a decision problem, it does not satisfy property one and thus is not in the class NP-complete. The optimization version of multi-way number partitioning is not NP-complete for this reason. Any problem which satisfies property two that all NP problems can be reduced to it in polynomial time but does not satisfy property one is called NP-hard. The optimization version of multi-way number partitioning is NP-hard. Informally, all NP-hard problems are at least as hard as the hardest NP-complete problems.

While all yes solutions to NP-complete problems must be verifiable in polynomial time, it is unknown whether these solutions can be found in polynomial time. Most computer scientists believe that they cannot. However, if one NP-complete problem could be solved in polynomial time, then they all could be. The question of whether NP-complete problems can be solved in polynomial time, the P=NP problem, is the most important open problem in computer science [GJ79].

At the same time that Cook introduced NP-complete problems, both Cook and Leonid Levin independently proved that boolean satisfiability is NP-complete [Lev73]. In 1972, Richard Karp proved that 21 different problems are in the set of NP-complete problems in his paper, "Reducibility among combinatorial problems" [Kar72]. Since then, thousands of problems have been proved to be in the set of NP-complete problems [GJ79].

This thesis explores algorithms for optimal solutions to the *multi-way number partitioning problem*, sometimes also referred to as the *multiprocessor scheduling problem*. The two-way number-partitioning decision problem is one of the original 21 problems Richard Karp proved NP-complete [Kar72]. It is also one of Garey and Johnson's six fundamental NP-complete problems [GJ79], and the only one involving numbers.

Number partitioning is perhaps the easiest NP-complete problem to describe, there is a pseudopolynomial time algorithm for solving it, and there are classes of instances for which there are exponential numbers of perfect solutions. However, it is NP-complete, the pseudopolynomial algorithms are intractable for high precision input (and slow even for low precision), and there are many instances with an exponential search space and only one optimal solution.

As compared to other NP-complete problems, number partitioning has very little structure. Nonetheless, there have been continuous algorithmic improvements leading to multiple orders of magnitude speedup for solving this problem for over four decades [Gra66, JGJ78, DM95, Kor98, Kor11, Mof13, SK13, SK14]. This leads us to believe that similar gains should be possible for more highly structured NP-complete problems.

This thesis explores the history of algorithms for finding optimal solutions to hard number partitioning problems with high precision numbers and one (or very few) optimal solutions. It also presents the author's current state-of-the-art algorithm for finding optimal solutions to multi-way number partitioning problems. We show experimentally that this algorithm is asymptotically faster than the previous state of the art, achieving over two orders of magnitude faster performance for the experiments we ran.

1.3 Applications of Multi-Way Number Partitioning

1.3.1 Multiprocessor Scheduling

The multi-way number number-partitioning problem is synonymous with the multiprocessor scheduling problem [GJ79, Sar89, Pin12, DIM08, GLL79]. The input set S of n positive integers correspond to the run times of a set of n jobs. The number of subsets k corresponds to a number of identical machines such as processor cores that execute in parallel. The goal of multiprocessor scheduling is to assign each of the n jobs to one of the k machines while minimizing the makespan, or time to complete all jobs in the schedule. Minimizing the makespan is equivalent to minimizing the time to complete all jobs on the machine with the maximum load.

1.3.2 Voting Manipulation

Consider an election with more than two candidates where instead of voting for a candidate, voters veto one candidate, with each voter's veto carrying a different weight [Wal09]. The

candidate with the smallest total veto weight wins the election. How can a coalition of voters manipulate the election to maximize the chance that their preferred candidate wins?

The coalition's best strategy is to partition their vetoes among the candidates they wish to lose such that the sum of the veto weights of each of these candidates is larger than the veto weight of the candidate they wish to win. Multi-way number partitioning with the objective function of maximizing the minimum subset sum can be used to solve this problem.

1.3.3 Public Key Encryption

An early encryption method, the Merkle-Hellman knapsack cryptosystem [MH78] is based on solving subset-sum problems, which is similar to two-way number partitioning. While polynomial time algorithms for their most basic encryption methods have since been found [Sha83], this work was an early pioneer towards more powerful encryption systems such as RSA [ARS83].

1.3.4 Choosing Fair Teams

The classic schoolyard method of choosing fair sports teams is to assign one captain for each team and then have each captain pick players in round-robin fashion. Assume that each player is given a positive integer representing their skill level. Also, assume that the strength of a team is equal to the sum of the skill levels of its players. The greedy method of picking the best remaining player does not necessarily lead to fair teams depending on the relative skill of the players. An optimal multi-way partition could lead to more equitable teams than the greedy method. [Hay02]

1.4 Thesis Overview

This thesis follows the history of algorithms for solving multi-way number-partitioning problems. Chapter 2 discusses both approximate and exact algorithms for solving two-way num-

ber partitioning, a special case of multi-way number partitioning. The algorithms in this chapter form the fundamental basis for multi-way number-partitioning algorithms. All of these algorithms are previous work.

Chapter 3 covers branch-and-bound algorithms for solving the multi-way number partitioning. The chapter begins with lower bounds and upper bounds (approximation algorithms) for multi-way number partitioning. It then covers generating subsets with sums in a range. Finally, three exact algorithms are described: recursive number partitioning (RNP), Moffitt algorithm (MOF) and sequential number partitioning (SNP). RNP and MOF are previous works. SNP is one of our contributions, and is the state-of-the-art algorithm for multi-way number partitioning for small numbers of subsets.

Chapter 4 covers solving number-partitioning problems using bin-packing solvers. This chapter begins by discussing the relationship between number partitioning and bin packing and presents the algorithm MULTIFIT, for solving number partitioning problems with a bin-packing solver. It describes lower and upper bounds for bin packing. Finally, it covers three bin-packing algorithms. Bin completion (BC) and branch-and-cut-and-price (BCP) are previous works in artificial intelligence and operations research respectively. Improved bin completion (IBC) is our improvement to the bin completion (BC) algorithm making it more efficient for solving number-partitioning problems.

Chapter 5 describes cached iterative weakening (CIW), our state-of-the-art multi-way number-partitioning algorithm. It employs iterative weakening instead of branch-and-bound to solve number-partitioning problems. Branch-and-bound algorithms start with an approximate solution and then search to improve the solution until a final solution is found and proved optimal. In contrast, iterative weakening algorithms start by looking for a solution whose cost equals the lower bound. If this is not possible, it looks for the next best solution. This process continues iteratively until finding the first complete solution, which is an optimal solution.

Notation	Definition
S	The input multiset of positive integers, $S = \{s_1, s_2, ..., s_n\}$.
n	The cardinality of S.
k	The number of subsets in a partition.
P	A partition of S into k subsets, $P = \langle S_1, S_2, ..., S_k \rangle$.
P_d	A partial partition of S into d subsets, $P_d = \langle S_1, S_2, ..., S_d \rangle$, where $d < k$.
S_i	Subset i of the k subsets in a partition.
C^*	The cost of an optimal partition of the integers of S.
C_P^*	The cost of a perfect partition of the integers of S.
$\text{sum}(S)$	The sum of the integers of S: $\text{sum}(S) = \sum_{s_i \in S} s$.
$\min(S)$	The minimal integer in S.
S^R	The remaining integers in S after some have been removed in a recursive algorithm.
ub	The upper bound on subset sums in a partition, which is also the upper bound on partition cost.
lb	The lower bound on subset sums in a partition.

Table 1.1: An overview of notation used repeatedly throughout this thesis. Some notation which is localized to only one section is not listed here.

Part I

Two-Way Number Partitioning

CHAPTER 2

Two-Way Number Partitioning

2.1 Overview

Given an input multiset $S = \{s_1, s_2, ..., s_n\}$ of n positive integers and a positive integer C, the two-way number-partitioning decision problem is: Can the integers of S be separated into two mutually exclusive and collective exhaustive subsets such that the larger subset sum is less than or equal to C? More formally, can S be separated into S_1 and S_2 such that $\max(\text{sum}(S_1), \text{sum}(S_2)) \leq C$. This thesis deals with the optimization version: separate S into two subsets such that the larger subset sum is minimized. A partition in which the difference is within one is a perfect partition, which is the best possible partition, its cost C_P^* is:

$$C_P^* = \left\lceil \frac{\text{sum}(S)}{2} \right\rceil$$

The two-way partition problem is one of Karp's original 21 NP-complete problems [Kar72] and the optimization version is NP-hard. The problem has been called the "easiest hard problem" because despite being NP-complete, there is an easy-hard-easy transition (section 2.5.1) which makes many problem instances trivial to solve, even for large n [Mer06, Hay02]. Despite being NP-complete, there is a pseudo-polynomial time dynamic programming algorithm [GJ79] for solving the two-way partition problem. This means that there are solutions whose run time is polynomial in the numeric value of the input. This is in contrast to polynomial time which means that the run time is polynomial in the length of the input. The run time of the dynamic programming algorithm is polynomial in the size of S and the

magnitude of the input integers.

2.2 Subset-Sum Problem

A closely related problem to the two-way partition problem is the subset-sum problem: Is there a subset of S whose sum equals a target value T? Finding a perfect partition for the two-way number partitioning problem is equivalent to the subset sum problem with $T = \left\lceil \frac{\text{sum}(S)}{2} \right\rceil$. The subset sum problem is NP-complete [GJ79].

The subset-sum problem can be solved with a number-partitioning solver. Consider the subset-sum problem with input set S and target value T. Call $\text{sum}(S)$ the sum of the integers of S. We create input set S' by adding an integer x to S so that when S' is partitioned perfectly, the two subset sums equal T. We calculate the value of x as follows:

$$\begin{aligned} \frac{\text{sum}(S) + x}{2} &= T \\ \text{sum}(S) + x &= 2T \\ x &= 2T - \text{sum}(S) \end{aligned}$$

Given this value of x, the subset sum of the new input set S' is:

$$\begin{aligned} \text{sum}(S') &= \text{sum}(S) + x \\ &= \text{sum}(S) + 2T - \text{sum}(S) \\ &= 2T \end{aligned}$$

Therefore, the sum of each subset in a perfect two-way partition of S' is $2T/2 = T$. If a perfect partition of S' exists, one subset will contain x and one will not. The subset that does not contain x is a subset whose sum is T. If no perfect partition of S' exists, then there is no subset of S whose sum equals T.

This chapter introduces algorithms for solving the optimization version of the two-way

partition problem both approximately and optimally. All of the algorithms can also be used to solve the subset sum problem. Furthermore, given a lower bound lb and an upper bound ub, the algorithms can be extended to find all subsets with sums in the range $[lb, ub - 1]$. This will be important for solving the multi-way partition problem in chapters 3, 4 and 5.

2.3 Polynomial Time Approximation Algorithms (Upper Bounds)

The polynomial time approximation algorithms covered in this section provide an upper bound for the two-way partition problem. These bounds are used by exact algorithms to help prune the search space. This section introduces two approximation algorithms.

Example 2.3.1 - Solving a Subset-Sum Problem with a Two-Way Number-Partitioning Solver

Consider the subset-sum problem with input set $S = \{2, 8, 11, 12, 17, 18\}$ and target value $T = 41$. This problem can solved using a two-way number-partitioning solver as described in section 2.2:

- Create $x = 2T - \text{sum}(S) = 82 - 68 = 14$
- Create $S' = S \cup \{x\} = S \cup \{14\} = \{2, 8, 11, 12, 14, 17, 18\}$
- Optimally partition S' two-ways: $P = \langle\{11, 12, 18\}, \{2, 8, 14, 17\}\rangle$ with each partition having sum equal to $T = 41$.
- The subset {11,12,18} which does not contain x has sum equal to T.

2.3.1 Greedy Algorithm

Perhaps the most obvious algorithm for two-way number partitioning is the greedy algorithm [Gra66]. The greedy algorithm first sorts the input set S into decreasing order. [1] It then considers the integers one at a time and places them into the subset with the smaller sum so far, either S_1 or S_2. If $\text{sum}(S_1)$ equals $\text{sum}(S_2)$, one of the subsets is chosen arbitrarily. The algorithm runs in time $O(n \log n + n)$ and space $O(n)$. The partition values are within 4/3 of optimal [KPP04]. The greedy algorithm is optimal for $n \leq 4$ [Kor11].

Example 2.3.2 - Greedy Algorithm

Consider the input set $S = \{2, 8, 11, 12, 17, 18\}$. The following table shows the steps the greedy algorithm takes to compute an upper bound on the cost of a two-way partition.

Action	**S**	$\boldsymbol{S_1}$	$\boldsymbol{S_2}$
Sort S into decreasing order.	$\{18, 17, 12, 11, 8, 2\}$	{}	{}
Put 18 into S_1	$\{17, 12, 11, 8, 2\}$	{18}	{}
Put 17 into S_2	$\{12, 11, 8, 2\}$	{18}	{17}
Put 12 into S_2	$\{11, 8, 2\}$	{18}	{17, 12}
Put 11 into S_1	$\{8, 2\}$	{18, 11}	{17, 12}
Put 8 into S_1	$\{2\}$	{18, 11, 8}	{17, 12}
Put 2 into S_2	–	{18, 11, 8}	{17, 12, 2}

The upper bound is calculated as $\max\{\text{sum}(S_1), \text{sum}(S_2)\} = \max\{37, 31\} = 37$.

[1] When we say that a multiset is sorted into decreasing order, it is technically being sorted in non-increasing order since it is possible that two values are the same.

2.3.2 Set Differencing: The Karmarkar-Karp Algorithm (KK)

The Karmarkar-Karp (KK) set differencing algorithm [KK82] provides an alternative to greedy. Like greedy, KK begins by sorting S into decreasing order. Then, it iteratively replaces the largest two integers of S with their difference. This is equivalent to placing the two integers into separate subsets without specifying which integer goes into which subset. KK continues in this manner, replacing the two largest integers with their difference until there is one integer left, which is the difference between the sums of the sets, $|sum(S_1) - sum(S_2)|$. Given this difference, the upper bound is $\frac{\text{sum}(S)+|\text{sum}(S_1)-\text{sum}(S_2)|}{2}$.

In order to reconstruct the two subsets, KK keeps track of the actions taken while calculating the upper bound. The actions are of the form (Integer 1 - Integer2) $\rightarrow$ Difference. The right arrow is read as "is replaced by." Starting with the upper bound, these actions are performed in reverse to generate the partition. This is illustrated in example 2.3.3 on the next page.

Example 2.3.3 - Karmarkar Karp Set Differencing Algorithm

Consider the input set $S = \{2, 8, 11, 12, 17, 18\}$. Table (a) shows the steps KK takes to generate the upper bound. Starting with the input set, the two largest numbers are iteratively removed and replaced by their difference until one number is left. The action column states the two numbers removed at each step and the difference which replaces them. The new differences at each step are shown in bold.

Table (b) shows the steps KK takes to construct the partition corresponding to the upper bound value. The KK set difference value of 4 starts in subset S_1. The list of actions from table (a) are performed in reverse. For each action, the difference is removed from the subset with the larger number replacing the difference in the same set and the smaller number being placed in the other set. Note that the difference between S_1 and S_2 is equal to the KK set difference value of 4 at every step.

(a) Calculate Difference

Action	**S**
-	$\{18, 17, 12, 11, 8, 2\}$
$(18 - 17) \rightarrow 1$	$\{12, 11, 8, 2, \mathbf{1}\}$
$(12 - 11) \rightarrow 1$	$\{8, 2, \mathbf{1}, 1\}$
$(8 - 2) \rightarrow 6$	$\{\mathbf{6}, 1, 1\}$
$(6 - 1) \rightarrow 5$	$\{\mathbf{5}, 1\}$
$(5 - 1) \rightarrow 4$	$\{\mathbf{4}\}$

(b) Generate Partition

Reverse Action	$\boldsymbol{S_1}$	$\boldsymbol{S_2}$
-	$\{4\}$	$\{\}$
$4 \rightarrow (5 - 1)$	$\{\mathbf{5}\}$	$\{\mathbf{1}\}$
$5 \rightarrow (6 - 1)$	$\{\mathbf{6}\}$	$\{\mathbf{1}, 1\}$
$6 \rightarrow (8 - 2)$	$\{\mathbf{8}\}$	$\{\mathbf{2}, 1, 1\}$
$1 \rightarrow (12 - 11)$	$\{\mathbf{11}, 8\}$	$\{\mathbf{12}, 2, 1\}$
$1 \rightarrow (18 - 17)$	$\{\mathbf{17}, 11, 8\}$	$\{\mathbf{18}, 12, 2\}$

The upper bound is calculated as $\max\{\text{sum}(S_1), \text{sum}(S_2)\} = \max\{36, 32\} = 36$.

2.4 Optimal Algorithms

2.4.1 Complete Greedy Algorithm (CGA)

The complete greedy algorithm (CGA) algorithm [Kor98] transforms the greedy algorithm into an optimal algorithm. Like the greedy algorithm, the CGA first sorts S into decreasing order. It then proceeds to partition S into two subsets S_1 and S_2.

CGA searches a binary tree with each level corresponding to an integer in S. At each node, the left branch puts the integer into the subset with smaller sum (the greedy choice), and the right branch puts it into the other subset. CGA keeps track of the sums of both subsets at each node and returns the smallest maximum subset sum encountered at any leaf node in the tree. The binary tree is searched depth first visiting left children before right. CGA employs two pruning rules to reduce the size of the search tree:

1. If the two subset sums are equal at any node, the the next integer of S is only put into one of the subsets since the trees underneath that node would be identical. This applies to the root node as well where the sum of each subset is zero and so the difference is zero. Therefore, the root of the tree forces the largest integer of S into the first subset.

2. If the sum of unassigned integers remaining in S is less than the difference between the two subsets, it places all the remaining integers in the smaller subset.

Since CGA searches a binary tree, its worst-case time complexity is $O(2^n)$ where n is the number of integers in the input set S. Since it is a depth first search, the space complexity is $O(n)$, which is linear in the depth of the tree.

Example 2.4.1 - Complete Greedy Algorithm

Figure 2.1 shows the binary tree that CGA searches to find an optimal partition for the input set $S = \{18, 17, 12, 11, 8, 2\}$. At each node, the ordered pair in parentheses is the sum of each of the two subsets and the numbers in the curly braces are the remaining numbers from S yet to be assigned. If the sum of the remaining integers is less than the difference between the two subsets, the remaining integers are added to the smaller subset. The bold ordered pairs are complete partitions. The optimal partition has subset sums (35,33), corresponding to the subsets $\langle\{18, 17\}, \{12, 11, 8, 2\}\rangle$.

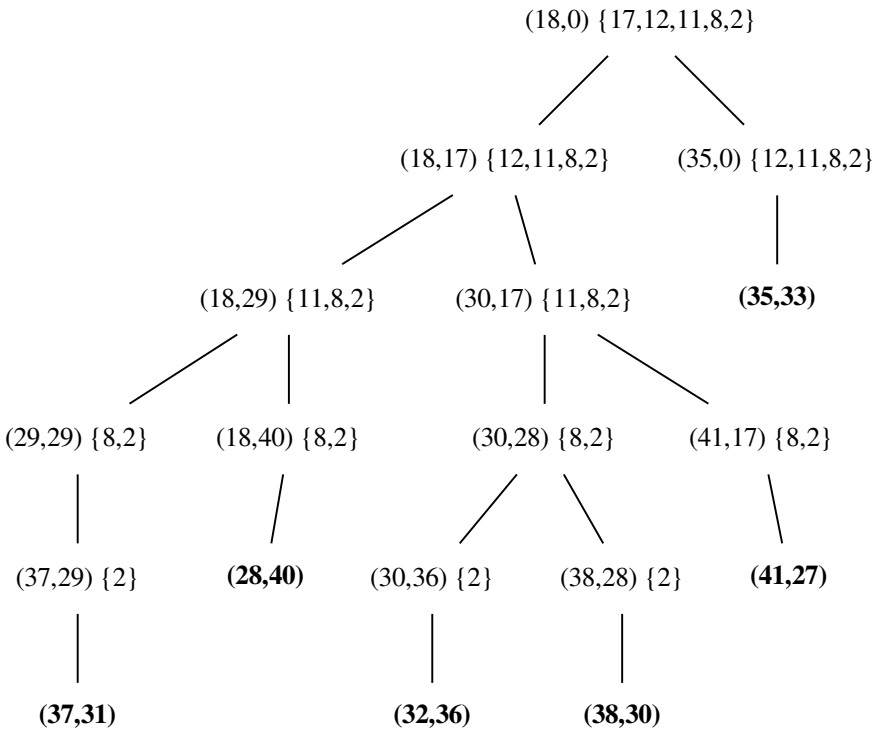


Figure 2.1: The CGA tree for the input set $S = \{18, 17, 12, 11, 8, 2\}$

2.4.2 Complete Karmarkar-Karp Set Differencing (CKK)

The complete complete Karmarkar-Karp (CKK) algorithm [Kor98] transforms the KK algorithm into an optimal algorithm. Like CGA, CKK first sorts S into decreasing order. It then proceeds to partition S into two subsets S_1 and S_2.

The KK algorithm places the two largest remaining integers into different subsets by replacing these numbers with their difference. The alternative is to place the numbers in the same subset by replacing them with their sum.

CKK searches a binary tree. At each node, the left branch puts the two largest remaining integers into different subsets by replacing them with their difference (the KK choice). The right branch puts the two largest remaining integers into the same subset by replacing them with their sum. The difference (or sum respectively) is inserted back into S in sorted order. This continues until there is one integer left, which is the difference between the two subset sums given the path to the leaf.

CKK keeps track of the leaf values and returns the smallest leaf value encountered. The binary tree is searched depth first visiting left children before right. When the largest remaining integer is larger than the sum of the rest, the largest integer is placed in one subset and the sum of the rest in the other. This is equivalent to pruning rule #2 from section 2.4.1.

Like CGA, since CKK performs a depth-first search on a binary tree, its worst-case time complexity is $O(2^n)$ while its space complexity is $O(n)$.

Example 2.4.2 - Complete Karmarkar Karp

Figure 2.2 shows the binary tree that CKK searches to find an optimal partition for the input set $S = \{18, 17, 12, 11, 8, 2\}$. At each node, all of the remaining integers are shown. The largest two integers are removed, the left branch inserts their difference, while the right branch inserts their sum. The newly inserted integer is shown in bold. The bold numbers at the leaves are the differences in sums between the two subsets. Like the CGA example 2.4.1, the optimal difference is 2, corresponding to the subsets $\langle\{35\}, \{12, 11, 8, 2\}\rangle$ with optimal cost 35.

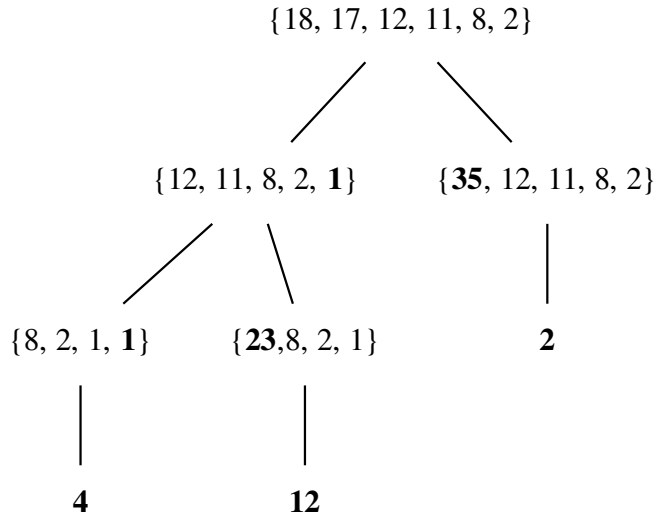


Figure 2.2: The CKK tree for the input set $S = \{18, 17, 12, 11, 8, 2\}$

2.4.3 Dynamic Programming (DP)

A dynamic programming (DP) algorithm [GJ79, MT90a, KS13] solves the partition problem in pseudo-polynomial time and space. Call sum(S) the sum of all integers of the input set S and min(S) the minimal integer of S. DP allocates a matrix M of bits with n rows and sum(S) columns. The first row corresponds to the largest integer in S, the second row the second largest, and so on. The columns correspond to sums in the range $[0, \text{sum}(S)]$. A bit at row r and column c is set to one if there is a subset of the largest r integers of S that sum to c.

DP begins by sorting S into decreasing order, setting all bits in M to zero except for the zero column which is set to one. It then sets the column corresponding to the largest integer of S in the first row to one. For each subsequent row r, it first copies all of the one-bits of the previous row, $r-1$ to the current row. Then, for each column c in the previous row that has a bit set to one, it sets column $c + s_r$ in the current row to one, where s_r is the integer in S corresponding to the current row. The one-bits in the final row correspond to all subset sums that can be formed from the input set S. The smallest subset sum greater than C_P^* (the perfect partition value) is the optimal value.

There are three optimizations to the basic DP algorithm. Since it is not possible to attain a better partition by adding to a value greater than the perfect sum, only columns corresponding to values less than or equal to C_P^* need to be stored. Furthermore, with the exception of the empty set with sum 0, it is impossible to get a value smaller than min(S). Therefore, DP needs only $C_P^* - min(S)$ columns corresponding to the range $[min(S), C_P^*]$. Column c corresponds to both a subset sum and its complement, sum(S) $- c$. Since the complement subsets are not stored in the matrix, DP keeps track of C^*, the value of the lowest cost partition found so far. The figure on the top of the following page lists the rules for scanning the row and updating the matrix and C^*.

For each new subset sum $c + s_r$ found, there are four possibilities:

- $\underline{\boldsymbol{c + s_r < C^*_P}}$: The bit corresponding to $c + s_r$ is set to 1 and C^* is set to $\min\{C^*, \text{sum}(S) - (c + s_r)\}$, the sum of this partition's complement set.
- $\underline{\boldsymbol{c + s_r == C^*_P}}$: DP returns immediately with the value $c + s_r$.
- $\underline{\boldsymbol{c + s_r \geq C^*}}$: $c + s_r$ is ignored.
- $\underline{\boldsymbol{C^*_P < c + s_r < C^*}}$: C^* is set to $c + s_r$.

Figure 2.3: The rules for filling the DP matrix.

Since one row is observed at a time, DP needs to store just one row. After populating an entire row, the second optimization foregoes copying row $r-1$ to row r, instead it continues working on the same row. In order to avoid adding the current input integer s_r to the set twice, the scan is done in reverse from the one-bit with the largest index to the one-bit with the smallest. When the scan is complete, the bit corresponding to column s_r is set to one.

The third optimization recognizes that only partitions with sum less than C^* can improve on the best partition so far. Therefore, when scanning row r, only columns c with $c+s_r < C^*$ are considered. Instead of beginning the reverse scan from the index with the largest index, the scan begins with the largest index with $c + s_r < C^*$.

The time and space complexity of DP is $O(n \cdot \text{sum}(S))$, the optimized version has space complexity $O(C^*_P$ - min(S) - 1) and time complexity $O(n \cdot [C^*_P$ - min(S) - 1]). DP is pseudo-polynomial since it is polynomial in the numeric values of the inputs, specifically sum(S).

S	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	C^*
18	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	50
17	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	35
12	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	1	1	-	-	-	-	-	-	-	-	-	-	1	1	-	-	-	-	35
11	-	-	-	-	-	-	-	-	-	1	1	-	-	-	-	1	1	-	-	-	-	1	-	-	-	-	1	1	1	-	-	-	-	35
8	-	-	-	-	-	-	1	-	-	1	1	-	-	-	-	1	1	1	1	-	-	1	-	1	1	-	1	1	1	1	-	-	-	35
2	1	-	-	-	-	-	-	-	1	1	1	1	1	-	-	1	1	1	1	1	1	1	-	1	1	1	1	1	1	1	1	1	-	35

Table 2.2: The DP matrix for the input set $S = \{18, 17, 12, 11, 8, 2\}$, (see example 2.4.3)

Example 2.4.3 - Dynamic Programming

Table 2.2 shows the matrix that DP generates to find an optimal partition for the input set $S = \{18, 17, 12, 11, 8, 2\}$. The sum of the integers of S is $\text{sum}(S) = 68$. There is one column for each integer between $min(S) = 2$ and $C_P^* = \left\lceil \frac{\text{sum}(S)}{2} \right\rceil = 34$. There is one row for each integer of S. These rows are shown for illustration; in practice only one row would be needed. The column labeled S shows the integer in S associated with each row. The column C^* shows the value of the lowest cost partition found so far after creating that row.

The algorithm starts by setting the bit in column 18 to 1 in the first ($S = 18$) row and C^* to $\text{sum}(S) - 18 = 50$. Then, this row is copied to the second ($S = 17$) row. The matrix is scanned and updated according to the rules in figure 2.3. The following table lists all of the bits that are set in each row given the value of S corresponding to the row summed with the appropriate columns c_{r-1} set in the previous row. The optimal partition $\langle\{18, 17\}, \{12, 11, 8, 2\}\rangle$ with partition cost $C^* = \max\{35, 33\} = 35$ is found in the second ($S = 17$) row, though the matrix needs to be completed to prove that no partition with cost $C_P^* = 34$ exists.

$\boldsymbol{S + c_{r-1} = value}$					
17+18=35	17+0=17				
12+18=30	12+17=29	12+ 0=12			
11+18=29	11+17=28	11+12=23	11+ 0=11		
8+23=31	8+18=26	8+17=25	8+12=20	8+11=19	
2+31=33	2+30=32	2+29=31	2+28=30	2+26=25	
2+25=27	2+23=25	2+20=22	2+19=21	2+18=20	
2+17=19	2+16=18	2+12=14	2+11=13	2+ 8=10	2+0=2

2.4.4 Horowitz and Sahni (HS)

The Horowitz and Sahni (HS) algorithm [HS74] uses memory to improve upon the 2^n time complexity of CKK and CGA. Call $\text{sum}(S)$ the sum of all integers of the input set S. The larger sum of a perfect partition is $\lceil \text{sum}(S)/2 \rceil$. HS begins by calculating an upper bound ub, in our case using the KK algorithm. The lower bound is calculated as $lb = \text{sum}(S) - ub$, the complement of ub. It then calculates the perfect partition value $C_P^* = \left\lceil \frac{\text{sum}(S)}{k} \right\rceil$.

HS sorts the input set S into decreasing order. It then divides S into two "half" sets S_A and S_B of size $n/2$ each, and generates the lists A and B of the sums of all $2^{\frac{n}{2}}$ subsets of each half set, including the empty set and the complete set. The sums are sorted, A increasing and B decreasing. It then iterates through each $a \in A$ and $b \in B$ starting with the first sums of A and B respectively.

If $a + b$ is less than the lower bound, HS gets the next a from A. If $a + b$ is between the lower bound and one minus the perfect value, the lower bound is set to $a + b$ and ub to its complement, $\text{sum}(S) - (a + b)$. Also, HS gets the next a from A. If $a + b$ is perfect or one less than perfect, the perfect value C_P^* is returned immediately since a perfect partition has been found. If $a + b$ is between perfect and the upper bound, the upper bound is set to $a + b$ and the lower bound to its complement $\text{sum}(S) - (a + b)$. Also, HS gets the next b from B. If $a + b$ is greater than the upper bound, HS gets the next b from B . This process continues until either the A or B lists are exhausted. Figure 2.4 also lists these rules for iterating through the A and B lists.

For two-way partitioning, there is an optimizing step [KS13]. The *largest* integer is removed and always assumed to be part of the current sum. *largest* is not included in the sums of A or B. For each $a \in A$ and $b \in B$, *sum* is calculated as $largest + a + b$. This cuts the size of either A or B in half.

Generating the A and B lists takes $O(2^{n/2})$ time since they are the power set of S_A and S_B which each contain $\frac{n}{2}$ integers. Sorting A and B takes $O(2^{n/2} \cdot \log 2^{n/2}) = O(2^{n/2} \cdot \frac{n}{2})$ time. The A and B lists are scanned in time linear in their size, $2^{\frac{n}{2}}$. Therefore, HS runs in

Condition	**Action**
$a + b \leq lb$	: Get next a from A.
$lb < a + b < C_P^* - 1$	: Set $lb = a + b$; $ub = \text{sum}(S) - (a + b)$. Get next b from B.
$C_P^* - 1 \leq a + b \leq C_P^*$	: Return C_P^*.
$C_P^* < a + b < ub$	: Set $ub = a + b$; $lb = \text{sum}(S) - (a + b)$. Get next b from B.
$a + b \geq upper$	: Get next b from B.

Figure 2.4: The rules for iterating using the HS algorithm.

time $O(\frac{n}{2} \cdot 2^{\frac{n}{2}} + 2 \cdot 2^{\frac{n}{2}}) = O(n \cdot 2^{\frac{n}{2}})$. Since A and B are of size $2^{\frac{n}{2}}$, HS also requires $O(2^{\frac{n}{2}})$ space, making it practical for up to about $n = 60$ integers.

In our implementation of HS, we have chosen to sort the input set S into decreasing order and choose the larger numbers to populate S_A and the smaller numbers to populate S_B. The original HS algorithm does not specify an order for the numbers. The run time of HS is approximately the same if the larger numbers populate S_A and the smaller numbers populate S_B.

Example 2.4.4 - Horowitz and Sahni

Consider the input set $S = \{225, \underbrace{216, 202, 148, 144, 121, 110}_{S_A}, \underbrace{102, 91, 82, 15, 13, 3}_{S_B}\}$. The following are the steps the Horowitz and Sahni (HS) algorithm takes to compute an optimal two-way partition.

1. Calculate $C_P^* = \left\lceil \frac{\text{sum}(S)}{2} \right\rceil = \left\lceil \frac{1472}{2} \right\rceil = 736$.
2. Store $largest = max(S) = 225$.
3. Calculate $best = 737$ using the KK approximation algorithm.
4. Calculate $lb = sum - (best - 1) = 736$.
5. Generate A as the sums of all subsets of S_A and B as the sums of all subsets of S_B. Sort A increasing and B decreasing:

A	**B**
0 , 110, 121, 144, 148, 202, 216, 231	306, 303, 293, 291, 290, 288, 278, 275,
254, 258, 265, 269, 292, 312, 323, 326,	224, 221, 215, 212, 211, 209, 208, 206,
337, 346, 350, 360, 364, 375, 379, 402,	204, 202, 201, 200, 199, 197, 196, 193,
413, 418, 433, 447, 456, 460, 467, 470,	191, 189, 188, 187, 186, 184, 176, 173,
471, 474, 481, 485, 494, 508, 523, 528,	133, 130, 122, 120, 119, 118, 117, 115,
539, 562, 566, 577, 581, 591, 595, 604,	113, 110, 109, 107, 106, 105, 104, 102,
615, 618, 629, 649, 672, 676, 683, 687,	100, 98, 97, 95, 94, 91, 85, 82,
710, 725, 739, 793, 797, 820, 831, 941	31 , 28, 18, 16, 15, 13, 3, 0

6. Starting with $a = A[0] = 0$, $b = B[0] = 306$ and $sum = a + b + largest = 0 + 306 + 225 = 531$, follow the steps of figure 2.4 to update a, b and sum while searching for the optimal partition cost:

Iter	a	b	sum	Iter	a	b	sum	Iter	a	b	sum
1	0	306	531	11	231	290	746	21	312	212	749
2	110	306	641	12	231	288	744	22	312	211	748
3	121	306	652	13	254	278	757	23	312	209	746
4	144	306	675	14	254	275	754	24	312	208	745
5	148	306	679	15	258	224	707	25	312	206	743
6	202	306	733	16	265	224	714	26	312	204	741
7	216	306	747	17	269	224	718	27	312	202	739
8	216	303	744	18	292	224	741	28	312	201	738
9	231	293	749	19	292	221	738	29	312	200	737
10	231	291	747	20	312	215	752	30	312	199	736

7. The algorithm stops at iteration 30 since $312 + 199 = 736 = C_P^*$

2.4.5 Schroeppel and Shamir (SS)

The Schroeppel and Shamir (SS) algorithm [SS81] is based on HS, but uses less memory. HS generates the entire A and B lists and sorts them in memory before scanning them. In contrast, SS generates the subsets of A and B on demand in the same order as HS.

SS divides S into four "quarter" sets $S_{A_1}, S_{A_2}, S_{B_1}, S_{B_2}$ of size $n/4$ each. It generates the lists A_1, A_2, B_1 and B_2 of all $2^{\frac{n}{4}}$ subsets of each quarter set sorted by their subset sums in increasing order. The subsets from the A_1 and A_2 lists are combined in a min heap to generate subsets in the same order as the list A in HS. Each subset of the heap consists of one subset from each of the A_1 and A_2 lists. Initially, it contains all pairs combining the empty set from the A_1 list with each subset from the A_2 list. The top of the heap contains the pair with smallest subset sum. Whenever a pair $(A_1[i], A_2[j])$ is popped off the top of the heap, it is replaced in the heap by a new pair $(A_1[i+1], A_2[j])$. Similarly, the subsets from the B_1 and B_2 lists are combined in a max heap, which generates subsets in the same

order as the B list from HS. SS uses these heaps to generate the subset sums in sorted order, then scans them in the same manner as the HS algorithm.

Generating the A_1, A_2, B_1 and B_2 lists takes $O(2^{n/4})$ time since they are the power sets of $S_{A_1}, S_{A_2}, S_{B_1}$ and S_{B_2} which each contain $\frac{n}{4}$ integers. Sorting A_1, A_2, B_1 and B_2 takes $O(2^{n/4} \cdot \log 2^{n/4}) = O(2^{n/4} \cdot \frac{n}{4})$ time each. Scanning the lists will generate the same subsets as HS. In the worst case, there are $2 \cdot 2^{\frac{n}{2}}$ of these subsets. However, in order to generate each of these subsets, a pop from and push into a heap of size $2^{\frac{n}{4}}$ is required. This takes $\log s^{\frac{n}{4}} = \frac{n}{4}$ time. Therefore, the scanning operation has time complexity $O(2^{\frac{n}{2}} \cdot \frac{n}{4})$. The overall time complexity is $O(2^{\frac{n}{4}} \cdot \frac{n}{4} + 2 \cdot 2^{\frac{n}{2}} \cdot \frac{n}{4}) = O(n \cdot 2^{\frac{n}{2}})$, the same time complexity as HS. However, SS only requires $O(2^{\frac{n}{4}})$ space for the four quarter sets and heaps, making it practical for up to about $n = 120$ integers.

Both HS and SS have time complexity $O(n \cdot 2^{\frac{n}{2}})$. However, both Horowitz and Sahni in [HS74] and Schroeppel and Shamir in [SS81] claim incorrectly that their algorithms have time $O(2^{\frac{n}{2}})$, neglecting the factor of n.

In our implementation of SS, we have chosen to sort the input set S into decreasing order and choose the larger numbers to populate A_1 and A_2 while the smaller numbers populate B_1 and B_2. This is about twice as fast as using the smaller numbers for A_1 and A_2 and the larger numbers for B_1 and B_2. Further research is needed to better understand this phenomena.

Example 2.4.5 - Schroeppel and Shamir

Consider the input set $S = \{225, \underbrace{216, 202, 148}_{S_{A_0}}, \underbrace{144, 121, 110}_{S_{A_1}}, \underbrace{102, 91, 82}_{S_{B_0}}, \underbrace{15, 13, 3}_{S_{B_1}}\}$. The Schroeppel and Shamir (SS) algorithm is very similar to HS. All of the steps are exactly the same as the steps in example 2.4.4 except for step 5:

5. Generate A_0, A_1, B_0 and B_1 as the sums of all subsets of S_{A_0}, S_{A_1}, S_{B_0} and S_{B_1} respectively. Sort A_0 and A_1 in increasing order; and B_0 and B_1 in decreasing

order as follows:

A_0	A_1	B_0	B_1
0, 148, 202, 216,	0, 110, 121, 144,	275, 193, 184, 173	31, 28, 18, 16,
350, 364, 418, 566	231, 254, 265, 375	102, 91, 82, 0	15, 13, 3, 0

A min heap is used to generate A from example 2.4.4 and a max heap for B. The min heap is initialized with pairs including all integers from A_0 paired with the smallest integer from A_1. Similarly, the max heap is initialized with pairs including all integers from B_0 paired with the largest integer from B_1. The pairs are sorted by their sums as follows:

Min Heap: (0, 0), (148, 0), (202, 0), (216, 0),
(350, 0), (364, 0), (418, 0), (566, 0)

Max Heap: (275, 31), (193, 31), (184, 31), (173, 31),
(102, 31), (91, 31), (82, 31), (0, 31)

To generate the next sum a of A, pop the min pair (0, 0) from the min heap. The sum of the pair 0 is the new value for a. Then, replace the pair with the same value from A_0 and the next value from A_1, in this case (0, 110).

Similarly, to generate the next sum b of B, pop the max pair (275, 31) from the max heap. The sum of the pair 306 is the new value for b. Then, replace the pair with the same value from B_0 and the next value from B_1, in this case (275, 28).

This process continues and generates the same sums in the same exact order as A and B from example 2.4.4.

2.5 Experimental Results

This section presents a series of experiments we ran to compare the two-way partitioning algorithms described in this chapter. Depending on the precision of the input integers and the number of input integers n, different algorithms are preferable. The precision of the input integers is measured in the number of bits b needed to represent them.

For each combination of b and n, we generated 100 problem instances. Each instance consists of n integers sampled uniformly at random within the range $[1, 2^b - 1]$. All experiments were run on an Intel Xeon X5680 CPU running at 3.33GHz.

2.5.1 Easy-Hard-Easy Transition for 32-Bit Instances

Every two-way number-partitioning problem has 2^n complete partitions. If the integers are sampled from the range $[1, 2^b - 1]$ where b is the number of bits used to represent an input integer, there are $n \times (2^b - 1)$ possible subset sums. For fixed b, the number of complete partitions grows exponentially with n while the number of possible unique subset sums grows linearly. If b is small, there will be many more complete partitions than unique sums possible for each of the subsets of the partitions, thereby making the chances of finding a perfect partition very high.

If a perfect partition is found, any partitioning algorithm can immediately return it as optimal. When perfect partitions exist, it is the ratio of perfect partitions to complete partitions which determine the difficulty of a problem instance. Two-way partitioning has an easy-hard-easy transition for a fixed precision. When no perfect partitions exist, as n increases, the problems tend to get more difficult. However, at some n, perfect partitions start to appear and the problems eventually get easier again as the number of perfect partitions grow exponentially. Both Korf [Kor98] and Mertens [Mer06] recognized this phenomena.

We ran experiments with integers sampled uniformly at random from the range $[1, 2^{32} - 1]$. We generated 100 problem instances for each n from $n = 20$ to $n = 200$ and report the average run times for each value of n. We chose to generate 32-bit numbers in order to show the

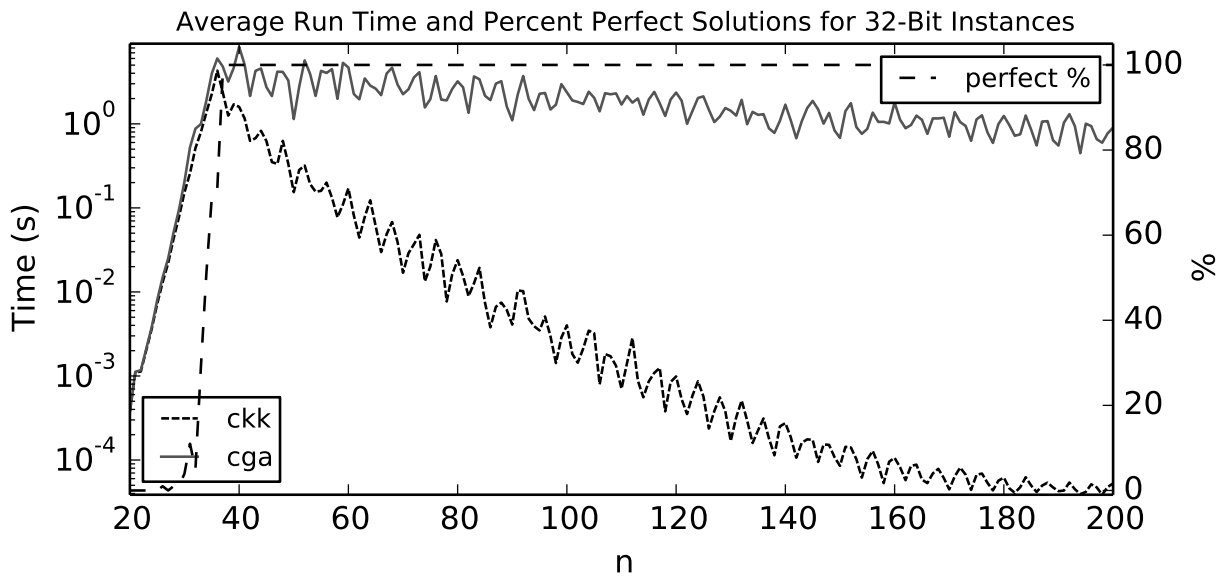


Figure 2.5: The average run times of CGA and CKK as well as the percent of perfect partitions for 32-bit partition instances.

easy-hard-easy transition of two-way number partitioning.

Figure 2.5 shows both the average run time of CGA and CKK against this data set (left axis) and the percentage of the 100 problem instances for each n whose optimal partition is perfect (right axis). For n from 20 to 29, none of the partitions are perfect. For $n = 38$ and above, all of the optimal partitions are perfect. Between $n = 30$ and $n = 37$, some of the optimal partitions are perfect and some are not.

The problems become harder for CGA and CKK as n increases from $n = 20$ to approximately $n = 40$. Then, both algorithms solve problems faster on average as n increases. CKK's performance improves more rapidly since it tends to find optimal solutions more quickly when many exist.

2.5.2 48-bit Experiments

We first ran experiments with integers sampled uniformly at random from the range $[1, 2^{48} - 1]$. We generated 100 problem instances for each n from $n = 20$ to $n = 70$ and report the average run times for each value of n. We chose to generate 48-bit numbers in order to

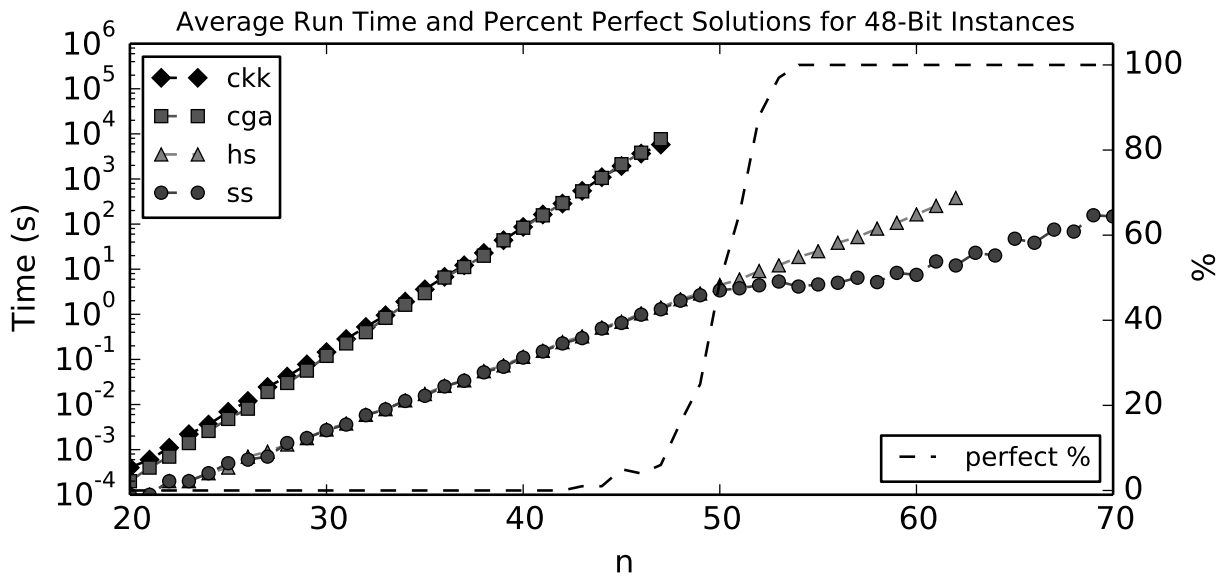


Figure 2.6: The average run times of CGA, CKK, HS and SS for solving 48-bit partition instances.

generate hard instances without perfect partitions [Kor11].

We ran complete Karmarkar-Karp (CKK), the complete greedy algorithm (CGA), Horowitz and Sahni (HS) and Schroeppel and Shamir (SS) against the benchmark set described above. For 48-bit numbers and n from 20 to 70, all four algorithms have the property that problem instances take longer to solve on average as n increases. Eventually, as n gets large enough, CKK and CGA would get easier again since there would be so many perfect partitions. However, HS and SS would run out of memory before getting to this point.

Figure 2.6 shows the timing results for the four algorithms. All four algorithms have an exponential explosion in run-time. Both CKK and CGA, with time complexity in $O(2^n)$, follow a very similar curve, though CGA is very slightly faster. Both HS and SS, with time complexity in $O(2^{\frac{n}{2}})$, are significantly faster than CKK and CGA.

Interestingly, even though HS and SS are $O(2^{\frac{n}{2}})$ algorithms, SS is an order of magnitude faster than HS for $n > 55$. For large n, all of the optimal partitions are perfect as can be seen by the dashed line in figure 2.6. Before beginning the search, HS must generate complete sets of size $2^{\frac{n}{2}}$ while SS generates complete sets of size only $2^{\frac{n}{4}}$. As n increases, it becomes easier

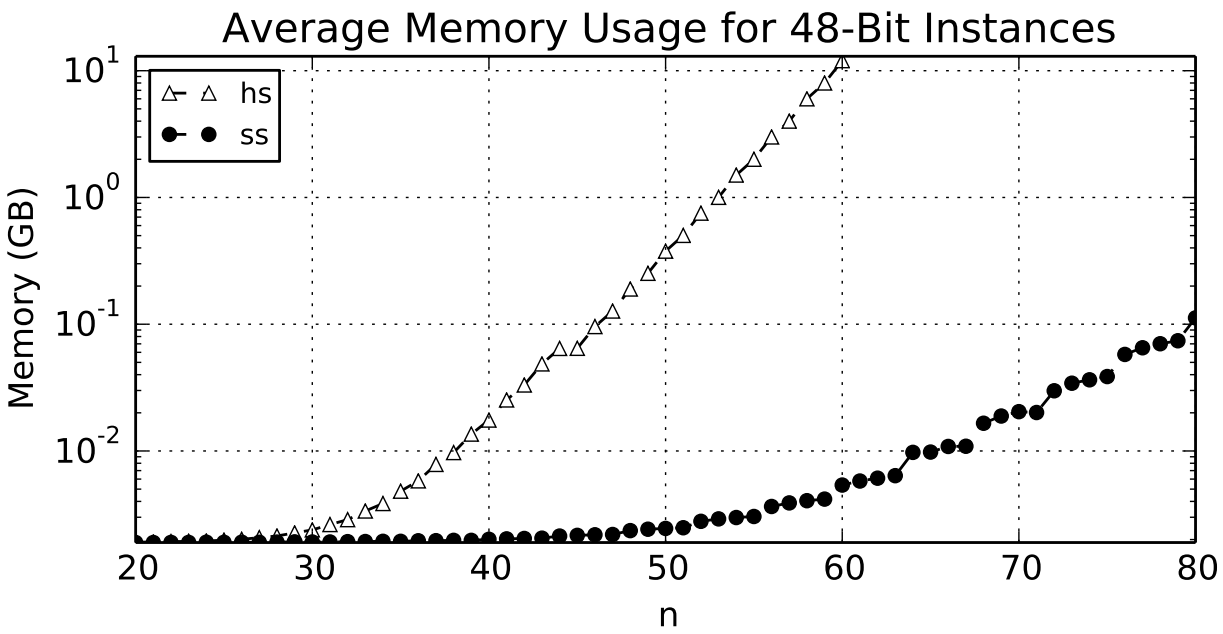


Figure 2.7: The memory usage of HS compared to SS for solving 48-bit partition instances.

and easier to find perfect partitions and the time to generate the complete sets dominates the total run time of HS and SS. This initial overhead explains the difference between the average run times of the two algorithms.

Both CKK and CGA use memory linear in n. As such, memory usage is not an issue for these algorithms for any n we are attempting to solve. HS and SS use memory exponential in n, specifically HS uses $O(2^{\frac{n}{2}})$ memory and SS uses $O(2^{\frac{n}{4}})$ memory. Figure 2.7 compares the memory usage between SS and HS for the 48-bit problem instances with n from 20 to 70. While HS maxes out our memory by $n = 60$, SS can solve problems of size $n = 70$ with less than 250 MB of memory.

2.5.3 Dynamic Programming Results on 16-Bit Instances

Section 2.4.3 covers the pseudo-polynomial time dynamic programming (DP) algorithm for solving two-way number-partitioning problems. The time and space complexity is in $O(n \cdot \text{sum}(S))$. Given this complexity class, 48-bit numbers are way too large to fit in memory as 2^{48} bytes is over 281 terabytes. In order to test DP, we ran experiments with integers

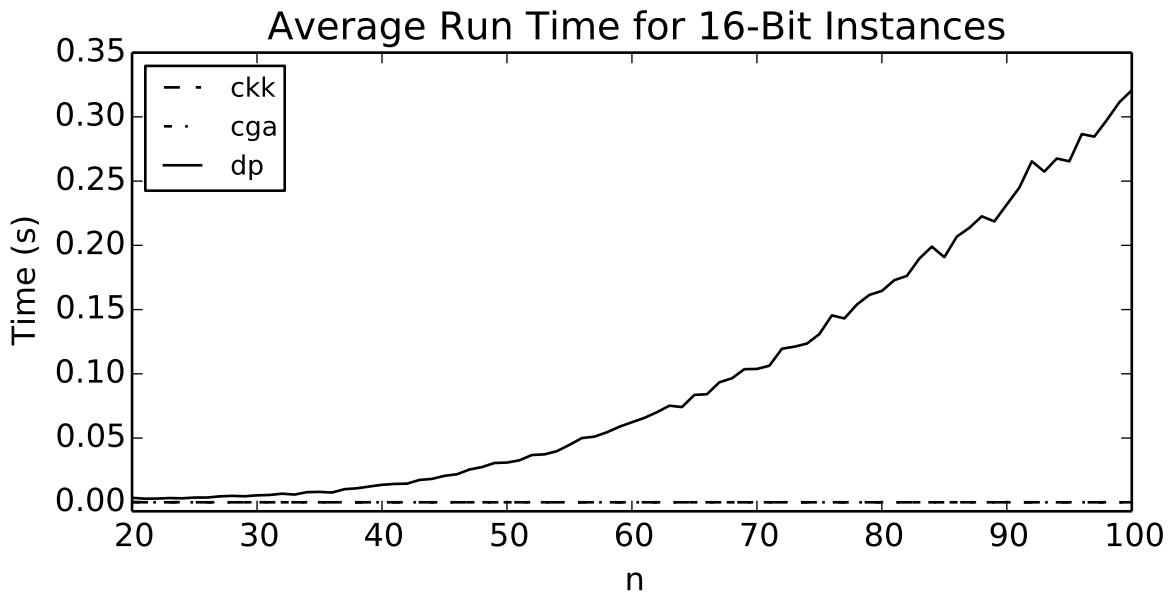


Figure 2.8: The average run times of CGA, CKK and DP for solving 16-bit partition instances.

sampled uniformly at random from the range $[1, 2^{16}-1]$. We generated 100 problem instances for each n from $n = 20$ to $n = 100$. The 16-bit input is tractable for DP.

Given the low precision of the input integers, almost all of the problems instances have perfect partitions. For all but $n = 20$, all of the 100 problem instances generated have perfect partitions. For $n = 20$, 80% of the problem instances have perfect partitions.

We ran CKK, CGA, and DP against the benchmark set described above. Figure 2.8 shows the timing results for the three algorithms. For 16-bit numbers and n from 20 to 100, both CKK and CGA solve all of the problem instances almost instantaneously regardless of the value of n. In contrast, dynamic programming gets slower as n gets larger.

The perceived wisdom is that given enough memory, DP, a pseudo-polynomial time algorithm, should be the fastest. CGA and CKK both run in time $O(2^n)$ while DP runs in time $O(n \times \text{sum}(S))$. Nonetheless, for this problem set in which n is in the range [20,100] and $\text{sum}(S) \leq 100 \times 2^{16}$, CGA and CKK dominate DP. This is because CKK and CKK find perfect partitions much faster than DP [KS13].

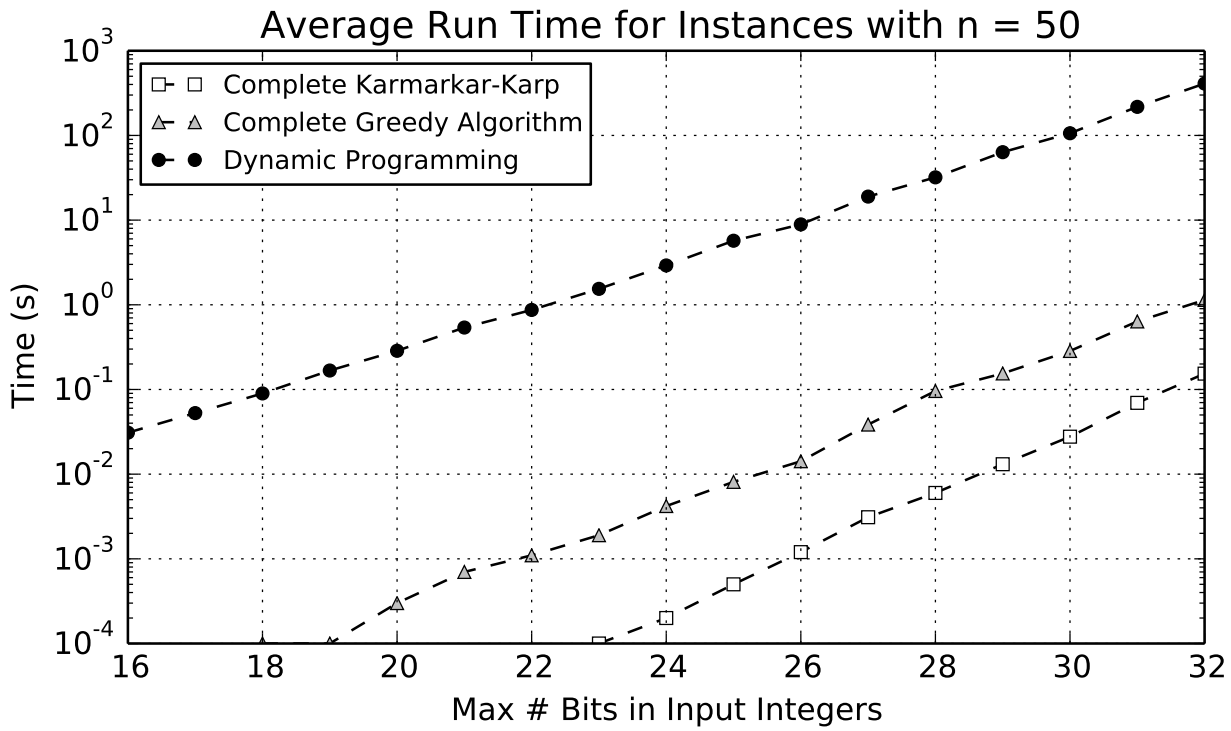


Figure 2.9: The average run times of CGA, CKK and DP for solving 16-bit partition instances.

2.5.4 Dynamic Programming Results Varying Precision

To further show that dynamic programming is dominated by CKK, we ran a set of experiments fixing $n = 50$ and varying the maximum number of bits needed to represent the input integers. The input integers were sampled from the range $[1, 2^b - 1]$ where b takes on the value of all integers from 16 to 32. For each value of b, 100 experiments were run.

Figure 2.9 shows the average run times of CKK, CGA and DP as b ranges from 16 to 32. For all of these relatively low precision input sets, CKK and CGA solve all instances almost instantaneously. DP is approximately three orders of magnitude slower than CGA and four orders of magnitude slower than CKK.

Figure 2.10 shows the memory usage of DP as b ranges from 16 to 32. As b increases, the memory usage of DP grows exponentially. DP is polynomial in $\text{sum}(S)$. Since $\text{sum}(S)$ grows exponentially with b, the memory usage of DP also grows exponentially. In contrast,

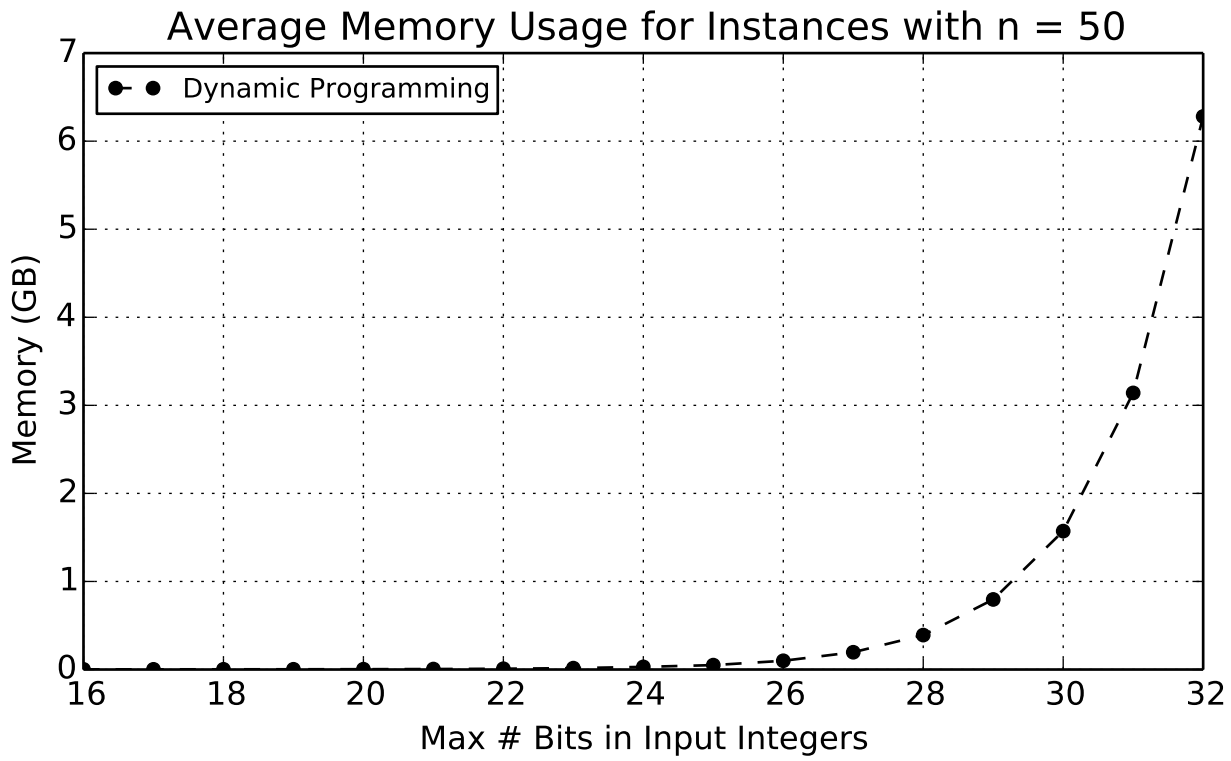


Figure 2.10: The memory usage of dynamic programming for solving instances with $n = 50$ as the precision of the input integers is varied.

both CGA and CKK use memory linear in n, the number of integers in the input set S. The memory usage of these two algorithms is negligible for $n = 50$, regardless of the precision of the input set.

Given that DP is slower than CKK and CGA for 16-bit integers, is intractable for 48-bit integers, and is dominanted for all precision datasets from 2^{16} to 2^{32}, it seems that it is not a useful algorithm for solving two-way number partitioning.

2.6 Summary

This chapter has introduced algorithms for solving the two-way number partitioning problem. The greedy algorithm and the Karmarkar-Karp (KK) heuristic generate approximate solutions. The rest of the algorithms we covered generate optimal solutions. CGA and CKK

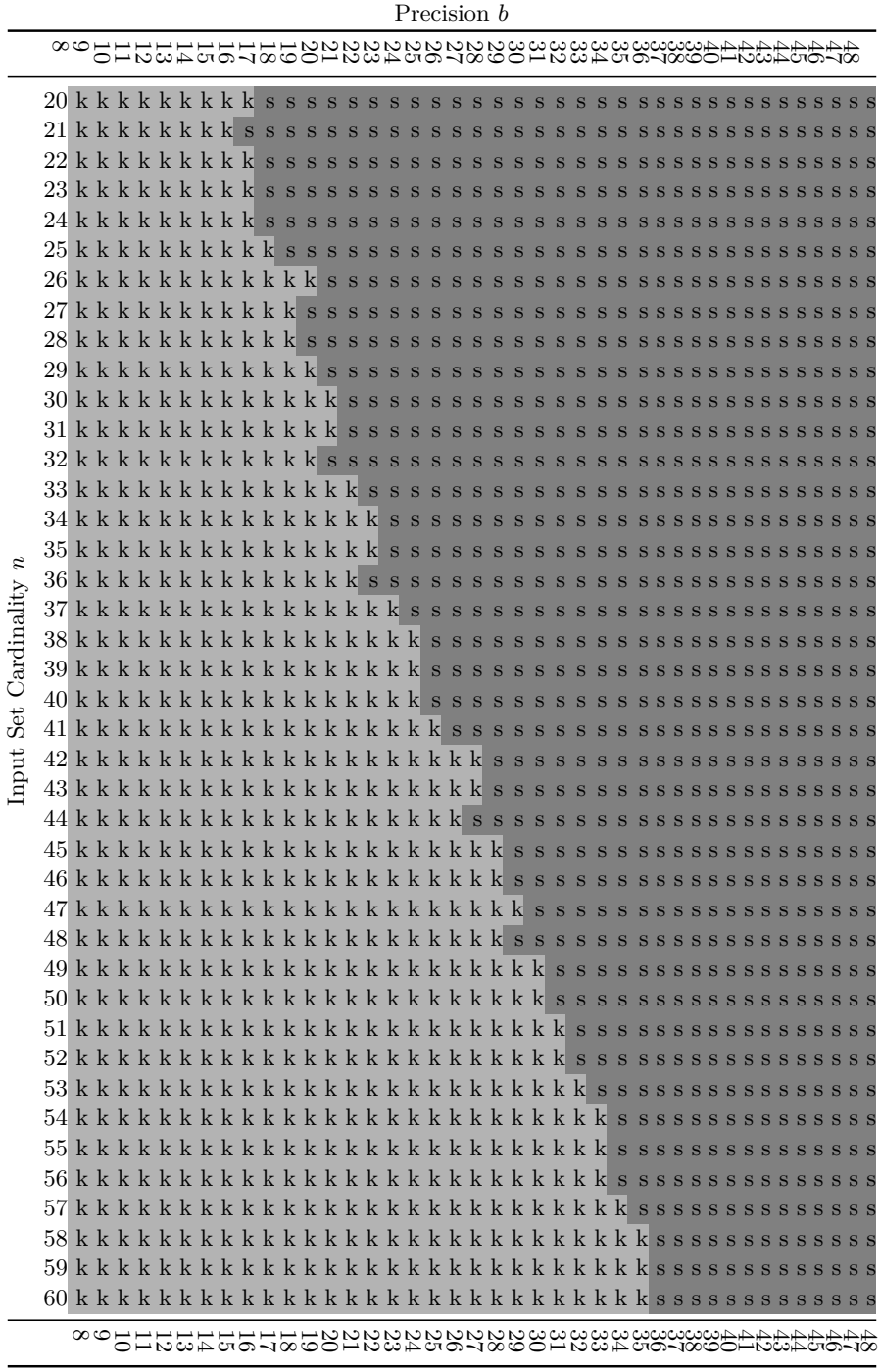


Table 2.3: The algorithm with the fastest average run time for two-way partitioning with $20 \leq n \leq 60$ and $8 \leq b \leq 48$.

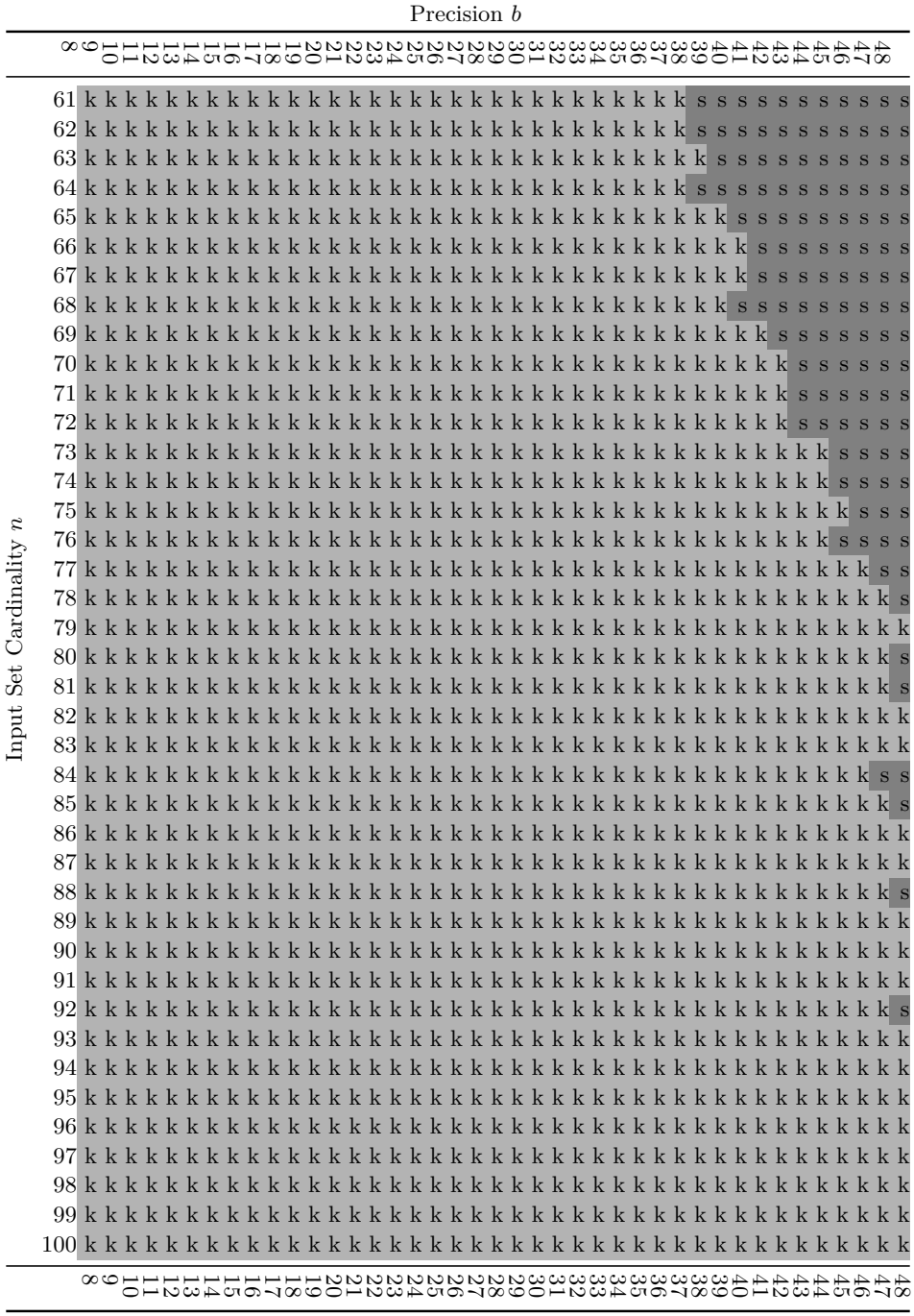


Table 2.4: The algorithm with the fastest average run time for two-way partitioning with $61 \leq n \leq 100$ and $8 \leq b \leq 48$.

are $O(2^n)$ time algorithms which use linear space to find optimal solutions. Both HS and SS use additional memory to find optimal solutions in $O(2^{\frac{n}{2}})$ time but use exponential space to do so. HS uses $O(2^{\frac{n}{2}})$ space while SS uses $O(2^{\frac{n}{4}})$. Extended versions of HS and SS will be used for some of the multi-way algorithms in subsequent chapters to generate subsets of integers whose sums fall within a specified range.

Finally, this chapter discussed dynamic programming for number partitioning, a pseudo-polynomial time algorithm whose time and space complexity depends on the number of input integers n and the precision of these integers represented by the sum of the input integers S. Its time and space complexity is $O(n \cdot \text{sum}(S))$. Despite the fact that it is pseudo-polynomial, it is dominated by the other algorithms.

The state-of-the-art algorithm for two-way number partitioning algorithm depends on the size of the input set n and the precision of the input numbers. For each value of n in the range [20,100] and b in the range [8,48], we generated 100 instances uniformly at random from within the range $[1, 2^b]$. Tables 2.3 and 2.4 show the algorithm that had the best average run time for each combination of n and b. k represents CKK and s represents SS. The trend is that for each n, CKK is the best algorithm for low precision and SS is best for high precision.

Part II

Multi-Way Number Partitioning

CHAPTER 3

Branch-and-Bound Algorithms

The multi-way number-partitioning problem is to separate a multiset $S = \{s_1, s_2, ..., s_n\}$ into k mutually exclusive and collectively exhaustive subsets such that the largest subset sum in minimized (section 1.0.1). This chapter introduces algorithms for solving this problem approximately as well as optimally using branch-and-bound algorithms. The approximate algorithms are used as upper bounds on each subset sum for the optimal algorithms. Lower bounds both on solution cost as well as the sum of each individual subset are also introduced.

3.1 Polynomial-Time Approximation Algorithms (Upper Bounds)

This section introduces two polynomial-time approximation algorithms used as upper bounds within the multi-way number-partitioning algorithms presented in this thesis. There is a large literature on approximation algorithms for the multi-way number-partitioning problem. See for example, [MJG01, IM08, Che04, DIM08]. While this large literature exists, we found the simple approximation algorithms we present to be sufficient for our purposes.

3.1.1 Multi-Way Greedy Algorithm or Longest Processing Time

The greedy algorithm for multi-way number partitioning extends the greedy algorithm for the two-way partition problem described in section 2.3.1. This algorithm is also known as longest processing time (LPT). Graham proved that LPT achieves an upper bound no worse than $\left(\frac{4}{3} - \frac{1}{3k}\right) \times \textit{optimal}$ [Gra66]. The greedy algorithm is optimal for $n \leq k + 2$ [Kor11].

The greedy algorithm first sorts the integers of S into decreasing order. It then considers

the integers one at a time and places them into the subset S_i, $1 \leq i \leq k$, with the smallest sum. If two or more subsets both have the same smallest sum, only one of the subsets is chosen arbitrarily.

Example 3.1.1 - Multi-Way Greedy Algorithm

Consider the input set $S = \{24, 21, 18, 17, 12, 11, 8, 2\}$. The following table shows the steps the greedy algorithm takes to compute an upper bound for the cost of a three-way partition. The first column shows the integers remaining in S at each step. The next three columns show the sums of the integers in subsets S_1, S_2 and S_3. The final column reports the action taken resulting in the values of S_1, S_2 and S_3 on the current row.

S	Subset Sum			Action
	S_1	S_2	S_3	
$\{24, 21, 18, 17, 12, 11, 8, 2\}$	0	0	0	Sort S into decreasing order.
$\{21, 18, 17, 12, 11, 8, 2\}$	24	0	0	Put 24 into S_1
$\{18, 17, 12, 11, 8, 2\}$	24	21	0	Put 21 into S_2
$\{17, 12, 11, 8, 2\}$	24	21	18	Put 18 into S_3
$\{12, 11, 8, 2\}$	24	21	35	Put 17 into S_3
$\{11, 8, 2\}$	24	33	35	Put 12 into S_2
$\{8, 2\}$	35	33	35	Put 11 into S_1
$\{2\}$	35	41	35	Put 8 into S_2
$\{\}$	37	41	35	Put 2 into S_1

The final heuristic value is 41, the largest subset sum, in this case S_2.

3.1.2 Multi-Way Karmarkar-Karp (KK)

The multi-way Karmarkar-Karp (KK) algorithm [KK82] extends the two-way KK algorithm described in section 2.3.2. A state of the multi-way KK algorithm, where k is the number of subset sums, is represented by a list of k-tuples. Each tuple corresponds to the difference in sums between each of the k sets the tuple represents. For example, the tuple (35,33,32) corresponds to three subsets with the sum of the integers in the first subset being three more than the sum of the integers in the third subset and two more than the sum of the integers in the second subset. The sum of the integers in the second subset is one more than the sum of the integers in the third set.

Since only relative values matter, the tuple is normalized by subtracting its minimum value. The tuple above is normalized to $(35\text{-}32, 33\text{-}32, 32\text{-}32) = (3, 1, 0)$. Each tuple is kept sorted in decreasing order and the list of tuples is kept sorted by the largest integer in each tuple, also in decreasing order. When a new tuple is inserted into the list, and has the same largest value as another tuple already in the list, the new tuple is arbitrarily placed first.

Initially, one tuple is created for each of the integers in S. The first integer of the tuple is the value from S, and the remaining integers are all set to 0. Each of these tuples corresponds to putting the integer from S into one subset, and nothing in each of the other subsets.

At each step of the KK algorithm, the first two tuples in the list (the two tuples whose largest integers are greatest) are combined. Given the tuples $A = (a_1, ..., a_k)$ and $B = (b_1, ..., b_k)$, both sorted in decreasing order, A is combined with B to form the new tuple C:

$$C = (a_1 + b_k,\ a_2 + b_{k-1},\ ...,\ a_k + b_1)$$

A and B are combined in this manner to try to minimize the largest values in C. After combining A and B, C is normalized by subtracting the minimum value in C from each element in C. This normalization is possible since KK only keeps track of the relative difference between subset sums. This combination process continues until one tuple remains.

Example 3.1.2 - Multi-Way Karmarkar-Karp Algorithm

Consider the input set $S = \{24, 21, 18, 17, 12, 11, 8, 2\}$. The following table shows the steps the Karmarkar-Karp algorithm takes to compute an upper bound for the cost of a three-way partition.

The initial step $i = 1$ puts each of the integers of S into their own tuple and sorts the tuples in decreasing order by their largest sum. At each subsequent step, the two tuples with the largest maximum sum (always tuples T_1 and T_2 in the table) are combined according to the rule $(a_1, a_2, a_3) + (b_a, b_2, b_3) = (a_1 + b_3, a_2 + b_2, a_3 + b_1)$. The resulting tuple is then normalized by subtracting the minimum integer in the tuple from each integer in the tuple.

The newly inserted tuple is shown in bold at each step. For example, at $i = 4$, $(17, 12, 0) + (11, 0, 0) = (17 + 0, 12 + 0, 11 + 0) = (17, 12, 11)$. This tuple is normalized by subtracting 11 from each integer resulting in (6,1,0), which is already sorted in decreasing order. The new tuple is inserted in bold at $i = 5$. It is placed before the tuple (6,3,0) with the same largest value 6 since newer tuples always come first in case of a tie.

i	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
1	(24,0,0)	(21,0,0)	(18,0,0)	(17,0,0)	(12,0,0)	(11,0,0)	(8,0,0)	(2,0,0)
2	**(24, 21, 0)**	(18,0,0)	(17,0,0)	(12,0,0)	(11,0,0)	(8,0,0)	(2,0,0)	
3	(17,0,0)	(12,0,0)	(11,0,0)	(8,0,0)	**(6, 3, 0)**	(2,0,0)		
4	**(17, 12, 0)**	(11,0,0)	(8,0,0)	(6,3,0)	(2,0,0)			
5	(8,0,0)	**(6, 1, 0)**	(6,3,0)	(2,0,0)				
6	**(7, 5, 0)**	(6,3,0)	(2,0,0)					
7	**(2, 1, 0)**	(2,0,0)						
8	**(1, 1, 0)**							

The final tuple, (1,1,0) corresponds to the partition, $\langle\{24, 12, 2\}, \{21, 17\}, \{18, 11, 8\}\rangle$, whose sums are 38, 38 and 37. KK has found a perfect partition with cost 38.

3.2 Lower Bounds

This section defines two classes of lower bounds for multi-way number partitioning. The first class is the lower bound on the overall solution cost. That is, what is the smallest possible value for the maximum sum of the k subsets in an optimal partition? The second class is the lower bound on the sum of any particular subset.

3.2.1 On Solution Cost

Dell'Amico and Martello [DM95] define three lower bounds on solution cost.

- L_0: Relax the constraint that an input integer cannot be split between multiple subsets.
- L_1: C^* must be at least as large as the largest integer of S.
- L_2: Since there are only k subsets, at least two of the $k+1$ largest integers must go into the same subset. The smallest sum for this subset is achieved by choosing the smallest two integers of the largest $k+1$, which are s_k and s_{k+1}.

Assuming $S = \{s_1, s_2, ..., s_n\}$ is sorted in decreasing order, the following are the mathematical formulas for these three bounds:

$$L_0 = \left\lceil \frac{\text{sum}(S)}{k} \right\rceil \qquad L_1 = \max\{L_0, s_1\} \qquad L_2 = \max\{L_1, s_k + s_{k+1}\}$$

The L_0 lower bound defines a perfect partition, the best solution cost possible for any problem instance. The L_1 and L_2 lower bounds are only useful if the largest input integer or the sum of the k^{th} and $k+1^{st}$ largest input integers are greater than the cost of a perfect partition. This is rarely the case for the experiments run in this thesis.

3.2.2 On Subset Sum

The lower bound on any subset sum is the smallest sum for one subset such that if the sum of the remaining integers were partitioned perfectly into $k-1$ subsets, it would have cost less than ub [Kor09]. This lower bound is defined as:

$$lb = \text{sum}(S) - (k-1) \times (ub-1)$$

This lower bound forces $k-1$ of the subsets (all but one) to have sum $ub-1$, the maximum sum that could lead to a solution better than the best found so far. It subtracts the total sum of these $k-1$ subsets from the total sum of the input set S. This remaining capacity is the lower bound for any one subset. If a subset had sum smaller than this lb, than one of the remaining $k-1$ subsets would be forced to have sum greater than ub and thus could not lead to a better solution.

3.3 Generating Subsets with Sums within a Range

Given a lower bound lb and upper bound ub on subset sums, an important subroutine in multi-way number partitioning algorithms is to generate subsets with sums within the range $[lb, ub-1]$. This section introduces algorithms for generating all subsets with sums within a range.

3.3.1 Inclusion-Exclusion (IE) Binary Tree Search

Perhaps the most straightforward way to generate subsets of S with sums within a range is using the inclusion-exclusion (IE) algorithm [Kor09]. IE traverses a binary tree with each node representing a collection of subsets. The root node corresponds to the power set and the leaves correspond to individual subsets. Each level corresponds to an integer of S with subsets including the integer on the left branch and excluding it on the right. IE sorts S then considers the integers in decreasing order, searching the tree from left to right always

including integers before excluding them.

IE prunes the tree under the following conditions:

1. If the sum of the integers included at a node exceeds $ub - 1$.

2. If the sum of the integers included at a node plus all non-assigned integers is less than lb.

In the worst case, IE runs in time $O(2^n)$, the size of the complete binary tree; and space $O(n)$, the depth of the complete binary tree.

Example 3.3.1 - Inclusion-Exclusion Binary Tree Search

Consider the input set S=$\{8, 6, 5, 3\}$. Figure 3.1 shows the complete IE binary search tree. The left child of the root includes the integer 8 while the right child excludes it. In general, the left children of the nodes of level i include the i^{th} integer of S while the right children exclude it. Each node displays the sum of the integers of S included on the current path. The left-most path corresponds to the input set S and the right-most path to the empty set.

Figure 3.2 shows the binary tree IE searches to find all subsets with sums within the range $[lb, ub - 1]$ where $lb = 13$ and $ub = 17$. This tree is pruned according to the two rules of section 3.3.1. The pruned nodes are labeled with the pruning rule used.

For example, the left-most node with sum 19 corresponding to the subset {8, 6, 4} is pruned since its sum 19 is greater than $ub - 1 = 16$. The right-most node with sum 0 corresponding to the empty set is also pruned since 0 plus the remaining integers 5 and 3 sum to 8, which is less than $lb = 13$.

The leaves 14, 16, 13 and 14 in bold correspond to the subsets {8, 6}, {8, 5, 3}, {8, 5} and {6, 5, 3} respectively; all the subsets with sums within the range [lb, $ub - 1$].

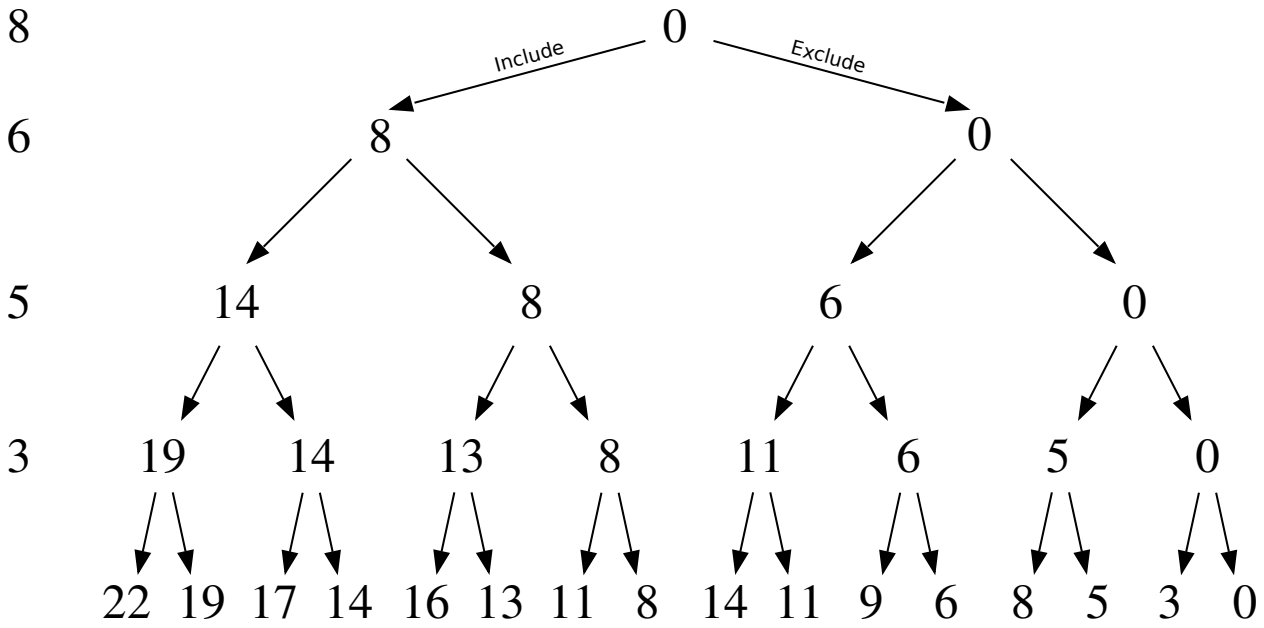


Figure 3.1: The full inclusion-exclusion binary tree for the input set $S = \{8, 6, 5, 3\}$.

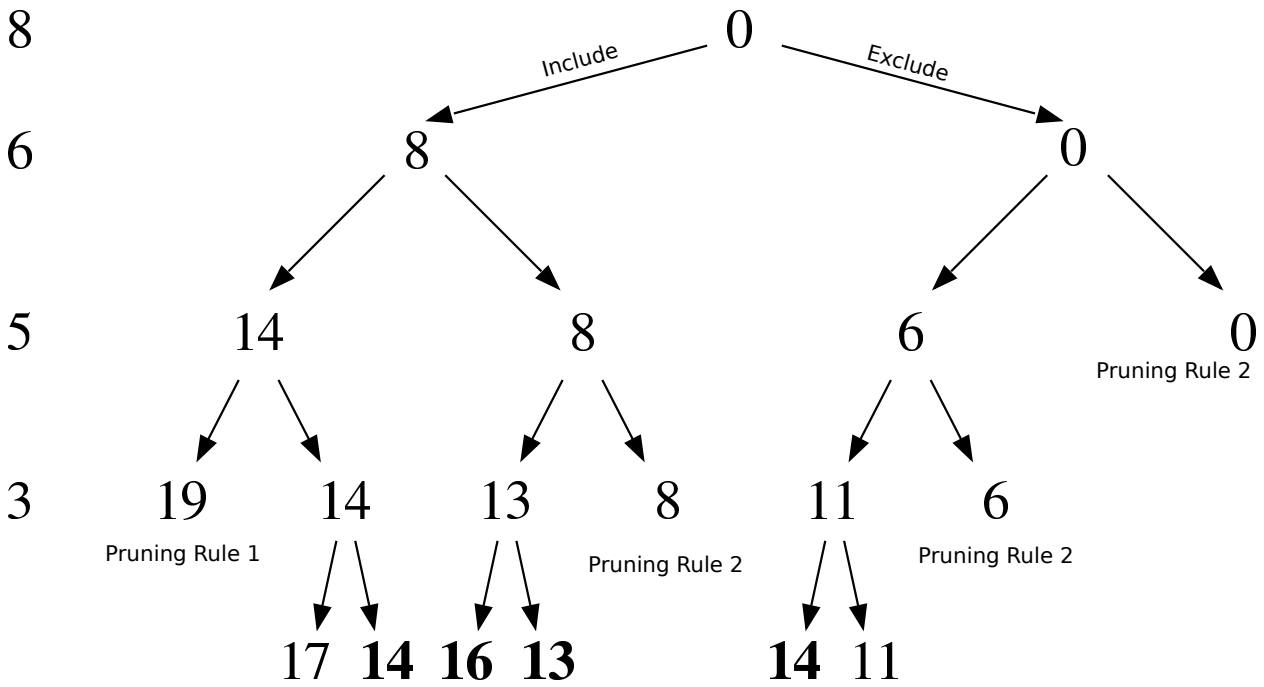


Figure 3.2: The pruned inclusion-exclusion binary tree for the input set $S = \{8, 6, 5, 3\}$ with $lb = 13$ and $ub = 17$. Leaves with sums in the range $[lb, ub - 1]$ are highlighted in bold.

3.3.2 Extended Horowitz and Sahni (EHS)

Given an input multiset $S = \{s_1, s_2, ..., s_n\}$ and a $target$ value, the Horowitz and Sahni (HS) algorithm, described in section 2.4.4, searches for a subset of S whose sum is as close as possible to $target$. The extended Horowitz and Sahni (EHS) algorithm [Kor11] extends HS to find all subsets with sums within the range $[lb, ub - 1]$.

Recall that HS breaks S into two "half sets" S_A and S_B; generates all $2^{\frac{n}{2}}$ subsets of S_A and S_B; and stores these subsets in A and B respectively. A is sorted in increasing order of subset sum and B is sorted in decreasing order. HS maintains two pointers a and b into A and B; and moves the pointers according to the rules in figure 2.4

EHS replaces the b pointer with two pointers b_1 and b_2 which are maintained throughout the running of EHS as follows:

- b_1 points to the first (largest) integer of S_B such that $a + b_1 < ub$.
- b_2 points to the last (smallest) integer of S_B such that $a + b_2 \geq lb$.

Each time a is incremented, b_1 and b_2 are incremented to maintain these invariants. For each value of a, all subsets consisting of a unioned with each subset between b_1 and b_2 have sums within the bounds $[lb,\ ub - 1]$.

3.3.3 Extended Schroeppel and Shamir (ESS)

The SS algorithm, described in section 2.4.5, is based upon the HS algorithm, but uses less memory. The extended Schroeppel and Shamir (ESS) algorithm [Kor11] extends SS to find all subsets with sums within the range $[lb, ub - 1]$.

Recall that SS breaks S into four "quarter sets" $S_{A_1}, S_{A_2}, S_{B_1}, S_{B_2}$ and generates all $2^{\frac{n}{4}}$ subsets of each "quarter set", storing them in A_1, A_2, B_1 and B_2 respectively. A_1 and A_2 are combined in a min heap to generate the subsets in the same order as A from HS while B_1 and B_2 are combined in a max heap to generate the subsets in the same order as B from HS.

The strategy for extending SS is similar to that of extending HS as described in the previous section. Instead of maintaining a pointer a to the current subset of the A set, the same value is generated from the min heap.

Since the subsets of the B set are generated from a max heap which combines subsets from the B_1 and B_2 lists, there is only access to one subset of B at a time. However, for each a, the EHS algorithm requires access to all subsets b from B such that when unioned with the a subset, have subset sums within the range $[lb, ub - 1]$.

To solve this problem, ESS maintains a list of the subsets from B having this property. The first subset in this list corresponds to the subset b_1 points to in the EHS algorithm while the last subset corresponds to the subset b_2 points to.

Each time the next a is popped from the min heap, all subsets b with the property that $sum(a \cup b) \geq lb$ are popped from the max heap and added to the end of the list. All subsets b from the front of this list with the property $sum(a \cup b) \geq ub$ are popped from the list. This list contains the exact subsets of the EHS algorithm with sums between those of b_1 and b_2.

The algorithm proceeds just as SS does, but for each a from the min heap and b from this stored list, it outputs $a \cup b$.

3.4 Improved Recursive Number Partitioning (IRNP)

Korf first introduced recursive number partitioning (RNP) in [Kor09]. He improved upon this algorithm with a hybrid algorithm called improved recursive number partitioning (IRNP) in [Kor11]. Both RNP and IRNP are anytime branch-and-bound algorithms. They both use the KK algorithm to generate an initial upper bound on the sums of the k subsets. This section describes IRNP, as this is the best version of RNP that Korf produced.

3.4.1 Recursive Principle of Optimality

At the core of IRNP is a recursive principle of optimality. Given a partition of S into k subsets $\langle S_1, S_2, ..., S_k \rangle$, a *decomposition* of the partition is a two-way partition of $\langle S_1, S_2, ..., S_k \rangle$ into collections of subsets of cardinality k_1 and k_2 where $k_1 + k2 = k$. For example, if $k = 3$, there are three different decompositions with $k_1 = 1$ and $k_2 = 2$:

$$\langle \langle S_1 \rangle, \langle S_2, S_3 \rangle \rangle \quad \langle \langle S_2 \rangle, \langle S_1, S_3 \rangle \rangle \quad \langle \langle S_3 \rangle, \langle S_1, S_3 \rangle \rangle$$

For $k = 4$, there are four different decompositions with $k_1 = 1$ and $k_2 = 3$ and three different decompositions with $k_1 = 2$ and $k_2 = 2$:

$$\langle \langle S_1 \rangle, \langle S_2, S_3, S_4 \rangle \rangle \quad \langle \langle S_2 \rangle, \langle S_1, S_3, S_4 \rangle \rangle \quad \langle \langle S_3 \rangle, \langle S_1, S_2, S_4 \rangle \rangle \quad \langle \langle S_4 \rangle, \langle S_1, S_2, S_3 \rangle \rangle$$
$$\langle \langle S_1, S_2 \rangle, \langle S_3, S_4 \rangle \rangle \quad \langle \langle S_1, S_3 \rangle, \langle S_2, S_4 \rangle \rangle \quad \langle \langle S_1, S_4 \rangle, \langle S_2, S_3 \rangle \rangle$$

Consider an optimal partition of S into k subsets and any decomposition of these k subsets into two collections of subsets of cardinality k_1 and k_2. The recursive principle of optimality states that any optimal partition of the integers in the first collection of subsets into k_1 subsets and any optimal partition of the integers in the second collection of subsets into k_2 subsets defines an optimal k-way partition. If either collection of subsets were not optimally partitioned k_1 or k_2 ways respectively, than optimally partitioning them could never raise their maximum subset sums. Thus, the principle is valid.

3.4.2 Initial Upper Bound

The initial upper bound for the branch-and-bound algorithm is computed using the KK polynomial-time approximation algorithm and its value stored in the variable *ub*. IRNP then proceeds to improve this bound until it finds an optimal partition.

3.4.3 Two-Way Balanced Recursive Partitioning

The recursive principle of optimality takes as a premise that S is partitioned optimally into k subsets, decomposes these k subsets into two collections of sets with k_1 and k_2 subsets in them and then asserts that any optimal k_1-way partition of the integers in the k_1 collection of subsets combined with any optimal k_2-way partition of the integers in the k_2 collection of subsets is an optimal partition.

In practice, IRNP does not start with an optimal k-way partition, so instead of decomposing the subsets, it considers all two-way partitions of S such that if the first set of integers is optimally subpartitioned k_1 ways and the second set of integers is optimally subpartitioned k_2 ways, it is possible that the combined partition could have cost better than ub.

At each recursive call, IRNP uses ESS (section 3.3.3) to generate all two-way partitions to be subpartitioned k_1-ways and k_2-ways respectively where $k_1 = \lfloor k/2 \rfloor$ and $k_2 = \lceil k/2 \rceil$. Call S^{k_1} the subset to be partitioned k_1 ways and S^{k_2} the subset to be partitioned k_2 ways. For example, for three-way partitioning, at the top level $k_1 = 1$ and $k_2 = 2$. For four-way partitioning, $k_1 = k_2 = 2$. For five-way partitioning, $k_1 = 2$ and $k_2 = 3$, etc. Since S^{k_2} is the complement of S^{k_1}, ESS only generates S^{k_1}, which determines S^{k_2}.

3.4.4 Two-Way Partition Bounds

In order to generate only the two-way partitions that could lead to a combined partition with cost less than ub, lower and upper bounds for the top-level partition must be chosen. Since S^{k_1} is to be partitioned $\lfloor k/2 \rfloor$ ways, and each of these subsets must have sums less than $ub - 1$, the upper bound for the sum of S^{k_1} is $\lfloor k/2 \rfloor \times (ub - 1)$. Similarly, the upper bound for the sum of S^{k_2} is $\lceil k/2 \rceil \times (ub - 1)$.

The upper bound for S^{k_2} implies a lower bound of $\text{sum}(S) - \lceil k/2 \rceil \times (ub - 1)$ for S^{k_1}, the sum remaining for S^{k_1} if S^{k_2} has sum equal to its upper bound. However, a better lower bound is available. Consider the cost of a perfect partition, $C_P^* = \left\lceil \frac{\text{sum}(S)}{k} \right\rceil$. Ignoring the ceiling, this is also the average sum of each of the k subsets in any partition. If we arbitrarily

$k_1 = \lfloor k/2 \rfloor$		$k_2 = \lceil k/2 \rceil$	
lb	ub	lb	ub
$\frac{\lfloor k/2 \rfloor}{k} \times \text{sum}(S)$	$\lfloor k/2 \rfloor \times (ub-1)$	$\frac{\lceil k/2 \rceil}{k} \times \text{sum}(S)$	$\lceil k/2 \rceil \times (ub-1)$

Table 3.1: Lower and upper bounds for the decomposition of input integers for IRNP.

enforce that the average sum of the subsets of S^{k_1} is greater than or equal to the average sum of the subsets of S^{k_2}, the lower bound for S^{k_1} becomes $\frac{k_1}{k} \times \text{sum}(S)$. That is, the ratio of the sum of the integers in the k_1 partition to the sum of the integers in S is at least as great as the ratio of k_1 to k. Since we enforce that the subset sum of S^{k_1} is greater than or equal to the subset sum of S^{k_1}, we avoid generating duplicates by permuting the two subsets.

For each S^{k_1} generated by ESS, IRNP has to decide whether to subpartition S^{k_1} or its complement S^{k_2} first. The subset with fewer integers is subpartitioned first because it can be done faster and is more likely to fail to achieve $ub - 1$. If the cost of subpartitioning the first subset (either S^{k_1} k_1 ways or S^{k_2} k_2 ways) is less than ub, then the other complement subset is recursively subpartitioned. If the cost of this second partition is also less than ub, the ub is set to the max of the two partition costs. This process continues until $lb = ub$ or the whole search space is exhaustively searched.

3.4.5 Partitioning Small Sets, a Hybrid Algorithm

For small sets of numbers, Korf reports that IRNP is slower than some other previous algorithms [Kor11]. He calls IRNP "A Hybrid Recursive Multi-Way Number Partitioning Algorithm" since depending on n and k, IRNP uses different algorithms to solve different instances. In general, for $n \leq k + 2$, greedy is an optimal algorithm, so it is used.

For two-way partitioning, CKK is used for $5 \leq n \leq 16$ and SS for $n \geq 17$. Even though CKK runs in $O(2^n)$ time and SS in $O(2^{\frac{n}{2}})$, there is constant overhead to SS which makes CKK faster for small n.

For three to ten-way partitioning, CGA is used for small n and IRNP for large n. Table

3.2 shows the algorithm used as a function of n for $3 \leq k \leq 10$.

$k \rightarrow$	3	4	5	6	7	8	9	10
CGA when $n \leq$	12	14	16	19	21	25	27	31
IRNP when $n \geq$	13	15	17	20	22	26	28	32

Table 3.2: The values of n for each k in which CGA or IRNP are used. From [Kor11].

3.4.6 Improvements of IRNP over RNP

At their core, both IRNP and RNP rely on the principle of recursive optimality and use recursive decomposition. However, there are three differences, described in this section.

Decomposition

For even values of k, IRNP and RNP decompose the numbers in the same manner. However, for odd values of k, they differ. IRNP partitions into two subsets such that the first subset is subpartitioned $\lfloor k/2 \rfloor$ ways and the second set is subpartitioned $\lceil k/2 \rceil$ ways. In contrast, RNP generates first subsets, then recursively partitions the remaining numbers $k-1$ ways.

Generating Subsets with Sums in Range

IRNP uses extended Schroeppel and Shamir (ESS) to generate subsets with sums in range. RNP uses inclusion-exclusion (IE) binary tree search. IE takes time $O(2^n)$ and space $O(n)$ while ESS takes time $O(2^{\frac{n}{2}})$ and space $O(2^{\frac{n}{4}})$. ESS in general is a much faster algorithm. Given the size of n and k that are tractably solved, the memory requirements of ESS are small.

Hybrid Algorithm

IRNP is a hybrid algorithm which also uses the CGA, CKK and SS algorithms depending on the values of n and k. RNP on the other hand uses CKK for all two-way partitions and is purely recursive for $k > 2$.

3.5 Moffitt Algorithm (MOF)

After Korf introduced RNP in 2009 and IRNP in 2011, Michael Moffitt introduced his algorithm in 2013 [Mof13], referred to as the Moffitt algorithm (MOF) in this thesis. Like IRNP, MOF is an anytime branch-and-bound-algorithm. However, there are also a number of major differences. The biggest difference is the method for decomposing subsets. While IRNP recursively decomposes its remaining integers into two balanced subsets, MOF sequentially generates all possible first subsets, then recursively partitions the remaining integers $k-1$ ways. MOF also uses a different algorithm for generating subsets with sums in a range. Finally, MOF introduces weakest-link optimality to replace the recursive principle of optimality. This section describes the MOF algorithm.

3.5.1 Weakest-Link Optimality

While the recursive principle of optimality is at the core of IRNP, weakest-link optimality is at the core of MOF. In order to optimally partition S into k subsets, IRNP partitions S into two subsets to be subpartitioned $k_1 = \lfloor k/2 \rfloor$ and $k_2 = \lceil k/2 \rceil$ ways. It then recursively subpartitions the k_1 set followed by the k_2 set, both optimally.

Call C^* the optimal cost of partitioning S into k subsets; and $C^*_{k_1}$ and $C^*_{k_2}$ the costs of optimally subpartitioning the numbers in the k_1 and k_2 subsets. If $C^*_{k_1} > C^*_{k_2}$, it is not necessary to optimally subpartition the k_2 subset in order to find an optimal solution. It is only necessary to find a subpartition with cost less than or equal to $C^*_{k_1}$ since $C^* = \max\{C^*_{k_1}, C^*_{k_2}\} = C^*_{k_1}$.

While recursively partitioning S, IRNP optimally partitions the remaining integers at every step. In contrast, MOF searches for subsets with cost less than or equal to the max of the costs of the subsets already constructed as ancestors in the recursive tree.

Example 3.5.1 - Weakest-Link Optimality

From [Mof13]: Consider the input set S=$\{1, 2, 3, 4, 5, 6, 7\}$ to be partitioned into $k = 3$ subsets. Here are two optimal partitions:

	S_1	S_2	S_3
Partition 1:	{3,7}	{4,6}	{1,2,5}
Partition 2:	{3,7}	{4,5}	{1,2,6}

Both partitions have the same cost of 10. However, look at S_2 and S_3. For partition 1, sum(S_2)=10 while sum(S_3)=8, but this is not an optimal partition of $S_2 \cup S_3$ into two subsets. For partition 2, both sum(S_2)=9 and sum(S_3)=9, which is an optimal partition of $S_2 \cup S_3$ into two subsets.

Since IRNP uses the recursive principle of optimality, it only searches for optimal partitions where all subpartitions are optimal as well, as in partition 2. MOF looks for any optimal solution, so it could terminate after finding either partition 1 or 2.

3.5.2 Sequential Recursive Partitioning

Improved recursive number partitioning (IRNP) decomposes S into two subsets to be subpartitioned $k_1 = \lfloor k/2 \rfloor$ and $k_2 = \lceil k/2 \rceil$ ways. It then recursively partitions the k_1 set followed by the k_2 set. In contrast, MOF generates all first subsets S_1 whose sums are within the range $[lb,\ ub - 1]$. Then, it recursively subpartitions the remaining integers $k - 1$ ways into the partition $\langle S_2, ..., S_k \rangle$. In this way, it generates the k subsets sequentially from S_1 to S_k by searching a recursive partitioning tree of depth k.

At each recursive call, MOF uses inclusion-exclusion (IE) binary tree search (section 3.3.1) to generate all first subsets with sums within the bounds $[lb, ub - 1]$.

For every optimal k-way partition $\langle S_1, ..., S_k \rangle$, there are $k!$ equivalent solutions possible by permuting the subsets S_i. For example, the partition $\langle \{8, 1\}, \{5, 2, 2\}, \{6, 3\} \rangle$ is equivalent

to $\langle\{5,2,2\},\{8,1\},\{6,3\}\rangle$. In order to eliminate all of these duplicates, the largest remaining integer is always included in the next subset of the partition.

3.5.3 Sequential Recursive Partitioning Bounds

The upper bound (ub) is the cost of the lowest-cost complete partition found so far. Initially, MOF sets ub to sum(S). As complete partitions with lower cost are found, ub is set to the new cost. For each partial partition $P_d = \langle S_1, ..., S_d\rangle$ at depth d of the branch-and-bound tree, MOF uses ub, the remaining integers S^R and the depth d to compute the lower bound $lb = \text{sum}(S^R) - (k-d)(ub-1)$ as described in section 3.2.2. If $lb \geq ub$, the search immediately returns ub.

Otherwise, if $lb < ub$, MOF generates all subsets S_{d+1} with sums within the range $[lb, ub-1]$ one at a time from S^R to create the partial partitions $P_{d+1} = \langle S_1, ..., S_d, S_{d+1}\rangle$ at depth $d+1$. For each partial partition P_{d+1}, the algorithm recursively partitions S^R, the remaining integers, $k-(d+1)$ ways. If the cost of any of these recursive partitions is less than or equal to the maximum sum of the subsets of P_{d+1}, the recursive call returns immediately. Since the cost of a partial partition is the maximum of its subset sums, the maximum sum of the subsets of P_{d+1} is the lowest possible cost for a complete partition that includes P_{d+1}. Otherwise, the algorithm returns the lesser of ub and the lowest-cost recursive partitioning.

3.5.4 Dominance Pruning for Inclusion-Exclusion

Along with the pruning rules described in section 3.3.1, Moffitt also adds a form of dominance pruning to IE [Mof13]. When running IE to generate subset S_i of the current partition $\langle S_1, S_2, ..., S_k \rangle$, the sum of the integers of S_i must be within the range $[lb, ub - 1]$. The remaining integers S^R are considered in decreasing order. Each integer is either included or excluded.

Consider input set $S = \{15, 12, 11, 6, 4, 3, 2\}$ and $ub = 23$. We run IE to generate candidate subsets for S_1 with $sum(S_1) < 23$. IE starts by including the largest (remaining) integer 15. It is forced to exclude 12 and 11 since either would exceed the upper bound if added to 15. IE first includes 6 resulting in the subset $\{15, 6\}$ with sum 21. No other integers can be added to $\{15, 6\}$ without equaling or exceeding the upper bound, so IE now excludes 6.

After 6 is excluded, the sum of all integers included below the corresponding exclusion branch must exceed 6. As IE continues, it will generate the subsets $\{15, 4, 3\}$ and $\{15, 4, 2\}$ with sums 22 and 21 respectively. The subset $\{15, 4, 2\}$ does not need to be considered, since in any partition that contains $S_1 = \{15, 4, 2\}$, the 4 and the 2 could be swapped for the 6 without changing the partition cost. Any place where 6 could fit in subsequent subsets $\langle S_2, S_3, ..., S_k \rangle$, 4 and 2 could also fit. Therefore, $S_1 = \{15, 4, 2\}$ is dominated and does not need to be considered.

More formally, call x an integer being considered by the IE algorithm while constructing subset S_i. Assume that $\text{sum}(S_i \cup x) < ub$. When x is excluded from S_i, the integers included below this exclusion branch must have sum greater than x. Any subset of integers whose sum is less than or equal to x can be pruned. This is a special case of the dominance pruning rules first introduced for bin packing by Martello and Toth [MT90a, MT90b]. These more general dominance rules are discussed in section 4.5.2 in our discussion of bin packing.

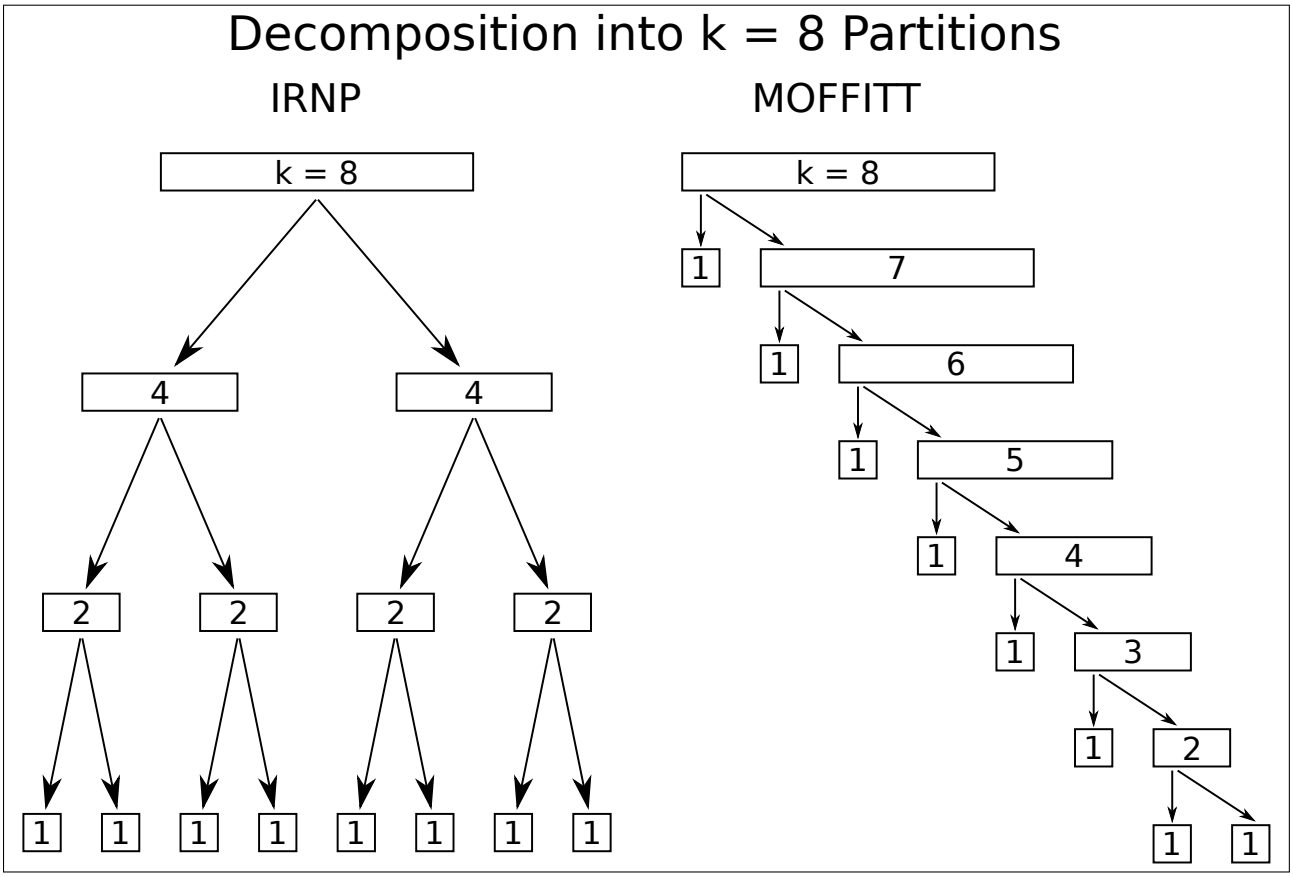


Figure 3.3: A comparison of the decomposition strategies of IRNP vs MOF for partitioning into $k = 8$ subsets. Each arrow represents a decomposition. The number of decompositions is exponential in the number of integers left to partition.

3.6 Experimental Results: RNP vs MOF

While there are many differences between improved recursive number partitioning (IRNP) and the Moffitt algorithm (MOF), there are two main differences. First, IRNP uses extended Schroeppel and Shamir (ESS) to generate subsets with sums within a range, while MOF uses inclusion-exclusion (IE). Second, IRNP recursively decomposes the input set into two balanced partitions to be subpartitioned, while MOF generates all subsets with sums within a range and sequentially subpartitions the remaining integers $k - 1$ ways.

To show the empirical differences between these two algorithms, we ran a series of experiments with integers sampled uniformly at random from the range $[1, 2^{48} - 1]$. We generated

100 problem instances for each n from $n = 30$ to 45. We chose to generate 48-bit numbers in order to generate hard instances without perfect partitions [Kor11]. All experiments were run on an Intel Xeon X5680 CPU running at 3.33GHz.

Table 3.3 reports the average run times for IRNP and MOF to partition the input sets of size $n = 30$ to 45 into $k = 3$ to 10 subsets. For each k, the column titled "R" reports the ratio of the average run times of MOF to IRNP. That is, it shows how many times longer on average it takes MOF to solve a problem instance as compared to IRNP.

For $k = 3$ to 5, IRNP dominates MOF and the ratio of their run times increases with n. For $k = 6$ to 10, MOF dominates IRNP with the ratio of the run times remaining relatively constant with respect to n. While IRNP dominates for small k, as k increases, MOF dominates.

Given a set of integers S, we could plot the sum of each of the subsets in the power set of S. These sums would form a bell shaped curve with a mode around $\text{sum}(S)/2$. For any multi-way partition problem with k subsets, the average sum of the integers in each subset is $\text{sum}(S)/k$. The goal is to minimize the largest subset sum which tends to push the subset sums towards the average sum. As k increases, there tends to be many fewer subsets with sums around $\text{sum}(S)/k$ than there are with sums around $\text{sum}(S)/2$. Since MOF generates subsets with sum around $\text{sum}(S)/k$ and IRNP generates subsets with sum around $\text{sum}(S)/2$, MOF tends to be much faster with increasing k as can be seen in table 3.3. This is despite the fact that RNP uses ESS with time complexity $O(2^{\frac{n}{2}})$ to generate subsets while MOF uses IE with time complexity $O(2^n)$.

3.7 Sequential Number Partitioning

In 2013, Korf and Schreiber combined the best ideas of IRNP and MOF to create sequential number partitioning (SNP) [1] [KSM13]. SNP is very similar to MOF, relying on weakest-link optimality. For k-way partitioning, like MOF, it uses sequential recursive partitioning to

[1] The name "sequential number partitioning" was also used for a different algorithm in [Kor09].

$k \rightarrow$	3-Way			4-Way			5-Way			6-Way		
$n \downarrow$	IRNP	MOF	R	IRNP	MOF	R	IRNP	MOF	R	IRNP	MOF	R
30	.00	.04	10	.03	.02	1	.11	.01	1/8	.72	.01	1/55
31	.00	.07	14	.04	.03	1	.20	.02	1/8	1.01	.02	1/48
32	.01	.12	15	.06	.05	1	.39	.04	1/10	1.70	.03	1/50
33	.01	.19	21	.10	.09	1	.54	.06	1/9	2.78	.06	1/50
34	.01	.36	25	.14	.16	1	.96	.11	1/9	4.28	.09	1/47
35	.02	.66	36	.17	.27	2	1.55	.18	1/9	6.84	.15	1/45
36	.03	1.21	40	.23	.45	2	2.58	.29	1/9	11.4	.23	1/50
37	.04	2.09	54	.32	.80	3	3.78	.49	1/8	17.3	.36	1/49
38	.06	3.80	64	.46	1.32	3	8.59	.82	1/10	28.8	.60	1/48
39	.08	6.77	86	.67	2.23	3	11.3	1.36	1/8	46.8	1.03	1/45
40	.13	11.7	87	1.03	3.72	4	19.9	2.15	1/9	80.8	1.50	1/54
41	.16	23.9	148	1.56	7.09	5	33.4	3.96	1/8	126	2.68	1/47
42	.27	39.5	148	2.23	11.2	5	60.4	6.08	1/10	224	4.63	1/48
43	.33	71.8	220	3.20	20.0	6	88.8	9.97	1/9	331	7.26	1/46
44	.54	135	252	5.11	34.0	7	163	17.3	1/9	565	11.8	1/48
45	.71	238	336	7.37	59.0	8	359	27.9	1/13	-	19.2	-

$k \rightarrow$	7-Way			8-Way			9-Way			10-Way		
$n \downarrow$	IRNP	MOF	R	IRNP	MOF	R	IRNP	MOF	R	IRNP	MOF	R
30	2.50	.01	1/197	24.9	.01	1/2k	41.3	.01	1/4k	144	.02	1/8k
31	3.35	.02	1/159	56.7	.07	1/811	85.0	.02	1/3k	265	.01	1/18k
32	5.21	.03	1/168	51.9	.03	1/1k	118	.02	1/4k	607	.02	1/33k
33	8.04	.05	1/167	72.0	.04	1/1k	281	.04	1/7k	1501	.03	1/44k
34	13.0	.08	1/165	99.4	7.04	1/14	442	15.8	1/28	2169	.09	1/23k
35	20.8	.14	1/154	169	.13	1/1k	543	.13	1/4k	6027	.11	1/56k
36	36.0	.20	1/184	253	.18	1/1k	973	.18	1/5k	-	.15	-
37	58.0	.34	1/173	428	.30	1/1k	1626	.30	1/5k	-	.40	-
38	106	.55	1/193	723	.55	1/1k	-	.59	-	-	.77	-
39	165	.88	1/188	1228	.89	1/1k	-	.79	-	-	.81	-
40	314	1.36	1/232	-	1.24	-	-	1.24	-	-	1.39	-
41	448	2.28	1/196	-	2.29	-	-	2.11	-	-	2.04	-
42	908	3.67	1/247	-	3.77	-	-	3.26	-	-	3.27	-
43	1351	5.99	1/226	-	5.89	-	-	5.44	-	-	5.26	-
44	2492	9.86	1/253	-	8.60	-	-	8.63	-	-	9.22	-
45	-	17.0	-	-	14.5	-	-	13.8	-	-	14.0	-

Table 3.3: The average time in seconds to optimally partition 48-bit integers 3 through 10 ways using IRNP and MOF.

generate all first subsets with sums within the range $[lb, ub-1]$. For each of these subsets, it recursively partitions the remaining integers $k-1$ ways. Given a partial partition $\langle S_1, ..., S_d \rangle$, if the remaining integers are subpartitioned $k-d$ ways such that the maximum subset sum is less than the maximum of the sums of S_1 through S_d, then the recursive partition can return immediately.

However, for generating subsets within a range, SNP uses the ESS algorithm like IRNP as opposed to the IE algorithm that MOF uses. IE always considers the input integers in decreasing order and includes integers before excluding them. It is this behaviour that allows it to use the dominance pruning rule described in section 3.5.4. ESS generates sets in a different and more complicated order so this dominance rule is not implemented.

3.8 Experimental Results: SNP vs MOF

We ran the SNP algorithm on the same set of instances described in section 3.6. The integers are sampled uniformly at random from the range $[1, 2^{48}-1]$. There are 100 problem instances for each n from $n = 30$ to 45. All experiments were run on an Intel Xeon X5680 CPU running at 3.33GHz.

Table 3.4 reports the average run times for SNP and MOF to partition input sets of size $n = 30$ to 45 into $k = 3$ to 10 subsets. For each k, the column titled "R" reports the ratio of the average run times of SNP to MOF. That is, it shows how many times longer on average it takes MOF to solve a problem instance as compared to SNP.

For $k = 3$ to 5, SNP dominates MOF and the ratio of their run times increases with n. For $k = 6$, MOF is fastest for small n but then SNP dominates for $n \geq 34$. For $k = 7$ and $k = 8$, MOF dominates for the numbers reported, but as n increases, the ratio decreases, suggesting that if n were to get large enough, SNP would dominate. There is no clear trend for $k = 9$. For $k = 10$, MOF dominates, and the margin increases with increasing n.

Notice that for $n = 34$ and $k = 9$, the average times for both SNP and MOF do not follow a trend with the rest of the series for $k = 9$. This is due to the results for one problem

which took 96455.9 seconds for SNP and 1572.27 seconds for MOF. This shows how sensitive these algorithms can be to individual problem instances.

3.9 Experimental Results: SNP vs MOF vs IRNP

Figure 3.4 shows the same experimental results for SNP, MOF and IRNP in graph form. There is a separate graph for each value of k from 3 to 10 and each graph shows average times for each n from 30 to 45. These graphs show that SNP is faster than IRNP for all k from 4 to 10 and about the same speed for $k = 3$. SNP outperforms MOF for $k \leq 6$ while MOF outperforms SNP for $k \geq 7$. As we mentioned in the last section, it seems that for larger n, SNP will eventually outperform MOF for $k = 8$ and $k = 9$.

3.10 Summary

This chapter has introduced three branch-and-bound algorithms for solving the multi-way number partitioning problem: improved recursive number partitioning (IRNP), the Moffitt algorithm (MOF) and sequential number partitioning (SNP). The multi-way version of the greedy algorithm, also known as longest processing time, and the multi-way version of the Karmarkar-Karp heuristic are approximation algorithms. These are used to compute upper bounds for the branch-and-bound algorithms.

Both IRNP and MOF recursively partition the input set S into k subsets. IRNP uses the recursive principle of optimality to recursively partition S into two subsets to be partitioned $\lfloor k/2 \rfloor$ and $\lceil k/2 \rceil$ ways. MOF uses the principle of weakest link optimality to consider all first subsets with sums within the lower and upper bounds, then recursively partitions the remaining integers $k - 1$ ways.

Both IRNP and MOF need to generate subsets with sums within a lower and upper bound. IRNP uses extended Schroeppel and Shamir (ESS), which is a more memory efficient version of extended Horowitz and Sahni. MOF uses inclusion-exclusion binary tree search

$k \rightarrow$	3-Way			4-Way			5-Way			6-Way		
$n \downarrow$	SNP	MOF	R	SNP	MOF	R	SNP	MOF	R	SNP	MOF	R
30	.00	.04	20	.00	.02	5	.01	.01	2	.02	.01	1/2
31	.00	.07	22	.01	.03	6	.01	.02	2	.03	.02	1
32	.00	.12	31	.01	.05	7	.02	.04	2	.04	.03	1
33	.01	.19	31	.01	.09	7	.03	.06	2	.06	.06	1
34	.01	.36	48	.02	.16	10	.03	.11	3	.08	.09	1
35	.01	.66	53	.02	.27	11	.05	.18	3	.13	.15	1
36	.02	1.21	78	.03	.45	14	.08	.29	4	.17	.23	1
37	.03	2.09	78	.05	.80	15	.12	.49	4	.25	.36	1
38	.03	3.80	119	.07	1.32	19	.17	.82	5	.39	.60	2
39	.05	6.77	129	.11	2.23	20	.26	1.36	5	.62	1.03	2
40	.07	11.7	178	.15	3.72	25	.36	2.15	6	.83	1.50	2
41	.11	23.9	210	.25	7.09	29	.59	3.96	7	1.29	2.68	2
42	.14	39.5	291	.33	11.2	34	.86	6.08	7	2.16	4.63	2
43	.22	71.8	325	.51	20.0	39	1.27	9.97	8	3.17	7.26	2
44	.28	135	485	.68	34.0	50	1.77	17.3	10	4.35	11.8	3
45	.49	238	489	1.13	59.0	52	2.64	27.9	11	7.07	19.2	3

$k \rightarrow$	7-Way			8-Way			9-Way			10-Way		
$n \downarrow$	SNP	MOF	R	SNP	MOF	R	SNP	MOF	R	SNP	MOF	R
30	.04	.01	1/3	.06	.01	1/5	.08	.01	1/8	.06	.02	1/3
31	.06	.02	1/3	.19	.07	1/3	.13	.02	1/6	.13	.01	1/9
32	.09	.03	1/3	.14	.03	1/5	.19	.02	1/8	.20	.02	1/11
33	.13	.05	1/3	.23	.04	1/5	.31	.04	1/8	.34	.03	1/10
34	.18	.08	1/2	.35	7.04	20	965	15.8	1/61	.48	.09	1/5
35	.29	.14	1/2	.59	.13	1/5	.92	.13	1/7	.90	.11	1/8
36	.41	.20	1/2	.80	.18	1/4	1.29	.18	1/7	1.47	.15	1/10
37	.64	.34	1/2	1.33	.30	1/4	2.29	.30	1/8	3.60	.40	1/9
38	.98	.55	1/2	2.24	.55	1/4	4.53	.59	1/8	6.91	.77	1/9
39	1.46	.88	1/2	3.44	.89	1/4	5.91	.79	1/7	9.63	.81	1/12
40	1.96	1.36	1	4.79	1.24	1/4	8.55	1.24	1/7	15.2	1.39	1/11
41	3.02	2.28	1	7.70	2.29	1/3	14.6	2.11	1/7	25.9	2.04	1/13
42	5.03	3.67	1	12.9	3.77	1/3	24.9	3.26	1/8	42.9	3.27	1/13
43	7.47	5.99	1	19.4	5.89	1/3	40.4	5.44	1/7	78.6	5.26	1/15
44	11.1	9.86	1	24.7	8.60	1/3	59.6	8.63	1/7	123	9.22	1/13
45	19.5	17.0	1	39.7	14.5	1/3	87.9	13.8	1/6	-	14.0	-

Table 3.4: The average time in seconds to optimally partition 48-bit integers 3 through 10 ways using SNP and MOF.

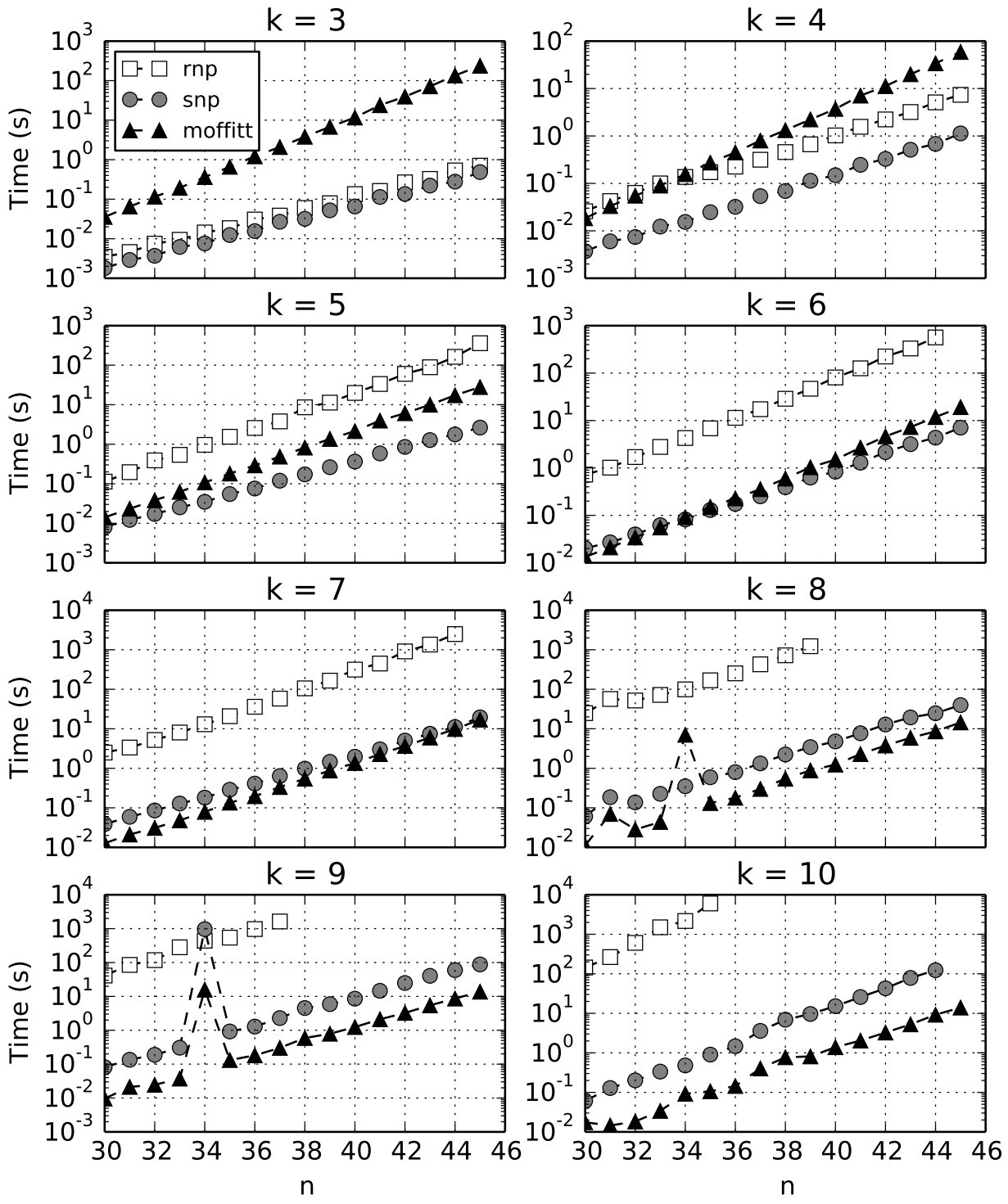


Figure 3.4: The average time in seconds to optimally partition 48-bit integers 3 through 10 ways using SNP, IRNP and MOF.

(IE). SS uses $O(2^{\frac{n}{2}})$ time and $O(2^{\frac{n}{4}})$ space. IE uses $O(2^{\frac{n}{)}}$ time and linear space.

SNP is an attempt to combine the best features of IRNP and MOF into one algorithm. It uses the recursive partitioning style of Moffitt but generates subsets with sums in a range like IRNP. SNP dominates both RNP and MOF for small k but MOF is the dominant algorithm for $k \geq 7$.

CHAPTER 4

Bin Packing Algorithm

This chapter describes the relationship between bin packing and number partitioning. Specifically, we focus on solving the multi-way partition problem (section 1.0.1) using solvers for the bin-packing problem (section 1.1.2).

We start by describing the dual relationship between bin-packing and number partitioning. We then introduce lower and upper bounds for bin packing. Then we cover MULTIFIT, an algorithm for solving number-partitioning problems using bin-packing algorithms.

Finally, we cover two different types of bin-packing solvers: bin completion (BC), which came from the artificial intelligence community; and branch-and-cut-and-price (BCP), which arose from operations research. While any bin-packing algorithm can be used to solve number-partitioning problems, BC and BCP are the only bin-packing algorithms we are aware of in the literature that have been used to solve number-partitioning problems optimally. For information on other optimal bin-packing algorithms in general, see for example [FK05, Sch02, MT90a].

4.1 Relationship Between Bin Packing and Number Partitioning

To describe how to solve number partitioning using bin-packing algorithms, let's revisit the definitions of the optimization versions of the bin-packing and multi-way number partitioning problems, underlining the differences:

Multi-Way Number Partitioning

Given an input multiset $S = \{s_1, s_2, ..., s_n\}$ of positive integers and a **number of**

subsets k,

separate the integers of S into $\boldsymbol{k}$ mutually exclusive and collectively exhaustive subsets, **minimizing the largest subset sum**.

Bin Packing

Given an input multiset $S = \{s_1, s_2, ..., s_n\}$ of positive integers and a **capacity C**, separate the integers of S into mutually exclusive and collectively exhaustive subsets, **with sums less than or equal to C, minimizing the number of subsets**.

	Bin Packing	**Number Partitioning**
Given:	S	S
Fixed:	Subset sum (capacity)	Number of subsets
Minimize:	Number of subsets (bins)	Subset sum

Table 4.1: Difference between bin packing and number partitioning.

In some sense, bin packing and number partitioning are dual problems. Bin packing fixes the subset sum and minimizes the number of subsets, while number partitioning fixes the number of subsets and minimizes the subset sum. In section 4.4, we will introduce MULTIFIT, an algorithm that exploits this duality to solve number-partitioning problems using bin-packing algorithms. First, we introduce lower and upper bounds for bin-packing problems. These bounds are required in order to run MULTIFIT.

4.2 Lower Bounds

A lower bound (lb) function for bin packing calculates the minimum number of subsets of sum less than or equal to C needed to pack the integers of the input set S. If a solution of lb subsets is found, search can terminate immediately and return this value as an optimal solution. Martello and Toth introduced two classic bin packing lower bounds which they refer to as L_1 and L_2 [MT90b]. Both algorithms first sort the input set S in decreasing

order.

4.2.1 L_1 Lower Bound

The lower bound L_1 relaxes of the constraint that integers cannot be split between multiple bins. It is calculated as:

$$L1 = \left\lceil \frac{\text{Sum of all input integers}}{\text{Bin Capacity}} \right\rceil = \left\lceil \frac{\text{sum}(S)}{C} \right\rceil$$

Example 4.2.1 - L_1 lower bound

Consider the input set $S = \{99, 97, 94, 93, 8, 5, 4\}$ and the bin capacity $C = 100$ (This example is taken from [Kor02].)

The L_1 bound is:

$$L1 = \left\lceil \frac{\text{sum}(S)}{C} \right\rceil = \left\lceil \frac{400}{100} \right\rceil = 4.$$

4.2.2 L_2 Lower Bound

Their second lower bound L_2 improves upon the L_1 bound. It estimates a minimum amount of wasted space w in any solution and adds this to the total sum of S before dividing by the bin capacity. The formula for the L_2 lower bound is:

$$L1 = \left\lceil \frac{w + \text{sum}(S)}{C} \right\rceil$$

We demonstrate how to calculate w through the following example 4.2.2:

Example 4.2.2 - L_2 lower bound

Consider again the input set $S = \{99, 97, 94, 93, 8, 5, 4\}$ and the bin capacity $C = 100$.

Bin 1: Place 99, the largest integer of S into bin 1. Since no other integer of S fits with the 99, there is $\boldsymbol{C - 99 = 1}$ unit of wasted space in this bin.

Bin 2: $S = \{97, 94, 93, 8, 5, 4\}$ - Place 97 into bin 2. Since no other integer of S fits with the 97, there are $\boldsymbol{C - 97 = 3}$ units of wasted space in this bin.

Bin 3: $S = \{94, 93, 8, 5, 4\}$ Place 94 into bin 3. Both the 5 and the 4 fit with the 94. However, to avoid an exponential time algorithm, we don't branch and consider each integer. Instead, we consider 9, the sum of the two integers. We place 6 units of the 9 into bin 3 which leaves **no waste** and carry over the remaining 3 units to subsequent bins.

Bin 4: $S = \{93, 8\}$ - Place 93 into bin 4. We then add the carry from the previous bin. Since the entire carry of 3 units fit, we add it to the bin, leaving $\boldsymbol{C - (93 + 3) = 4}$ more units of wasted space in this bin. If the entire carry had not fit, we would have used enough of the carry to fill this bin, and carried over the remainder to subsequent bins.

Bin 5: $S = \{8\}$ - Place 8, the only remaining integer of S into bin 5. This leaves $\boldsymbol{C - 8 = 92}$ more units of wasted space in this bin.

The total waste is the sum of the waste from the five bins we just described, $w = 1 + 1 + 0 + 4 + 92 = 100$. The L_2 bound is:

$$L1 = \left\lceil \frac{w + \text{sum}(S)}{C} \right\rceil = \left\lceil \frac{100 + 400}{100} \right\rceil = \left\lceil \frac{500}{100} \right\rceil = 5.$$

The following are the steps of the algorithm for calculating L_2:

L_2 lower bound, calculating w :

- Let w be the wasted space, initialize this to 0.
- Let *carry* be an amount carried over from previous bins, initialize this to 0.

While S is not empty, repeat:

1: Calculate the residual bin capacity left as $r = C - s_0$, where s_0 is the largest remaining number.
   - Remove s_0 from S.

2: Remove from S all integers with value less than or equal to r and store the sum of these integers plus *carry* in the variable *sum*.

3a: If $r \geq sum$: Add $r - sum$ to w, set $carry = 0$.

3b: If $r < sum$, set $carry = sum - r$.

After calculating w, the L_2 lower bound is defined as:

$$L2 = \left\lceil \frac{w + \text{sum}(S)}{C} \right\rceil$$

Figure 4.1: The steps to calculate the L_2 wasted space heuristic

4.3 Polynomial Time Approximation Algorithms (Upper Bounds)

This section introduces two classic polynomial time approximation algorithms from the bin packing literature, first-fit decreasing and best-fit decreasing [Joh73, GGU72]. These are used by the bin completion algorithms as initial upper bounds to help prune the search space. For a more complete survey on approximation algorithms for bin packing, see [CGJ96].

4.3.1 First-Fit Decreasing Upper Bound

The first-fit decreasing (FFD) algorithm first sorts the input integers S in decreasing order. It keeps a list of bins, initially empty. FFD iterates through the integers of S and places each into the first bin in the list in which it fits. If it does not fit into any of the bins in the list, it is added to a new empty bin appended to the list.

Example 4.3.1 - First-Fit Decreasing

Consider the input set $S = \{15, 10, 6, 4, 3, 2\}$ and the bin capacity $C = 20$. The following table shows the steps that first-fit decreasing takes to compute an upper bound for the number of bins in an optimal packing.

S	**Action**	**Bin 1**	**Bin 2**	**Bin 3**
$\{15, 10, 6, 4, 3, 2\}$	Put 15 into Bin 1	{15}	-	-
$\{10, 6, 4, 3, 2\}$	Put 10 into Bin 2	{15}	{10}	-
$\{6, 4, 3, 2\}$	Put 6 into Bin 2	{15}	{10,6}	-
$\{4, 3, 2\}$	Put 4 into Bin 1	{15,4}	{10,6}	-
$\{3, 2\}$	Put 3 into Bin 2	{15,4}	{10,6,3}	-
$\{2\}$	Put 2 into Bin 3	{15,4}	{10,6,3}	{2}

FFD packs S into three bins of capacity 20.

4.3.2 Best-Fit Decreasing Upper Bound

The best-fit decreasing (BFD) algorithm also first sorts the input integers S into decreasing order and keeps a list of bins, initially empty. BFD iterates through the integers of S and places each into the bin in the list with the least capacity remaining in which it fits. If it does not fit into any of the bins in the list, a new empty bin is appended to the list.

Example 4.3.2 - Best-Fit Decreasing

Consider the input set $S = \{15, 10, 6, 4, 3, 2\}$ and the bin capacity $C = 20$. The following table shows the steps that best-fit decreasing takes to compute an upper bound for the number of bins in an optimal packing.

S	Action	Bin 1	Bin 2	Bin 3
$\{15, 10, 6, 4, 3, 2\}$	Put 15 into Bin 1	{15}	-	-
$\{10, 6, 4, 3, 2\}$	Put 10 into Bin 2	{15}	{10}	-
$\{6, 4, 3, 2\}$	Put 6 into Bin 2	{15}	{10,6}	-
$\{4, 3, 2\}$	Put 4 into Bin 2	{15}	{10,6,4}	-
$\{3, 2\}$	Put 3 into Bin 1	{15,3}	{10,6,4}	-
$\{2\}$	Put 2 into Bin 1	{15,3,2}	{10,6,4}	-

BFD requires two bins, which for this instance, is also an optimal solution.

4.4 MULTIFIT

In 1978, Coffman, Garey and Johnson introduced MULTIFIT, an approximation algorithm for number-partitioning problems using an approximation algorithm for bin packing and binary search [JGJ78]. MULTIFIT partitions S into k subsets while minimizing the maximum sum of the subsets by solving a sequence of bin-packing problems on S, varying the bin capacity C. It searches for the smallest C such that the number of subsets required is less than or equal to k, and calls this smallest capacity C^*.

Starting with lower (lb) and upper (ub) bounds on C^*, it performs a binary search over the bin capacities between lb and ub. Each probe of the binary search sets C to a value within the range $[lb, ub - 1]$ and uses FFD (section 4.3.1) to approximate the number of subsets needed to pack S into subsets with sum less than or equal to C. If the solution requires fewer than k subsets, the lower half of the remaining capacity space is searched,

otherwise the upper half is searched.

While the original algorithm uses FFD, MULTIFIT can be run with any bin-packing solver. Dell'Amico, Iori, Martello and Monaci [DIM08] proposed an optimal algorithm using branch-and-cut-and-price (see section 4.6) as the bin-packing solver. Schreiber and Korf [SK13] use an improved version of the bin-completion algorithm (see section 4.5) as well as Gleb Belov's branch-and-cut-and-price algorithm [BS06] as the bin-packing solver.

Listing 4.1 shows the source code for MULTIFIT. The function packBins is a call to either FFD, BCP or improved bin completion (IBC) depending on which version of the algorithm is being run. When packBins is a call to IBC, the algorithm is called binary-search improved bin completion (BSIBC). When packBins is a call to BCP, the algorithm is called binary-search branch-and-cut-and-price (BSBCP).

Listing 4.1: The MULTIFIT algorithm

```
1   MULTIFIT(S,k,CMin,CMax) {
2     while (CMax > CMin) {
3       C  = (CMax + CMin) / 2
4         numBins = packBins(S,C)
5         if (numBins > k)
6           CMin = C + 1
7         else
8           CMax = C
9     }
10    return CMin
11  }
```

It is also possible to solve bin-packing problems using number partitioning algorithms. Given input set S and bin capacity C, we can first use an approximation such as FFD or BFD to generate an upper bound ub for the number of bins in an optimal packing. We can then use a number-partitioning algoritm with k set to $ub - 1$. If this algorithm finds a partition with value less than or equal to C, we then try $ub - 2$. This continues until the number-partitioning algorithm finds an optimal partition with cost greater than C. The smallest value of k for which the number-partitioning algorithm can find a partition with cost less than or equal to C is the number of bins in an optimal bin packing. Solving bin-packing

problems with a number-partitioning algorithm is beyond the scope of this thesis.

Notation	Definition
C	The capacity of a bin.
Feasible Set	A set of input integers with sum less than or equal to C.
waste	Given feasible set F, the amount of space left in the bin, $C - sum(F)$.
lb	The L_2 lower bound.
ub	The number of bins used in the best solution found so far.
w	The sum of the waste in all bin completions on the path to the current node.
W	The total allowed waste in a solution better than ub.

Table 4.2: Notation and terms used to describe the bin-completion algorithm.

4.5 Bin Completion (BC)

Classic bin-packing algorithms such as those of Eilon and Christofides [EC71] or Martello and Toth [MT90a] consider input integers one at a time and assign them to bins. These are item-oriented branch-and-bound algorithms and beyond the scope of this thesis. In contrast, bin completion (BC) [Kor02, Kor03] considers the bins one at a time and assigns a complete set of integers to them. This is a bin-oriented branch-and-bound algorithm.

We continue by first describing Korf's original BC algorithm [1]. We then introduce improved bin completion (IBC), our algorithm which extends BC to make it more effective for solving number-partitioning problems.

4.5.1 The Original Bin-Completion Algorithm

BC is a branch-and-bound algorithm with initial lower bounds computed using the L_2 wasted space heuristic (section 4.2.2) and upper bounds using BFD (section 4.3.2). If these bounds are equal, the BFD solution is returned as optimal. Otherwise, a tree search is performed with the variable ub initialized to the number of bins in the BFD solution.

[1] Korf wrote two papers on bin completion. When we say original, we are referring to his 2003 paper which had some implementation improvements over his 2002 paper.

A *feasible* set is a set of input integers with sum less than or equal to C. The assignment of a feasible set to a bin is called a *bin completion*. Each node of the branch-and-bound tree except the root corresponds to a bin completion. The children of the root correspond to the completions of the bin containing the largest integer. The grandchildren of the root correspond to the completions of the bin containing the largest remaining integer, and so forth. The largest integer is included for two reasons. First, to avoid duplicates that differ only by a permutation of the bins. Second, to shrink the remaining capacity of the bin which results in fewer feasible bin completions to consider.

The *waste* of a bin completion is the capacity remaining after it has been packed with feasible set F. For example, if C=10 and F=$\{4,5\}$, the waste is 1. BC keeps track of the sum of the waste in all bins on the path to the current node in the variable w. At any point during the branch-and-bound search, there is always a solution requiring ub bins. Therefore, only solutions of ub-1 or fewer bins must be considered. To achieve such a solution, w must be less than or equal to $[(ub-1) \times C] - \text{sum}(S)$, which is called the *total allowed waste*, or W.

At each node of the search tree, BC generates all feasible sets which include the largest remaining integer and whose waste when added to the previous bins' waste does not exceed W. These sets are sorted by their sums in decreasing order. BC then branches on each feasible set F, removing the integers of F from S for the subtree beneath the node corresponding to F. If all integers are packed in fewer than ub bins, ub is updated to the new value. If this new value equals the lower bound, BC terminates, returning this packing. Otherwise, the search terminates when all completions have been exhaustively searched.

4.5.2 Dominance

Some bin completions are dominated by others and need not be considered. Given feasible sets F_1 and F_2, F_1 dominates F_2 if an optimal solution after packing a bin with F_1 is at least as good as an optimal solution after packing the bin with F_2.

Martello and Toth [MT90a, MT90b] present the following dominance relation for bin packing. If all the integers of feasible set F_2 can be packed into bins whose capacities are the integers of feasible set F_1, then F_1 dominates F_2. For example, if $F_1 = \{6, 4\}$ and $F_2 = \{6, 3, 1\}$, then F_1 dominates F_2 since the integers of F_2 can be packed into bins whose capacities are the integers of F_1 as follows: $\langle\{6\}, \{3, 1\}\rangle$. Intuitively, this dominance rule makes sense. Assume we have a solution that includes $F_2 = \{6, 3, 1\}$ and the 4 in some other bin. If we swap the 4 for 3 and 1, we have $F_2 = \{6, 4\}$. The 3 and the 1 can fit anywhere that the 4 could fit. However, there are also places where 3 or 1 can fit in which 4 cannot fit. We gain strictly more flexibility in packing the remaining bins if $F_2 = \{6, 4\}$. BC uses this dominance rule to prune parts of the search space, dramatically cutting down the size of the search tree.

4.5.3 Generating Completions

At each node of the search tree, BC must generate all undominated feasible subsets that include the largest remaining integer. These subsets must all have sums less than or equal to the bin capacity C. If the sums are smaller than C, the bin will contain wasted space. The variable W is the total amount of wasted space allowed in a solution better than the current best and w is the total amount of wasted space in bins already completed. Therefore, $W - w$ is the maximum amount of space we have left to waste in the current bin, and the generated subsets must have sum greater than or equal to $C - (W - w)$. Given these bounds, BC generates all bin completions with sums in the range $[C - (W - w), C]$. It does so using inclusion-exclusion (IE) binary-tree search (section 3.3.1).

To eliminate some dominated subsets, the IE tree search is modified. Consider a node in the search tree with a *sum* corresponding to the sum of its included integers. The left branch includes the integer x corresponding to the depth of the node and the right branch excludes x. There are three cases, if:

- $\boldsymbol{sum + x > C}$: x must be excluded since including it would exceed the bin capacity.

- $\boldsymbol{sum + x = C}$: x must be included and the excluded branch is terminated below the corresponding node since any child in the exclusion branch is dominated, and including any more numbers besides x would exceed C.

- $\boldsymbol{sum + x < C}$: x can be either included or excluded. However, if x is excluded, the sum of the integers included below the exclusion node must exceed x or the corresponding subset would be dominated by the subsets generated in the inclusion branch.

While these rules prevent the IE algorithm from generating some dominated feasible subsets, it does not prevent all of them. To eliminate the remaining dominated subsets, an additional dominance test is necessary. Call F a feasible bin completion. If any subset of the included integers can be replaced with a single remaining excluded integer without exceeding the bin capacity C, than F is dominated. See [Kor03] for pseudocode for this dominance test.

4.5.4 Improved Bin Completion (IBC and BSIBC)

We have implemented three improvements to bin completion and call the resulting bin completion algorithm IBC. While the changes are simple conceptually, the experimental results are dramatic, speeding up the BC algorithm by up to five orders of magnitude. This improved version of bin completion is used with MULTIFIT to solve the number-partitioning problem (BSIBC).

4.5.5 Incrementally Generated Completions

The original BC algorithm generates all completions at each node before sorting them. At any node, if there are n integers remaining, then there are 2^n possible subsets. If a large fraction of these subsets have sums within the valid range $[C - (W - w), C]$ and are undominated, then there are an exponential number of completions to generate. This situation tends to arise when the number of integers per bin in an optimal solution is large. In this case, BC spends almost all of its time generating completions.

To avoid generating an exponential number of completions, we generate and buffer up to m undominated feasible completions at a time, sort them, and branch. If we don't find an optimal solution recursively searching the subtrees of each of these m completions, we then generate the next m completions.

The implementation of the IE tree for the original BC algorithm used a recursive depth-first search (DFS). To perform the search over the inclusion-exclusion tree incrementally, we use an iterative DFS that maintains its own explicit stack data structure. This way, we can generate as many completions at a time as necessary and keep our explicit stack in memory so we can come back later to resume the search.

4.5.6 Variable Ordering

As mentioned in the last section, after all feasible completion sets are generated, they are sorted in decreasing order of subset sum. However, this is not a total order. If two subsets have the same subset sum, it is arbitrary which completion will be tried first. This can cause a large disparity in search time depending on the implementation of the sort algorithm.

With our new sort comparator, the subsets are still sorted by sum in decreasing order, but ties are broken in favor of smaller cardinality subsets. The rationale is that with fewer integers filling the current bin, there is more flexibility for filling the remaining bins. If both the sum and cardinality are equal, the subset with the smallest unique integer comes second. For example, for $A = \{9, 7, 4, 1\}$ and $B = \{9, 7, 3, 2\}$, B comes before A.

Fukunaga suggested min-cardinality max-weight cardinality which first sorts on cardinality, and then subset sum [FK05]. Given the heuristic nature of variable ordering, different orders can work better or worse for different problem instances.

4.6 Branch-and-Cut-and-Price (BCP and BSBCP)

Branch-and-cut-and-price (BCP) is an operations research algorithm for solving an integer linear program. At the core of BCP is linear programming, which is a technique for solving a maximization or minimization problem given a set of constraints in the form of linear inequalities. The time complexity of solving a linear program is polynomial.

The values of the solution to a linear program are real numbers. However, the linear programming model for solving the bin-packing problem requires that the solution values be integer. In order to enforce integer values, branch-and-bound is used over a series of linear programming problems. Furthermore, the model for solving the bin-packing problem requires a number of variables exponential in n, the number of input integers. To deal with this large number of variables, a technique called column generation is used. Finally, there is an optimization for integer linear programming which adds constraints to cut down the feasible region for optimal solutions called cutting planes. In this section, we will discuss linear programming, branch-and-bound, cutting planes and column generation. Together, these four techniques form BCP. BCP is used with the MULTIFIT algorithm to solve the number-partitioning problem (BSBCP).

4.6.1 Linear Programming (LP)

At the core of BCP is linear Programming (LP). [Chv83, DT97, DT03] all provide an excellent introduction to the field. LP is a method for maximizing or minimizing a linear objective function given a set of linear constraints.

Let's consider the following simple example. Talia likes to knit and sell hats and scarves. It takes her three hours to make a hat and two hours to make a scarf. A hat requires 10 yards of yarn while a scarf requires 15 yards. She budgets 176 hours each month for her knitting and acquires 1000 yards of yarn for free on the first of each month. Hats sell for $20.00 each while scarves sell for $15.00. How many hats and how many scarves should Talia make each month in order to maximize her revenue? Let:

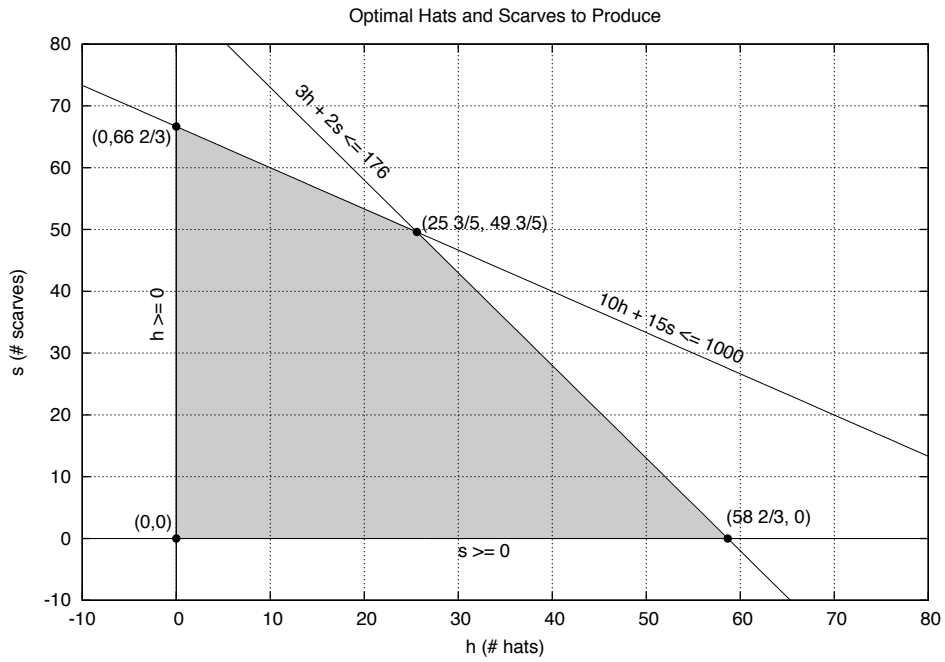


Figure 4.2: The shaded area is the feasible region for our linear programming example with three constraints. The points a,b and c are the corners of the feasible polytope.

- h be the number of hats Talia knits.
- s be the number of scarves Talia knits.

$$\boldsymbol{x} = \begin{bmatrix} h \\ s \end{bmatrix} \qquad \boldsymbol{b} = \begin{bmatrix} 176 \\ 1000 \end{bmatrix} \qquad \boldsymbol{c} = \begin{bmatrix} 20 \\ 15 \end{bmatrix} \qquad \boldsymbol{A} = \begin{bmatrix} 3 & 2 \\ 10 & 15 \end{bmatrix}$$

We have the following LP:

$$\begin{aligned} \text{maximize} \quad & 20h + 15s && && \text{(Revenue)} \\ \text{subject to} \quad & 3h + 2s && \leq 176 && \text{(Time constraint)} \\ & 10h + 15s && \leq 1000 && \text{(Yarn constraint)} \\ \text{and} \quad & h, s && \geq 0 && \text{(Cannot produce negative hats or scarves)} \end{aligned}$$

The fundamental theorem of linear programming states that both the maximum and minimum of a linear function constrained to a convex polytope occur at a corner of the

polytope. Figure 4.2 shows the plot of the constraints of our example. The corners of the polytope occur at $(0,0), (0,66\frac{2}{3}), (23\frac{3}{5}, 49\frac{3}{5})$ and $(58\frac{2}{3}, 0)$. To find the maximum, we plug the coordinates of the four corners into our maximization function $20h + 15s$ to obtain, 0, 1000, 1256 and 1073. Since $(25\frac{3}{5}, 49\frac{3}{5})$ has the greatest value, producing $25\frac{3}{5}$ hats and $44\frac{3}{5}$ scarves gives the most revenue. Unfortunately, she cannot sell $\frac{3}{5}$ of a hat. We will deal with this problem in the next section.

Figure 4.3 Shows the same feasible region but also plots the maximization functions which intersect the corners. The maximization function is $20h+15s = Revenue$. The goal of linear programming is to find the greatest revenue line that intersects the feasible region. For this problem, the line $20h + 15s = 1256$ intersects the feasible region at the point $(25\frac{3}{5}, 44\frac{3}{5})$.

For such a small problem, it is easy to graph the problem and solve it using simple algebra. However, with more variables, the problem becomes impossible to visualize and there can be a huge number of corners of a high-dimensional polytope. Nontheless, there are a number of methods for solving the LP in polynomial time.

The most common method is the simplex method which is not guaranteed to run in polynomial time in the worst case but is usually polynomial in practice. While the details of the algorithm are beyond the scope of this thesis, the general idea is that the algorithm starts by examining a corner on the convex polytope. For a maximization problem, it then traverses the adjacent edge in the convex polytope to the neighboring corner with the largest objective value. This operation is called a pivot and is performed by removing one variable from the basis and adding a different variable back. Since a maximization function over a convex polytope has no local maxima, only global, it can keep traversing edges this way until all neighbors have a smaller objective value. This point is guaranteed to be the global maximum. We refer the reader to [Chv83] for more information on the simplex algorithm as well as other linear programming algorithms.

Given a vector of variables $\boldsymbol{x} = \{x_1, x_2, ..., x_n\}$; two vectors of coefficients $\boldsymbol{b} = \{c_1, c_2, ..., b_n\}$

and $\boldsymbol{c} = \{c_1, c_2, ..., c_n\}$; and a matrix of coefficients $\boldsymbol{A}$, the standard form of an LP is:

$$\begin{aligned} \text{maximize:} \quad & \boldsymbol{c}^T\boldsymbol{x} \\ \text{subject to:} \quad & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \\ \text{and:} \quad & \boldsymbol{x} \geq 0 \end{aligned}$$

The values of $\boldsymbol{A}$, $\boldsymbol{b}$ and $\boldsymbol{c}$ are all given in the problem statement. The goal is to determine the values of the elements of $\boldsymbol{x}$ that satisfy all equations in $\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$ while maximizing the formula $\boldsymbol{c}^{\boldsymbol{T}}\boldsymbol{x}$.

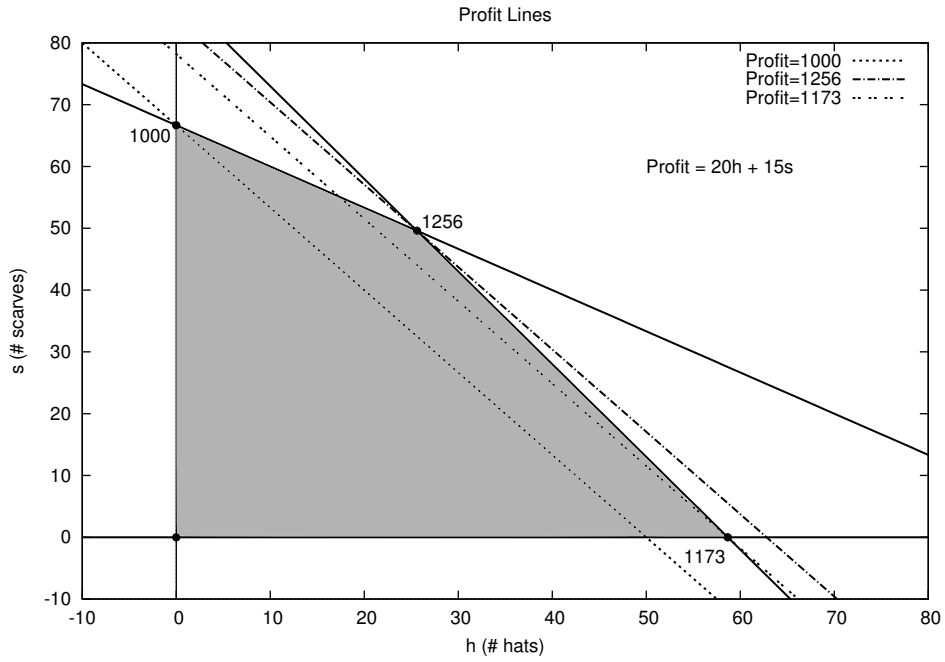


Figure 4.3: The same feasible region as Figure 4.2 with three revenue lines drawn for revenues 1000, 1173 and 1256 which intersect with the non-trivial corners of the feasible region.

4.6.2 Branch-and-Bound

Integer linear programming (ILP) is similar to linear programming, the difference being that the variables are constrained to having integer values. While linear programming is solvable

in polynomial time, ILP is NP-Complete, and no polynomial time algorithm is known [GJ79]. Branch-and-bound can be combined with linear programming to solve an ILP.

If we plug the ILP into an LP solver while relaxing the integrality constraint and the value of the variables returned happen to be integer, that solution is guaranteed to be optimal even for the ILP. However, if the solution returned has even one non-integer variable, it is not a feasible solution to the ILP.

However, since we know that no variable can be non-integral, we can add a constraint to rule out the non-integral solution and solve the new program. For example, in the example from the last section, the solution was $h = 25\frac{3}{5}, s = 49\frac{3}{5}$. Let's say we are now searching for an integer solution to the problem. We create two new linear programs by adding the constraint $h \leq 25$ to one program and $h \geq 26$ to the other. Since the value of h cannot be between 25 and 26 and also be integral, this does not rule out any feasible solutions to the integer linear program. The solution to the first problem is $h = 25, s = 50$ with objective value 1250. The solution to the second problem is $h = 26, s = 49$ with objective value 1255. Since this is a maximization problem, the solution is $h = 26, s = 49$.

4.6.3 Cutting Planes

[Gom58] introduced the idea of cutting planes in his short 1958 paper. A cutting plane is a constraint added to a linear program that shrinks the size of the feasible region but does not exclude any feasible integer solutions.

The problem of finding valid cutting planes is called the separation problem. Although Gomory described cutting planes in the 1950s, he did not believe they would be a useful technique. It was not until the 1990s that [BCC93] figured out how to efficiently use cutting planes to improve upon branch-and-bound techniques for ILP. Gomory cuts are ubiquitous since they work well in concert with the simplex algorithm. However, there are many different separation algorithms. The description of these algorithms is beyond the scope of this thesis.

4.6.4 Column Generation

Column generation is a very complex subject beyond the scope of this thesis. We will give just a very high level view. In section 4.6.1, we discussed the simplex algorithm. At the most abstract level of explanation, the simplex algorithm traverses the corners of the convex polytope enclosing the feasible region of the problem. Due to the nature of simplex, at any corner, only a subset of the variables have non-zero values and are part of the solution. These variables are called basic variables while the zero variables are called nonbasic. The set of all basic variables is called the basis. At each pivot step, one of the nonbasic variables replaces one of the basic variables in the basis to traverse an edge to a neighboring corner.

What happens if we have a problem with an exponential number of nonbasic variables? If we used the simplex method, finding the nonbasic variable to enter the basis in order to find a neighbor with a better objective value would be too costly as we would have to examine the cost of each of these exponentially many variables entering the basis. In these situations, we use column generation [DW60, AC05].

Instead of considering all variables for the simplex algorithm, only a subset of variables are used. This subset is called the master problem. We first solve the master problem optimally. Then, an optimization problem called the pricing problem involving the dual of the LP solution is solved to find if there is a variable that can enter the master problem possibly resulting in a better solution. If there is, this new LP is solved optimally. This process continues until no new variable can improve the objective function. At this point, the problem is solved and the solution to this final master problem is the optimal solution.

4.6.5 The Cutting Stock Problem

The cutting stock problem is closely related to the bin packing problem, the difference being in spirit. The bin packing problem assumes that there are relatively few duplicate numbers in the input set S while the cutting stock problem expects many duplicates.

The story behind the problems is different though this has no effect on solution tech-

niques. The bin-packing problem packs numbers into the fewest number of bins of capacity C possible. An example cutting stock problem starts with a set of rolls of paper of fixed length L. There are orders of a certain number of sheets of paper of varying lengths, all less than L. These orders are fulfilled by cutting the large rolls of length L into the sizes requested. The goal is to cut the large rolls such that all the orders are fulfilled and the waste is minimized.

For example, in a typical problem, we might have a large number of rolls of length 20". We need to cut sheets of sizes 6",7",8" and 9" out of the 20" rolls. We need 325 6" sheets, 428 7" sheets, 512 8" sheets and 231 9" sheets. Cut the 20" rolls into sheets of size 6", 7", 8" and 9" such that all orders are fulfilled and the total number of rolls used is minimized.

The solution to the cutting stock problem consists of defining cutting patterns and then specifying how many cuts of each pattern need to be made so that all of the required sheets are obtained. A cutting pattern is a set of sheets that can be cut from one roll. The goal is to minimize the total number of rolls used, or equivalently, the waste. In the example above, some possible patterns are {6",6",6"}, {9",6"} and {7",7",6"}. Note that all of the patterns can be cut from one sheet, and the waste per use of our example cutting patterns are 2", 5", and 0" respectively.

Historically, BCP algorithms have been used to solve the cutting stock problem [GG61] while heuristic search methods have been used to solve bin-packing problems [MT90a, Kor03]. However, it is clear that the bin-packing problem is the cutting stock problem where the number of items of each size happens to be very small, typically one.

More recently, BCP algorithms have also been used to solve the bin packing problem [BS06, Van99]. We will discuss these models in the next section.

4.6.6 Branch and Cut and Price for Bin Packing

Gilmore and Gomory proposed the first linear programming model for the cutting stock problem [GG61]. Given that there are m possible cutting patterns and n different lengths

of paper l_i that are being ordered:

$$\begin{aligned}
\text{Minimize: } & \sum_{j=1}^{m} x_j & \\
\text{Subject to: } & \sum_{j=1}^{m} a_{ij} x_j \geq N_i & \forall i \in [1, n] \\
& a_{ij} \text{ is integer} & \forall i \in [1, n], j \in [1, m] \\
& x_j \text{ is integer} & j \in [1, m]
\end{aligned}$$

Where:

- x_j is the number of times the j^{th} cutting pattern is used.
- N_i is the number of rolls of length l_i demanded.
- a_{ij} is the number of rolls of length l_i produced each time the j^{th} pattern is used.

This formulation works as is for the bin-packing problem as well if we reinterpret the meaning of the variables:

- x_j is a 0/1 variable which is 1 if a particular feasible packing of a single bin is used, and 0 otherwise.
- N_i is usually 1 since there tends to be a single copy of each item with bin packing, though some duplicates may occur.
- a_{ij} is the number of times item i appears in bin j. This is again usually 0 or 1.

To the best of our knowledge, [BS06] represents the state of the art solver for bin packing using branch-and-cut-and-price. They use the Gilmore and Gomory model [GG61] as the basis for their linear program. The details of this algorithm are beyond the scope of this paper so we direct the reader to the original paper for details.

$k \rightarrow$	6-Way			7-Way			8-Way			9-Way		
$n \downarrow$	IBC	BCP	R	IBC	BCP	R	IBC	BCP	R	IBC	BCP	R
30	.20	2.62	13	.19	1.09	6	.18	.58	3	.12	.32	3
31	.28	3.87	14	.27	1.51	6	.41	.72	2	.20	.40	2
32	.44	6.55	15	.38	2.25	6	.39	1.19	3	.32	.61	2
33	.73	12.4	17	.59	3.41	6	.60	1.53	3	.50	.86	2
34	1.05	19.2	18	.87	4.40	5	1.95	2.13	1	.84	1.15	1
35	1.97	44.5	23	1.62	8.31	5	1.74	3.19	2	1.43	1.41	1
36	2.73	76.8	28	2.22	13.7	6	2.05	5.21	3	1.77	2.05	1
37	4.44	152	34	3.96	23.2	6	3.20	6.92	2	3.38	3.20	1
38	7.45	342	46	6.43	41.5	6	5.60	11.4	2	6.52	5.07	1
39	13.8	891	64	10.6	73.1	7	10.0	19.7	2	8.04	6.68	1
40	18.0	2623	146	15.1	126	8	12.6	26.9	2	12.5	11.4	1
41	34.7	8922	257	24.5	252	10	25.8	57.9	2	22.6	16.8	1
42	63.6	13632	214	44.3	552	12	40.9	68.2	2	33.2	22.2	1
43	95.4	14701	154	70.2	1269	18	65.9	136	2	56.3	35.0	1/2
44	156	14717	95	125	4123	33	94.8	234	2	89.3	52.3	1/2
45	253	16689	66	225	10134	45	165	420	3	135	99.7	1

$k \rightarrow$	10-Way			11-Way			12-Way		
$n \downarrow$	IBC	BCP	R	IBC	BCP	R	IBC	BCP	R
30	.07	.20	3	.02	.15	8	.00	.11	50
31	.10	.24	2	.06	.17	3	.01	.13	15
32	.21	.34	2	.11	.21	2	.02	.16	8
33	.37	.48	1	.19	.28	1	.05	.18	3
34	.68	.56	1	.39	.30	1	.13	.21	2
35	1.07	.77	1	.74	.50	1	.26	.25	1
36	1.31	1.02	1	1.27	.64	1/2	.70	.31	1/2
37	3.09	1.67	1/2	1.96	.81	1/2	1.05	.45	1/2
38	5.46	1.93	1/3	3.54	1.24	1/3	2.45	.55	1/4
39	7.77	3.38	1/2	6.56	1.62	1/4	3.99	.90	1/4
40	13.3	4.37	1/3	11.6	2.70	1/4	10.1	1.10	1/9
41	20.3	7.14	1/3	18.5	3.32	1/6	12.2	1.52	1/8
42	30.8	9.87	1/3	34.0	4.62	1/7	24.6	2.33	1/11
43	50.1	12.1	1/4	66.8	6.29	1/11	47.8	3.35	1/14
44	91.4	21.8	1/4	97.3	8.74	1/11	89.9	4.27	1/21
45	154	38.2	1/4	114	16.9	1/7	123	6.27	1/20

Table 4.3: The average time in seconds to optimally partition 48-bit integers 3 through 12 ways using BSIBC and BSBCP.

4.6.7 An Integer Linear Program for Multi-Way Number Partitioning

Moffitt presented an integer linear program (ILP) for solving multi-way number partitioning directly without using MULTIFIT [Mof13]. We are given input set $S = \{s_1, s_2, ..., s_n\}$ to be partitioned k ways. The integer linear program creates an indicator variable v_{ij} which is 1 if integer s_i is assigned to subset j and 0 otherwise.

$$\text{Minimize: } ub \tag{4.1}$$

$$\text{Subject to: } \sum_{i=1}^{n} s_i \times v_{ij} \leq ub \qquad \forall j \in [1, k] \tag{4.2}$$

$$\sum_{j=1}^{k} v_{ij} = 1 \qquad \forall i \in [1, n] \tag{4.3}$$

$$v_{ij} \in \{0, 1\} \text{ and integer} \qquad \forall i \in [1, n], \forall j \in [1, k] \tag{4.4}$$

Objective function 4.1 tells the ILP to minimize the upper bound. Constraint 4.2 enforces that the sum of all integers s_i assigned to subset j is less than or equal to ub. Constraint 4.3 enforces that each input integer must be assigned to exactly one subset. Constraint 4.4 enforces that the indicator variables are integer and constrained to either 0 or 1.

Moffitt reported that he ran experiments using this ILP with the IBM cplex linear program solver [CPL09]. The results were not competitive with IRNP or MOF.

4.7 Experimental Results: BSIBC vs BSBCP

In order to show the relative performance of BSIBC vs BSBCP (using the BCP solver from [BS06]) empirically, we ran these algorithms on the same dataset used to test IRNP, MOF and SNP in chapter 3. Again, this dataset is composed of problem instances with integers sampled uniformly at random from the range $[1, 2^{48} - 1]$. There are 100 problem instances for each n from $n = 30$ to 45. All experiments were run on an Intel Xeon X5680 CPU running at 3.33GHz.

Table 4.3 reports the average run times for BSIBC and BSBCP to partition the input sets of size $n = 30$ to 45 into $k = 6$ to 12 subsets. For each k, the column titled "R" reports the ratio of the average run times of BSBCP to BSIBC. That is, how many times longer on average it takes BSBCP to solve a problem instance as compared to BSIBC.

For $6 \leq k \leq 8$, BSIBC outperforms BSBCP. For $k = 6$, the ratio of run times of BSIBC to BSBCP increases with increasing n up to $n = 41$, but then it reverses and the ratio starts decreasing again. For $k = 7$, the ratio keeps increasing though we suspect that with large enough n, BSBCP would eventually catch BSIBC. There is no obvious trend for $k = 8$. For $k > 8$, BSIBC is faster for small n but then BSBCP eventually becomes the dominant algorithm as n increases.

In 2013, when BSIBC and BSBCP were introduced, the state-of-the-art algorithm for multi-way number partitioning was IRNP. For $k \geq 8$, both BSIBC and BSBCP outperform IRNP. However, at the same conference in which these algorithms were presented, IJCAI-13, MOF was also introduced. MOF outperforms both of these algorithms for all reported values between $k = 6$ and $k = 10$. See table 3.3 for the results of MOF on the same dataset.

4.8 Summary

This chapter has discussed the dual relationship between bin packing and multi-way number partitioning. Using the MULTIFIT algorithm, any bin-packing algorithm can be used to solve number-partitioning problems. Two lower bounds are introduced. The L_1 lower bound which simply sums the input integers and divides by the bin capacity. The L_2 lower bound, also called the wasted space heuristic, calculates an amount of space that must be wasted in any packing and uses this space to calculate a potentially larger lower bound. Two upper bounds are also discussed. First-fit decreasing and best-fit decreasing are both greedy algorithms which give approximate solutions to the bin-packing problem.

MULTIFIT performs a binary search between the calculated lower and upper bounds on bin capacity. For each probe, it sets the bin capacity and solves a bin-packing problem given

the capacity of the probe. If it is possible to pack the input integers into no more than k subsets, a smaller capacity is tried, otherwise a larger capacity is tried. MULTIFIT continues until the smallest capacity that allows the input integers to be packed into k subsets is found.

MULTIFIT can use any bin-packing algorithm to solve the bin-packing instances. We have covered two such algorithms. Improved bin completion is an artificial intelligence algorithm that uses branch-and-bound to solve bin-packing problems. Branch-and-cut-and-price is an operations research algorithm which at its core is a linear programming algorithm. It also uses branch-and-bound to guarantee integer solutions, cutting planes to reduce the feasible region, and column generation to deal with an exponential number of variables. MULTIFIT using these two bin-packing algorithms is called binary-search improved bin completion (BSIBC) and binary-search branch-and-cut-and-price (BSBCP).

At the time these algorithms were published, IRNP was the state-of-the-art algorithm. Both BSIBC and BSBCP outperform IRNP for $k \geq 8$. However, for most values of n and k, these algorithms are not competitive with algorithms which have been created since BSIBC and BSBCP were published, namely SNP and MOF. They are also dominated by cached iterative weakening, which is the subject of the next chapter.

CHAPTER 5

Cached Iterative Weakening

Previous work on multi-way number partitioning fits into one of two classes. Recursive number partitioning (section 3.4), Moffitt partitioning (section 3.5), and sequential number partitioning (section 3.7) are all branch-and-bound algorithms. Binary-search improved bin completion (sections 4.4 and 4.5) and binary-search branch-and-cut-and-price (sections 4.4 and 4.6) both use the MULTIFIT algorithm along with bin-packing algorithms to solve number partitioning. These are all anytime algorithms that start with an approximate partition and then improve it until the best partition is found and proved optimal. In contrast, cached iterative weakening (CIW), our current state-of-the-art algorithm for optimal multi-way number partitioning, starts with a lower bound and iteratively increases it until an optimal partition is found. The first complete partition found is optimal.

Call C^* the largest subset sum of an optimal partition for a particular number-partitioning instance. While searching for C^*, the branch-and-bound algorithms start with an approximation ub such as that returned by the KK heuristic (section 3.1.2), which is typically larger than C^*. They then search for better partitions until they find one with cost C^*. At this point, they need to verify it is optimal by proving there is no partition with all subset sums less than C^*. In contrast, CIW only considers partitions with cost less than or equal to C^*.

For each partial partition, the previous algorithms generate the next subsets using exponential algorithms. RNP uses ESS to generate the next subsets, MOF uses IE, and SNP uses both depending on the situation. In contrast, CIW generates complete subsets only once using ESS and caches them before performing its recursive partitioning.

5.1 Iterative Weakening

CIW begins by calculating the cost of a perfect partition $C_P^* = \lceil \text{sum}(S)/k \rceil$, a lower bound on the optimal partition cost. In any partition, there must be at least one subset whose sum is at least as large as C_P^*.

The branch-and-bound algorithms start with an ub and lb. They recursively partition S into k subsets, decreasing ub and increasing ub until the optimal cost C^* is found and subsequently verified. In contrast, CIW has iterative upper and lower bounds which we refer to as ub_{it} and lb_{it}. On the first iteration $it = 1$, CIW sets ub_1 to the smallest existing subset sum greater than or equal to C_P^* and calculates lb_1 based on ub_1. It then tries to recursively partition S into k subsets with sums no greater than ub_1. In subsequent iterations, CIW increases ub_{it} and decreases lb_{it} until it finds $ub_{it} = C^*$, the first value for which a complete partition is possible. This process is called iterative weakening [Pro93]. In order to verify optimality, any optimal algorithm must consider all partial partitions with costs between C_P^* and C^*. Even after a branch-and-bound algorithm finds an optimal partition of cost C^*, it still needs to verify its optimality by proving that there is no partition with cost within the range $[C_P^*, C^* - 1]$. Iterative weakening **only** explores partial partitions with costs between C_P^* and C^*.

Suppose we could efficiently generate subsets one by one in sum order starting with C_P^*. CIW iteratively chooses each of these subsets as the first subset S_1 of a partial partition. It sets ub_{it} to $\text{sum}(S_1)$ and lb_{it} to $\text{sum}(S) - (k-1)(ub_{it})$. Then, given that it can efficiently generate all subsets within the range $[lb_{it}, ub_{it}]$, it determines whether there are $k-1$ of these subsets that are mutually exclusive and contain all the integers in the remaining set of integers $S^R = S - S_1$. If this is possible, ub_{it} is returned as the optimal partition cost C^*. Otherwise, CIW moves onto the subset with the next larger sum. In this scheme, the cost of a partial partition is always the sum of its first subset S_1.

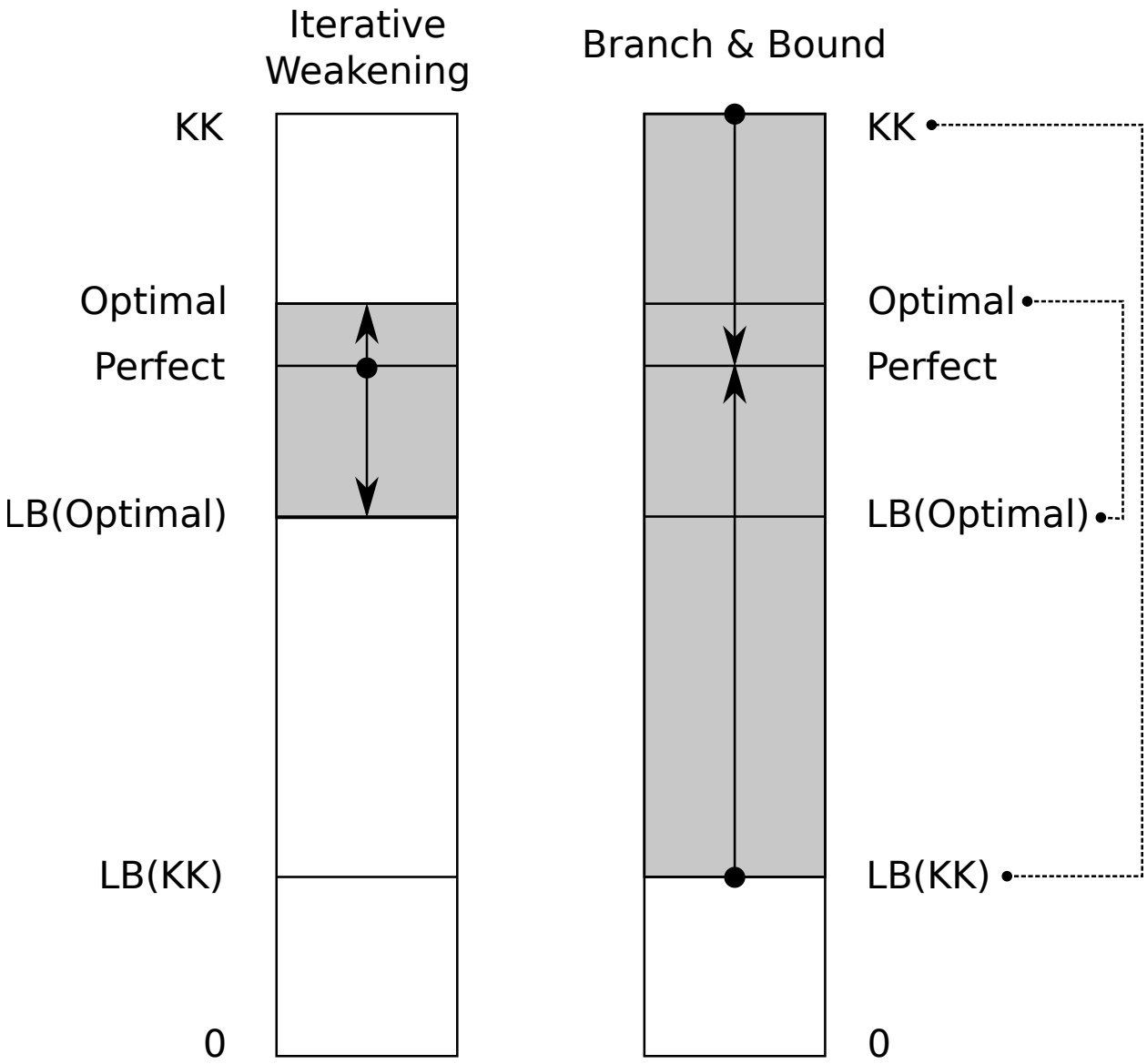


Figure 5.1: A comparison of the search spaces of iterative weakening and branch-and-bound. Iterative weakening starts by theorizing a perfect partition and increases this upper bound while decreasing the implied lower bound until an optimal partition is found. Branch-and-bound calculates an upper bound with an approximation algorithm such as KK. It then refines this bound while increasing the implied lower bound until an optimal partition is found and proved optimal.

The labels on the number line from top to bottom are an upper bound approximation such as the KK heuristic, the cost of an optimal partition, the cost of a perfect partition, the lower bound implied by the optimal partition, and the lower bound implied by the upper bound approximation.

5.2 Precomputing: Generating Subsets in Sum Order

We are not aware of an efficient algorithm for generating subsets in order of their sums. Instead, we describe an algorithm for efficiently generating and storing the m subsets with the smallest sums greater than or equal to C_P^*, the perfect subset sum.

Call max the m^{th} smallest subset sum greater than or equal to C_P^*. The minimum sum of any subset in a partition of cost max is $min = \text{sum}(S) - (k-1)(max)$. CIW generates all subsets with sums within the range $[min, max]$, which includes m subsets with sums within the range $[C_P^*, max]$ and all subsets with sums within the range $[min, C_P^* - 1]$.

Section 3.3 described three algorithms for generating subsets with sums within a given range: IE, EHS and ESS. We wish to generate all subsets with sums within the range $[min, max]$. Unfortunately, we do not know the values of min and max before generating the m subsets with sums greater than or equal to C_P^*.

In order to generate all subsets with sums within the range $[min, max]$, we use a min-heap and a max-heap. We initially set max to the KK ub and min to the corresponding lb. We then generate subsets with sums in this range. We could use any algorithm from section 3.3 for this purpose. We put each subset found with sum within the range $[min, C_P^* - 1]$ into the min-heap and those in the range $[C_P^*, max]$ into the max-heap. This continues until the max-heap contains m subsets. At this point, we reset max to the sum of the largest subset in the max-heap and recalculate min as $\text{sum}(S) - (k-1)(max)$. We then pop all subsets with sums less than min from the min-heap.

We now continue searching for all subsets with sums in the new range $[min, max]$. Each time a subset with sum greater than or equal to C_P^* but less than max is found, we pop the top subset from the max-heap and push the new subset onto the heap. max is set to the new max sum and min is updated accordingly, popping all subsets with sum less than min from the min-heap. When this search is complete, the subsets from the min-heap and the max-heap are moved to a single array sorted by subset sum.

After this is done, iterative weakening iterates through this array one by one in sum order

starting with the subset with smallest sum no less than C_P^*. If m iterations are performed without finding an optimal partition, the algorithm is run again from scratch to generate the next $2m$ subsets greater than or equal to C_P^*. If the $2m$ subsets are exhausted without finding an optimal partition, then the next $4m$ subsets are generated, then the next $8m$, etc. Thus, m is a parameter of CIW. We discuss setting m experimentally in section 5.6.

ESS has a run time which is the square root of the run time of IE and memory usage which is the square root of the memory usage of EHS. Therefore, CIW uses ESS to generate the subsets with sums within the range $[min, max]$.

Example 5.2.1 - Generating Subsets in Sum Order

Consider the example number-partitioning problem with $S = \{127, 125, 122, 105, 87, 75, 68, 64, 30, 22\}$ and $k = 4$. Both figures 5.2 and 5.3 show the array of 19 sets generated by modified ESS with $m = 6$ in the table at the bottom left. The final range $[min, max]$ is $[192, 211]$ and $C_P^* = 207$. There are 13 sets with sums within the range $[min, C_P^* - 1]$ shown below the horizontal line and 6 sets with sums within the range $[C_P^*, max]$ shown above the horizontal line. The 6 subset sums above the horizontal line are the candidate first subsets that CIW iterates over.

The last column of the table is called *Iter*. This column corresponds to the iteration in which the sum of the subset in that row first appears within the range $[lb_{it}, ub_{it}]$. For example, in the first iteration, $ub_1 = 207$ and $lb_1 = 825 - 3 \times 207 = 204$ with two subsets with sums within the range. In the second iteration, $ub_2 = 208$ and $lb_2 = 825 - 3 \times 208 = 201$ with seven sets in range, namely the rows with *Iter* equal to 1 or 2. In the sixth iteration, all 19 sets in the table are in range.

5.3 Recursive Partitioning

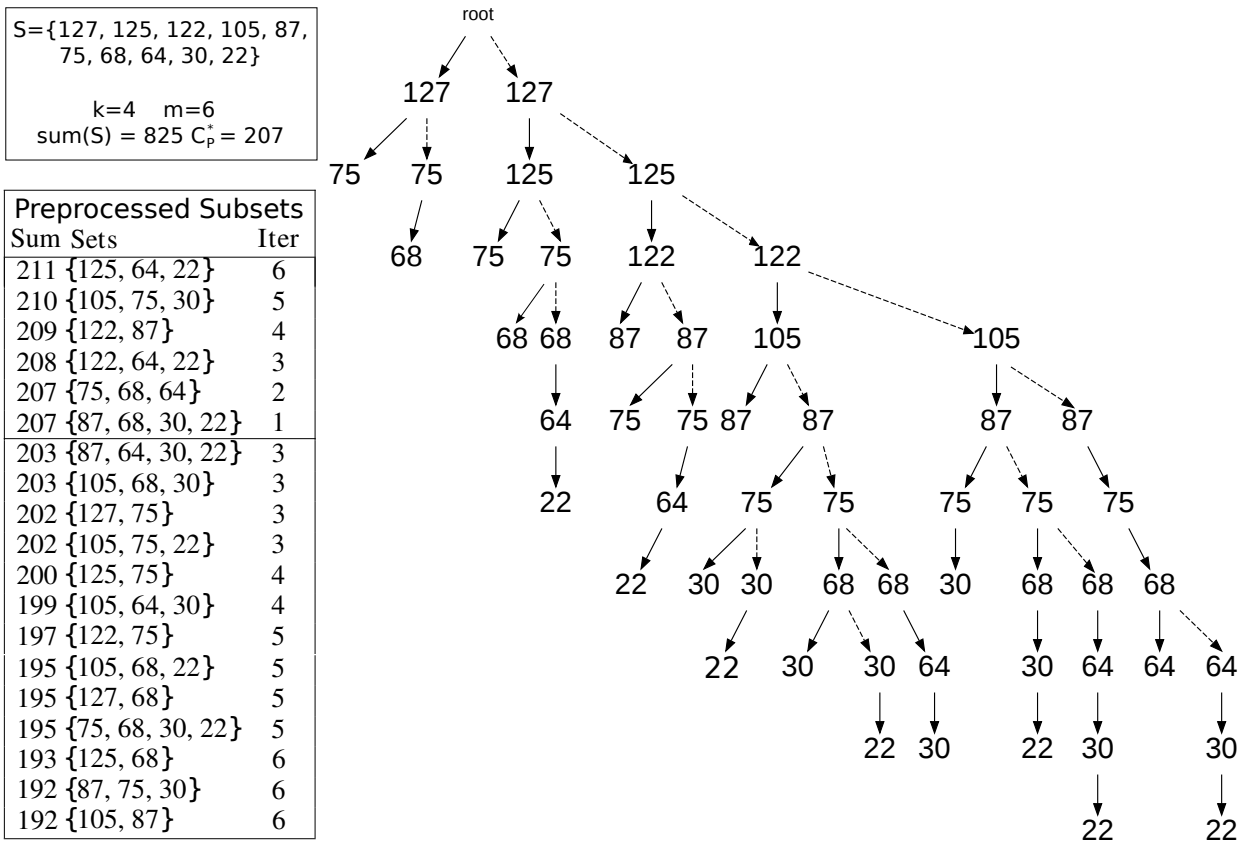
At iteration it, CIW chooses the first subset S_1 as the next subset with sum at least as large as C_P^* from the stored array of subsets, sets $ub_{it} = \text{sum}(S_1)$ and $lb_{it} = \text{sum}(S) - (k-1)(ub_{it})$. This guarantees that S_1 always has the largest sum in any partition. At this point, CIW attempts to recursively partition $S^R = S - S_1$ into $\langle S_2, ..., S_k \rangle$.

This task is very similar to the search of the recursive partitioning tree used by MOF, described in section 3.5.2. The difference is that CIW is searching for any complete partition since it is guaranteed to be optimal, while MOF must search for the lowest cost complete partition. In fact, the MOF algorithm could be used to partition S^R into $k-1$ subsets. Simply set $lb_{it} = sum(S_1)$ and $ub_{it} = lb + 1$. If MOF returns lb_{it}, then a complete partition was found, otherwise there is no complete partition.

However, using MOF to partition S^R into $k-1$ subsets does not take advantage of the fact that all of the subsets with sums in the range $[lb_{it}, ub_{it}]$ have already been generated. Instead, MOF repeatedly calls IE, an exponential time algorithm, to generate complete subsets at each node of the recursive partitioning tree. We next discuss how to leverage the precomputed subsets having sums in the range $[lb_{it}, ub_{it}]$ to determine if a complete partition of cost ub_{it} exists.

5.3.1 A Simple but Inefficient Algorithm

We start with a simple algorithm to motivate the discussion. Given an array A of subsets, we present a recursive algorithm for determining whether there are k mutually exclusive subsets which contain all the integers of S. For each first subset S_1 in A, copy all subsets of A that do not contain an integer in S_1 into a new array B. Then, recursively try to select $k-1$ disjoint subsets from B which contain the remaining integers of $S - S_1$. If $k = 0$, return true, else if the input array A is empty, return false. While this algorithm is correct, it is inefficient, as the entire remaining input array must be scanned for each recursive call. We next present a more efficient algorithm which performs the same function.



Preprocessed Subsets

Sum	Sets	Iter
211	{125, 64, 22}	6
210	{105, 75, 30}	5
209	{122, 87}	4
208	{122, 64, 22}	3
207	{75, 68, 64}	2
207	{87, 68, 30, 22}	1
203	{87, 64, 30, 22}	3
203	{105, 68, 30}	3
202	{127, 75}	3
202	{105, 75, 22}	3
200	{125, 75}	4
199	{105, 64, 30}	4
197	{122, 75}	5
195	{105, 68, 22}	5
195	{127, 68}	5
195	{75, 68, 30, 22}	5
193	{125, 68}	6
192	{87, 75, 30}	6
192	{105, 87}	6

Figure 5.2: A complete cached inclusion-exclusion tree for the input set S and the bounds $lb_5 = 192, ub_5 = 211$ containing all of the subsets listed in the table on the bottom left.

5.3.2 Simplified Cached Inclusion-Exclusion (CIE) Trees

In this section, we describe a simplified version of cached inclusion-exclusion (CIE) trees to describe conceptually how they work. In the subsequent section, we will describe the CIE trees used by CIW, which are a little more complicated.

Each iterative weakening iteration it corresponds to the collection of all subsets having sums in the range $[lb_{it}, ub_{it}]$. After selecting S_1, the remaining integers $S^R = S - S_1$ need to be partitioned $k-1$ ways. We need to determine if there exists $k-1$ subsets from the collection of all subsets which are mutually exclusive and collectively exhaustive of all integers in S^R. As discussed in section 5.3, we could use the MOF algorithm for this purpose. However, the IE algorithm that MOF uses to generate subsets is inefficient.

CIE trees are similar to IE trees (Section 3.3.1). IE searches an implicit tree that repre-

sents all $2^{|S^R|}$ subsets of the remaining integers S^R. **In contrast, a CIE tree is explicitly stored in memory, and represents only the collection of subsets with sums within the range $[lb_{it}, ub_{it}]$.** The nodes of the CIE tree correspond to one of the integers in S. The integer is included on the left branch and excluded on the right.

Figure 5.3 depicts the CIE tree for iteration five of an example four-way partitioning problem of the set $S = \{127, 125, 122, 105, 87, 75, 68, 64, 30, 22\}$. For iteration five, $S_1 = \{125, 64, 22\}$, $ub_5 = \text{sum}(S_1) = 211$ and $lb_5 = 192$. The list of all 20 subsets of S with sums in the range $[lb_{it}, ub_{it}]$ is shown on the bottom left. The 20 subsets are also stored in the CIE tree shown in the same figure. The solid arrows correspond to inclusion of the integer pointed to while dashed arrows correspond to exclusion. For example, from the root, following the left solid arrow to 127, then the left solid arrow to 75 ($root \rightarrow 127 \rightarrow 75$) corresponds to the subset $\{127, 75\}$. Similarly, $root \dashrightarrow 127 \dashrightarrow 125 \rightarrow 122 \dashrightarrow 87 \dashrightarrow 75 \rightarrow 64 \rightarrow 22$ corresponds to the subset $\{122, 64, 22\}$.

Given this CIE tree, we can use the MOF algorithm as described in section 5.3 to determine if a complete partition of S-S_1=$\{127, 122, 105, 87, 75, 68, 30\}$ into k-1=3 subsets exists. However, instead of IE binary-tree search to find subsets, we search the CIE tree instead.

5.3.3 Cached Inclusion-Exclusion (CIE) Trees

We now discuss the full version of cached inclusion-exclusion (CIE) trees used by CIW. For iteration it, after CIW chooses the first subset S_1 from the precomputed array, it uses CIE to test if there are $k-1$ mutually exclusive subsets which contain all integers in $S^R = S - S_1$. CIE trees store all subsets whose sums are in the range $[lb_{it}, ub_{it}]$ for the current iteration and are built incrementally by inserting all new subsets with sums in the range $[lb_{it}, ub_{it}]$ that are not in the range $[lb_{it-1}, ub_{it-1}]$.

In IE trees, all complete subsets, regardless of cardinality, can be found in one tree. In contrast, there is one CIE tree for each unique cardinality of complete subset. The distribution of the cardinality of the subsets with sums in the range $[lb_{it}, ub_{it}]$ is not even.

The average cardinality of a subset in an optimal partition is n/k. Typically, most subsets in an optimal partition have cardinality close to this average. Yet, there are often many more subsets with higher cardinality than n/k. In section 5.3.4, we will show how to leverage these cardinality trees so CIW never has to examine the higher cardinality subsets. In the example of figure 5.3, there are separate trees for subsets of cardinality 2, 3 and 4 storing all subsets with sums within the range $[192, 211]$ (every subset in iteration 1 through 5).

IE searches an implicit tree, meaning only the recursive stack of IE is stored in memory. In contrast, the entire CIE trees are explicitly stored in memory before they are searched. In each iterative weakening iteration, all subsets with sums within the range $[lb_{it}, ub_{it}]$ are represented in the CIE tree of appropriate cardinality. These subsets were already generated in the precomputing step, so this is a matter of iterating over the array of subsets and adding all subsets with sums in range that were not added in previous iterations.

5.3.4 Recursive Partitioning with CIE Trees

For iteration it, after selecting S_1 and calculating lb_{it} and ub_{it}, CIW adds all subsets with sums newly within the range $[lb_{it}, ub_{it}]$ from the stored array of subsets into the CIE tree of proper cardinality. If CIW finds $k-1$ of these subsets which are mutually exclusive and contain all the integers of $S^R = S - S_1$, then the optimal cost is $ub_{it} = \text{sum}(S_1)$.

Like the standard IE algorithm, CIE searches its trees left to right, including integers before excluding them. However, each node of a CIE tree corresponds to an integer in S and not all of these integers still remain in S^R. At each node of the CIE tree, an integer can only be included if it is a member of S^R, the integers remaining.

Iterative weakening selects the first subset S_1. To generate each possible S_2, CIE searches the tree of smallest cardinality first. Call $card$ the cardinality of the tree CIE is searching to generate S_d in partial partition $\langle S_1, ..., S_d \rangle$. If CIE finds a subset S_d of cardinality $card$, the recursive search begins searching for subset S_{d+1} in the $card$ CIE tree. If no more subsets are found in the $card$ CIE tree, the $card + 1$ CIE tree is searched until no higher cardinality

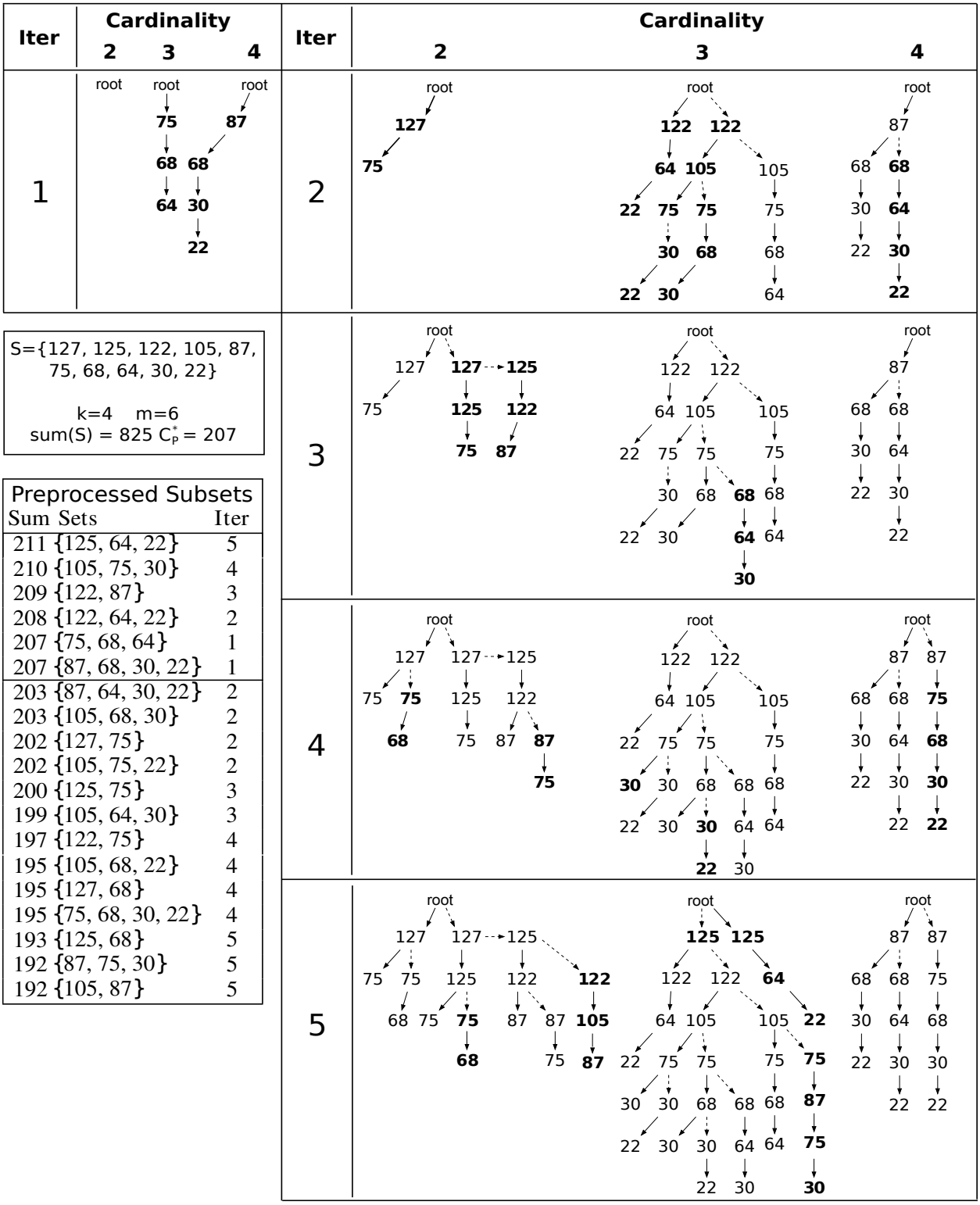


Figure 5.3: CIW example with the list of preprocessed subsets sorted by sum and cached inclusion-exclusion trees for cardinalities 2, 3 and 4 during iterations 1,2,3,4 & 5. The bold numbers were added during that iteration.

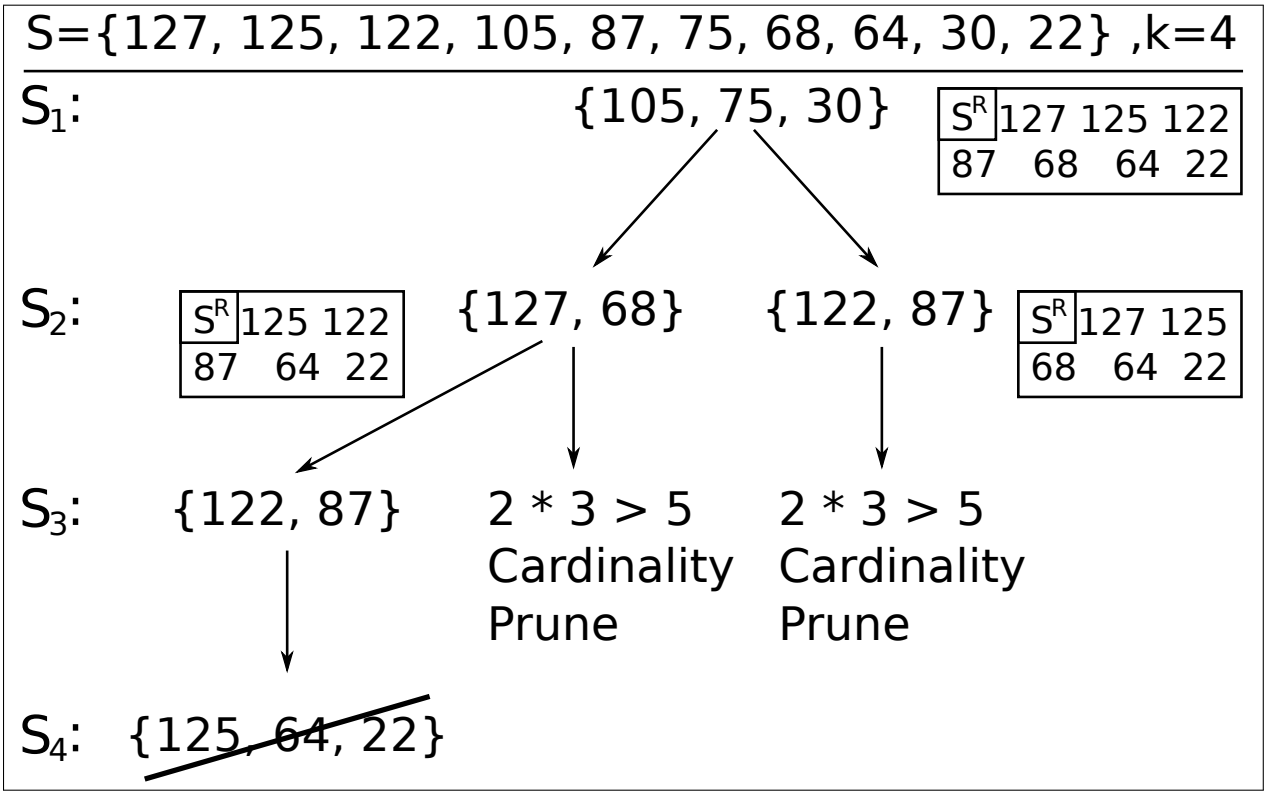


Figure 5.4: The recursive partitioning search tree for iteration 4.

CIE tree exists. CIW can prune if $k - d \times card > |S^R|$ since there would not be enough integers left in S^R to create $k - d$ subsets, each with cardinality $\geq card$.

5.3.5 Avoiding Duplicates

Choosing subsets in cardinality order avoids many duplicates. However, if S_d and S_{d+1} in partial partition P_{d+1} have the same cardinality, in order to remove duplicates, the largest integer in S_{d+1} must be smaller than the largest integer in S_d. For example, CIE generates the partition $\langle\{8,1\},\{6,3\},\{5,2,2\}\rangle$ and not $\langle\{6,3\},\{8,1\},\{5,2,2\}\rangle$ since the '8' in $\{8,1\}$ is larger than the '6' in $\{6,3\}$.

5.4 Example: Iteration 4 of Iterative Weakening

Figure 5.4 shows the recursive partitioning search tree for iteration 4 of our running example using the CIE trees from figure 5.3. The root of the tree is the subset $S_1 = \{105, 75, 30\}$ whose sum 210 is the upper bound ub_4 for iteration 4 of iterative weakening. The search of candidate subsets for S_2 begins in the cardinality 2 CIE tree of figure 5.3 (there are no singletons subsets with sums within the bounds), including nodes before excluding them. Starting from the root of the CIE tree, CIW includes 127 but cannot include 75 since it is included in S_1. It excludes 75 and includes 68 giving us $S_2 = \{127, 68\}$. We continue to search the cardinality 2 CIE tree for S_3 but the largest integer must be less than 127 to avoid duplicates, so we exclude 127 and then include 125. We cannot include 75 since it is in S_1, so we backtrack to exclude 125 and include 122 and then 87 giving us the 3rd subset $S_3 = \{122, 87\}$. Since there is only S_4 left, we can put all remaining integers into S_4, but the sum of the remaining integers 125+64+22 = 211 is greater than the ub_4, so we prune. (Note that the set {125, 64, 22} is not in the cardinality 3 CIE tree.) We backtrack to generate the next S_3 subset. Continuing where we left off in the cardinality 2 tree when we generated $S_3 = \{122, 87\}$, we backtrack to exclude 87 but since 75 is not in S^R, we have exhaustively searched the cardinality 2 CIE tree. We move to the cardinality 3 tree. However, there are five integers left to partition into two subsets and the cardinality of the subsets left must be three or greater. Since $2 \times 3 > 5$, we prune.

We now backtrack to generate the next S_2 subset continuing where we left off in the cardinality 2 cached-IE tree when we generated $\{127, 68\}$. Since the first child was $\{127, 68\}$, and there are no more subsets containing 127 we backtrack to exclude 127. We then include 125 but 75 is not in S^R, so we backtrack and exclude 125. We then include 122 and 87, giving us $S_2 = \{122, 87\}$. We continue to search the cardinality 2 tree for S_3 but the largest integer must now be less than 122 to avoid duplicates. We exclude 127 and 125, but there are no more exclusion branches so we move to the cardinality 3 tree. Again, we can prune since $2 \times 3 > 5$ and thus there is no optimal partition of cost 210.

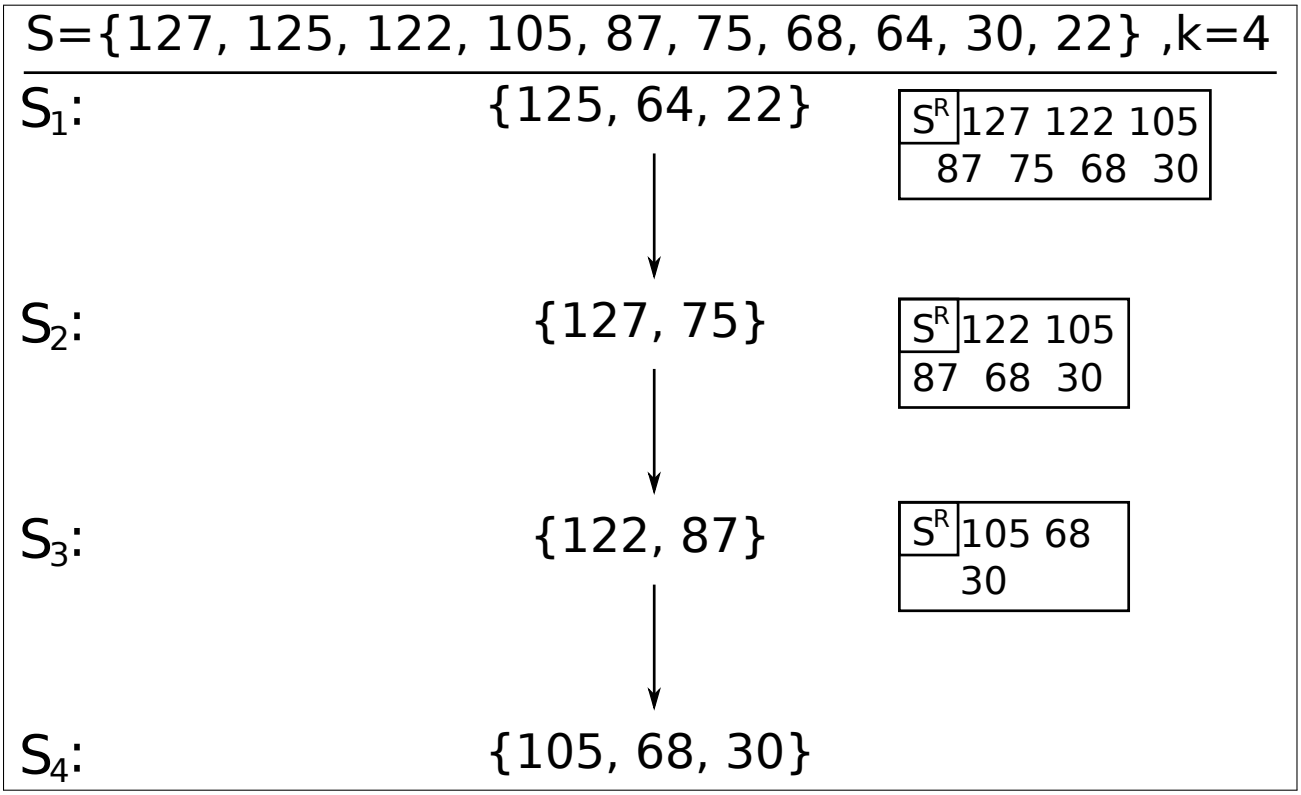


Figure 5.5: The recursive partitioning search tree for iteration 5.

5.5 Example: Iteration 5 of Iterative Weakening

In iteration 5, iterative weakening sets $S_1 = \{125, 64, 22\}$ with $ub_5 = \text{sum}(S_1) = 211$ and adds the four rows with $Iter = 5$ from the table in figure 5.3 to the CIE trees. Figure 5.5 shows the recursive partitioning search tree for iteration 5. CIW partitions S into $k = 4$ subsets all with sums less than 211 without having to backtrack, resulting in the optimal partition $\langle\{125, 64, 22\}, \{127, 75\}, \{122, 87\}, \{105, 68, 30\}\rangle$.

5.6 Experimental Results: CIW

In order to show the relative performance of CIW to the previous state-of-the-art algorithms empirically, we ran CIW against the same dataset used to test all of the previous algorithms. Again, this dataset is composed of problem instances with integers sampled uniformly at

random from the range $[1, 2^{48} - 1]$. There are 100 problem instances, but this time, because of the improved performance of CIW, n ranges from $n = 40$ to 60 as opposed to $n = 30$ to 45. All experiments were run on an Intel Xeon X5680 CPU running at 3.33GHz.

Generally, for $n = 40$ to 60, SNP (section 3.7) was the previous state of the art for $k = 3$ to 7, the MOF algorithm (section 3.5) for $k = 8$ to 10 and BSBCP (sections 4.4 and 4.6) for $k = 11$ and 12. There are exceptions for $(n \leq 45; k{=}7)$, where MOF is faster than SNP and $(n \leq 46; k{=}11)$, where MOF is faster than BSBCP. We compare CIW to SNP for $k \leq 7$, to MOF for k from 8 to 10 and to BSBCP for $k = 11$ and 12. All experiments were run on an Intel Xeon X5680 CPU running at 3.33GHz.

For a particular k, since the previous state of the art depends on n, one might infer that a hybrid recursive algorithm that uses one of SNP, MOF or BSBCP for recursive calls depending on n and k would outperform the individual algorithms. However, [KSM13] shows that the hybrid algorithm in fact does not significantly outperform the best of the individual algorithms.

For CIW, we need to choose a value for m, the number of subsets to initially generate during the precomputing phase (section 5.2). Ideally, we want m to be exactly the number of subsets with sums within the range $[C_P^*, C^*]$, but we do not know this number in advance. If m is set too small, CIW will have to run ESS multiple times. If m is set too large, CIW will waste time generating subsets in the precomputing step that are never used in the iterative weakening step.

For each combination of n and k, we initially set m to 10,000. Call m_i the number of subsets with sums within the range $[C_P^*, C^*]$ for problem instance i. After instance 1 is complete, m is set to m_1. After instance i is complete, m is set to the max of m_1 through m_i. The values of m used ranged from 24 for the second instance of $(n = 56; k = 3)$ to 180,085 for the last eight instances of $(n = 59; k = 12)$.

Table 5.1 compares CIW to SNP for k from 3 to 12. Each row corresponds to a value of n. There are three columns for each value of k. In the top table for $k = 3$ to 7, the first two

columns report the average time in seconds to partition n integers into k subsets over 100 instances using CIW and SNP respectively. The third column is the ratio of the run time of SNP to CIW.

CIW is faster than SNP for $k \geq 4$ with the exception of $(n \leq 40; k = 4)$. SNP is faster than CIW for $k = 3$. For fixed n, SNP tends to get slower as k gets larger while CIW tends to get faster as k gets larger. For $5 \leq k \leq 7$, the ratios of the run times of SNP to CIW tend to grow as n gets larger, suggesting that CIW is asymptotically faster than SNP. For $k = 3$ and 4, there is no clear trend. The biggest difference in the average run times is for $(n = 58; k = 7)$ where SNP takes 91 times longer than CIW.

The bottom table shows data in the same format as the top table but for k from 8 to 12 and this time comparing CIW to MOF for k from 8 to 10, and CIW to BSBCP for 11 and 12. CIW outperforms both MOF and BSBCP for all n and k. The ratios of the run times of MOF to CIW and BSBCP to CIW grow as n gets larger for all k, again suggesting that CIW is asymptotically faster than MOF and BSBCP. The biggest difference in the average run times of MOF and CIW is for $(n = 60; k = 10)$ where MOF takes 401 times longer than CIW. The biggest difference for BSBCP is for $(n = 60; k = 11)$ where BSBCP takes 250 times longer than CIW.

There is memory overhead for CIW due to both ESS and the CIE trees, proportional to the number of subsets with sums in the range $[C_P^*, C^*]$. All of the experiments require less than 4.5GB of memory and 95% require less than 325MB. However, it is possible that with increased n, memory could become a limiting factor as well as time. Better understanding of and reducing the memory usage is the subject of future work.

For each value of k, we are showing a comparison of CIW to the best of SNP, MOF and BSBCP. If we compared CIW to any of the other two algorithms, the ratio of the run time of the other algorithms to CIW would be even higher, up to multiple orders of magnitude more.

$k \rightarrow$	3-Way			4-Way			5-Way			6-Way			7-Way		
$n \downarrow$	CIW	SNP	R	CIW	SNP	R	CIW	SNP	R	CIW	SNP	R	CIW	SNP	R
40	.21	.07	1/3	.16	.15	1	.17	.36	2	.27	.83	3	.26	1.96	8
41	.28	.11	1/2	.23	.25	1	.23	.59	3	.22	1.29	6	.24	3.02	13
42	.33	.14	1/2	.29	.33	1	.33	.86	3	.32	2.16	7	.34	5.03	15
43	.57	.22	1/3	.47	.51	1	.47	1.27	3	.46	3.17	7	.45	7.47	17
44	.74	.28	1/3	.60	.68	1	.54	1.77	3	.58	4.35	7	.58	11.1	19
45	1.15	.49	1/2	.92	1.13	1	.82	2.64	3	.81	7.07	9	.83	19.5	23
46	1.47	.59	1/2	1.20	1.47	1	1.07	4.27	4	.99	10.9	11	.96	26.7	28
47	2.63	.97	1/3	2.20	2.26	1	1.85	6.30	3	1.69	16.6	10	1.52	41.4	27
48	3.51	1.18	1/3	2.58	3.18	1	2.16	9.30	4	1.92	24.6	13	1.69	62.9	37
49	4.46	2.06	1/2	3.87	5.09	1	3.54	14.0	4	3.09	37.9	12	2.79	92.9	33
50	6.44	2.44	1/3	5.08	6.89	1	4.28	18.6	4	3.45	50.7	15	2.99	135	45
51	11.2	4.10	1/3	9.17	10.7	1	7.54	32.2	4	6.75	89.2	13	5.57	233	42
52	13.9	4.93	1/3	10.7	14.4	1	9.02	44.4	5	7.70	125	16	6.63	364	55
53	25.2	8.81	1/3	17.4	22.6	1	15.2	67.1	4	13.7	199	14	10.9	529	49
54	44.8	10.5	1/4	22.0	31.7	1	18.7	104	6	15.2	309	20	12.7	864	68
55	43.3	17.2	1/3	38.7	48.4	1	34.1	157	5	30.7	472	15	24.9	1328	53
56	70.5	20.7	1/3	51.1	67.8	1	40.7	225	6	35.0	698	20	27.5	1945	71
57	139	36.7	1/4	83.8	103	1	75.7	349	5	61.4	1119	18	47.1	3158	67
58	131	45.4	1/3	105	143	1	82.5	536	6	68.1	1677	25	55.4	5015	91
59	228	73.6	1/3	186	237	1	165	771	5	139	2448	18	109	7520	69
60	304	89.6	1/3	254	307	1	201	1140	6	156	3728	24	123	10715	87

$k \rightarrow$	8-Way			9-Way			10-Way			11-Way			12-Way		
$n \downarrow$	CIW	MOF	R	CIW	MOF	R	CIW	MOF	R	CIW	BCP	R	CIW	BCP	R
40	.23	1.24	5	.35	1.24	4	.51	1.39	3	.49	2.68	5	3.27	1.12	1/3
41	.27	2.29	8	.31	2.11	7	.52	2.04	4	.43	3.29	8	1.98	1.52	1
42	.39	3.77	10	.44	3.26	7	.74	3.27	4	.57	4.54	8	1.60	2.25	1
43	.51	5.89	12	.56	5.44	10	.77	5.26	7	.88	6.22	7	1.23	3.33	3
44	.60	8.60	14	.62	8.63	14	1.00	9.22	9	1.18	8.60	7	1.12	4.28	4
45	.79	14.5	18	.86	13.8	16	1.18	14.0	12	2.00	15.1	8	1.63	6.55	4
46	.92	23.6	26	1.10	24.8	23	1.08	21.9	20	2.43	21.2	9	1.91	8.85	5
47	1.39	38.2	28	1.63	41.4	25	1.63	42.6	26	2.79	30.5	11	3.38	14.7	4
48	1.62	60.0	37	1.78	59.4	33	1.65	56.5	34	3.10	48.9	16	4.72	24.4	5
49	2.40	100	42	2.32	98.0	42	2.25	92.8	41	3.32	65.6	20	7.56	32.4	4
50	2.47	166	67	2.53	154	61	2.62	141	54	3.14	92.8	30	13.9	55.3	4
51	4.54	281	62	3.78	274	72	4.26	263	62	3.91	146	37	13.5	64.3	5
52	5.47	418	77	4.43	382	86	6.02	362	60	4.23	187	44	13.8	85.6	6
53	8.98	707	79	7.39	724	98	8.31	735	88	5.44	273	50	16.2	109	7
54	10.0	1189	119	8.90	1116	125	8.86	1120	126	8.30	445	54	13.6	138	10
55	19.0	2082	110	15.6	1701	109	16.3	1920	118	14.9	721	48	17.4	250	14
56	21.4	3078	144	17.9	2878	160	16.1	2700	167	18.9	1325	70	14.6	315	22
57	36.6	5589	153	26.3	4983	189	19.7	4950	251	25.7	2354	92	16.2	464	29
58	42.0	9182	219	31.2	7636	245	26.0	8508	327	32.8	3962	121	23.8	807	34
59	73.6	13441	183	52.4	11874	227	39.1	12249	313	41.3	7292	177	36.1	1245	35
60	84.9	23085	272	60.3	21414	355	45.0	18036	401	47.8	11971	250	68.2	2225	33

Table 5.1: The average time in seconds to optimally partition 48-bit integers 3 through 12 ways using CIW, SNP, MOF and BSBCP.

5.7 Low Cardinality Search

This section describes low cardinality search (LCS), which combines the best ideas of CIW with branch-and-bound algorithms such as SNP or MOF. We start by describing the weaknesses of the previous algorithms, and then describe how LCS improves upon them.

Please note that work on LCS is preliminary. Papers on all of the other algorithms in this thesis have been peer reviewed and published. This is a new line of research that has not been previously published and thus is still a work in progress.

5.7.1 Weakness of CIW

While performing cached iterative weakening, each subset with a sum in the range $[C_P^*, C^*]$ is considered in ascending sum order as the first subset S_1 of partitions $\langle S_1, ..., S_k \rangle$. These subsets have varying cardinalities. As discussed in section 5.3.3, the distribution of the cardinality of the subsets with sums in the range $[lb_{it}, ub_{it}]$ is not uniform. Consider a partition problem with $n = 50$ integers and $k = 10$ subsets. The average number of integers per subset is $\frac{50}{10} = 5$. Now consider all of the subsets of the input integers S. For example, there are $\binom{50}{5} \approx 2.1 \times 10^6$ subsets of cardinality five and $\binom{50}{10} \approx 1.0 \times 10^{10}$ subsets of cardinality ten. There are approximately 5,000 times more cardinality ten subsets than cardinality five. However, it is unlikely that an optimal partition will contain any cardinality ten subsets since this would require at least half of the subsets to have cardinality less than five, which is the average cardinality. In general, it is harder to find low cardinality subsets with sums in a particular range since there are many fewer such subsets. For our example, there are 9.2 times as many cardinality five subsets as cardinality four subsets for $n = 50$ and $k = 10$ and 108.1 times as many cardinality five subsets as cardinality three. As we choose very high cardinality subsets, the problem gets more constrained and it becomes less likely that there is an optimal partition.

While performing iterative weakening, many of the S_1 subsets whose sum is within the range $[C_P^*, ub_{it}]$ have cardinality much larger than the average subset cardinality, and are thus

unlikely to lead to an optimal partition. Since they are considered at the root of the recursive partitioning tree, it can be expensive to prove there is no optimal partition involving these high cardinality subsets. Avoiding iterative weakening allows us to avoid considering these high cardinality subsets at the root of the recursive partitioning tree and push them down towards the leaves where they often never have to be considered because of the cardinality pruning rule discussed in section 5.3.4.

5.7.2 Weakness of Branch-and-Bound Algorithms

A weakness of branch-and-bound algorithms such as RNP, MOF and SNP is that at every node of the recursive partitioning tree, they have to solve an exponential problem (using either IE or ESS) to find all subsets with sum within the lower and upper bounds. The caching of CIW allows us to run this search for subsets with sums in a range once and then cache them. Searching CIE trees is much faster than using ESS or IE trees to find subsets. This is because CIE trees only contain subsets with sums in the range $[lb_{it}, ub_{it}]$ while the other algorithms must search through the entire search space attempting to prune out subsets with sums out of the range.

5.7.3 Low Cardinality Extended Horowitz and Sahni

The average cardinality of any complete partition of n integers into k subsets is n/k. We define low cardinality as less than or equal to $\lceil n/k \rceil + 1$, one more than the ceiling of this average cardinality. We call this threshold the max cardinality, or MC.

In order to perform low cardinality search, we must be able to generate low cardinality subsets with sums within the range $[lb_{it}, ub_{it}]$. Section 3.3.2 describes the EHS algorithm for generating all subsets with sums within this range. Low Cardinality Extended Horowitz and Sahni (LCEHS) modifies EHS to generate subsets within the range $[lb_{it}, ub_{it}]$, having cardinality less than or equal to MC.

EHS starts by generating two "half sets" S_A and S_B. S_A consists of all $2^{\frac{n}{2}}$ subsets of the

largest $n/2$ numbers in S while S_B consists of all subsets of the smallest $n/2$ numbers in S. LCEHS generates half sets containing only subsets with cardinality less than or equal to MC, having sums less than or equal to ub_{it}. For subsets whose cardinality is equal to MC, their sum must also be greater than or equal to lb_{it} since when the half sets are combined, any subset whose cardinality is MC can only be combined with the empty set.

After generating the low cardinality half sets, the rest of the LCEHS algorithm is almost identical to EHS. The only difference is that when combining a subset from S_A with a subset from S_B, if the cardinality of the union of the two subsets exceeds MC, it is discarded.

5.7.4 The New Algorithm

Low cardinality search (LCS) is a hybrid of CIW and sequential recursive partitioning, the partitioning technique used by SNP and MOF. LCS works in two phases. The first phase is very similar to CIW, but the iterative weakening only considers low cardinality subsets generated using LCEHS. The second phase is a branch-and-bound search that either proves that the partition found in the first phase is optimal, or finds the optimal partition.

Phase One

Phase one of LCS is almost identical to CIW, with two differences. First, as described in section 5.2, CIW uses ESS to generate the m subsets (m is a parameter) with smallest sums that are also greater than or equal to C_P^*. In contrast, LCS generates m subsets as well, but it only generates subsets with cardinality less than or equal to $MC = \lceil n/k \rceil + 1$ using LCEHS. Because LCS only considers low cardinality subsets, for the same dataset, m can be much lower than for CIW since there will be many fewer low cardinality subsets with sums in the range $[C_P^*, C^*]$. The iterative weakening is performed using these low cardinality subsets as the S_1 subset.

The second difference is that instead of storing all generated subsets in CIE trees, only low cardinality subsets are stored. While performing the recursive partitioning as described

in section 5.3.4, CIE trees are used to search for low cardinality subsets (cardinality $\leq MC$) and IE binary tree search is used to search for high cardinality subsets (cardinality $> MC$).

For our example with $n = 50$ and $k = 10$, LCS would first search for cardinality two through six subsets in CIE trees. The subsets are always considered in cardinality order. Consider the partial partition of d subsets with $d < k$, $P = \langle S_1, S_2, ..., S_d \rangle$. If there are no subsets remaining in the CIE trees that are mutually exclusive of $S_1 \cup S_2 \cup ... \cup S_d$, then LCS will attempt to complete the partition using IE to search for $k - d$ subsets with cardinalities greater than or equal to 7. Recall, we can prune if $|S^R| \geq (k - d) \times 7$, that is if the number of integers remaining is greater than or equal to the number of subsets remaining times the minimum cardinality of the remaining subsets. If the IE binary tree search is necessary, the number of integers left in S will be small and so the search will be relatively inexpensive.

Phase one continues iterative weakening until the first complete partition is found. Since phase one considers only low cardinality subsets as the S_1 subset, the first complete partition found is not guaranteed to be optimal as it is with CIW. With LCS, the S_1 subset is always the subset with the greatest sum. If it turns out that the subset with greatest sum in the optimal partition has a high cardinality, then phase one will not find an optimal partition.

Phase Two

After phase one is complete, phase two either proves that the partition found in phase one is optimal, or it finds an optimal partition. Phase two performs a branch-and-bound recursive partitioning using the CIE trees created in phase one. However, unlike phase one, it does not force the S_1 subset using iterative weakening. Instead, the CIE trees are used to generate S_1. Furthermore, the first complete partition is not returned. Instead, the recursive partitioning continues until a partition is found with cost equal to the lower bound, or until the search space is exhausted. Again, like in phase one, when searching for high cardinality subsets, IE binary tree search is used. Since no assumption is made about the cardinality of the subset with largest sum, phase two guarantees an optimal partition.

Discussion

Phase two is typically more computationally expensive than phase one. Since the iterative weakening happens only over low cardinality subsets and the CIE trees contain only low cardinality subsets, phase one is significantly faster than the iterative weakening of CIW.

Phase one accomplishes two tasks. First, it creates a better upper bound for phase two. Since a complete partition is found in phase one, the optimal partition is guaranteed to have cost no greater than the found partition cost. Second, it populates the CIE trees with all low cardinality subsets with sum less than or equal to the upper bound. These CIE trees are therefore ready to use for phase two.

5.7.5 Experimental Results: LCS

In order to show the relative performance of LCS and CIW, we ran LCS against the same dataset used to test all of the previous algorithms. Again, this dataset is composed of problem instances with integers sampled uniformly at random from the range $[1, 2^{48} - 1]$. There are 100 problem instances for each combination of n and k. For $k = 3$ to 6, we ran experiments from $n = 45$ to 60. For $k = 7$ to 10, we ran experiments from $k = 45$ to 70. Experiments that were not complete at the time of publication either due to time or memory constraints are shown with a "-".

Table 5.2 reports the average run times for LCS and CIW to partition the input sets into $k = 3$ to 10 subsets. For each k, the column titled "R" reports the ratio of the average run times of CIW to LCS. That is, it shows how many times longer on average it takes CIW to solve a problem instance compared to LCS.

For $k \leq 4$, CIW outperforms LCS and there is no clear trend for the ratio of the run times of the two algorithms as a function of n. For $k = 5$, CIW outperforms LCS for the values of n we tested. However, as n increases, the ratio of the run time of CIW to LCS also increases. The trend suggests that with high enough n, LCS might eventually outperform CIW. The trend is similar for $k = 6$, but LCS is faster than CIW for $n \geq 53$. For $k \geq 7$, LCS

$k \rightarrow$	3-Way			4-Way			5-Way			6-Way		
$n \downarrow$	LCS	CIW	R	LCS	CIW	R	LCS	CIW	R	LCS	CIW	R
45	13.2	1.15	1/12	6.99	.92	1/8	2.80	.82	1/3	1.72	.81	1/2
46	12.7	1.47	1/9	9.09	1.20	1/8	4.86	1.07	1/5	1.95	.99	1/2
47	23.1	2.63	1/9	13.4	2.20	1/6	6.39	1.85	1/3	2.36	1.69	1
48	55.2	3.51	1/16	18.6	2.58	1/7	7.90	2.16	1/4	2.78	1.92	1
49	41.2	4.46	1/9	27.9	3.87	1/7	9.89	3.54	1/3	5.58	3.09	1/2
50	59.4	6.44	1/9	37.2	5.08	1/7	12.1	4.28	1/3	6.41	3.45	1/2
51	188	11.2	1/17	59.2	9.17	1/6	25.1	7.54	1/3	7.99	6.75	1
52	142	13.9	1/10	81.5	10.7	1/8	31.2	9.02	1/3	9.73	7.70	1
53	220	25.2	1/9	119	17.4	1/7	39.3	15.2	1/3	11.6	13.7	1
54	917	44.8	1/20	153	22.0	1/7	51.7	18.7	1/3	14.2	15.2	1
55	574	43.3	1/13	161	38.7	1/4	45.4	34.1	1	21.2	30.7	1
56	290	70.5	1/4	454	51.1	1/9	72.2	40.7	1/2	21.1	35.0	2
57	-	139	-	-	83.8	-	107	75.7	1	33.3	61.4	2
58	-	131	-	-	105	-	111	82.5	1	33.6	68.1	2
59	-	228	-	-	186	-	187	165	1	50.3	139	3
60	-	304	-	-	254	-	263	201	1	57.9	156	3

$k \rightarrow$	7-Way			8-Way			9-Way			10-Way		
$n \downarrow$	LCS	CIW	R	LCS	CIW	R	LCS	CIW	R	LCS	CIW	R
50	3.57	2.99	1	2.17	2.47	1	1.43	2.53	2	2.29	2.62	1
51	4.24	5.57	1	2.28	4.54	2	1.59	3.78	2	3.42	4.26	1
52	4.98	6.63	1	2.54	5.47	2	1.87	4.43	2	3.17	6.02	2
53	5.61	10.9	2	2.90	8.98	3	2.44	7.39	3	2.91	8.31	3
54	6.47	12.7	2	3.38	10.0	3	2.94	8.90	3	2.50	8.86	4
55	8.06	24.9	3	3.92	19.0	5	4.95	15.6	3	2.55	16.3	6
56	9.40	27.5	3	4.46	21.4	5	5.22	17.9	3	2.93	16.1	6
57	24.1	47.1	2	11.0	36.6	3	5.27	26.3	5	3.72	19.7	5
58	27.1	55.4	2	12.1	42.0	3	5.61	31.2	6	5.61	26.0	5
59	33.8	109	3	14.1	73.6	5	6.24	52.4	8	8.84	39.1	4
60	39.4	123	3	16.4	84.9	5	7.62	60.3	8	11.9	45.0	4
61	28.2	219	8	18.6	154	8	9.19	98.8	11	19.6	70.4	4
62	31.1	246	8	21.8	167	8	11.8	115	10	18.0	87.3	5
63	52.5	487	9	26.9	332	12	15.4	214	14	15.5	151	10
64	97.8	579	6	34.2	443	13	35.8	294	8	15.1	206	14
65	144	1068	7	90.9	750	8	38.8	509	13	15.5	352	23
66	149	1111	7	102	794	8	41.0	550	13	19.8	391	20
67	202	2377	12	122	1685	14	44.7	996	22	29.2	610	21
68	225	2590	12	144	1873	13	50.4	1244	25	43.0	792	18
69	444	-	-	168	3424	20	63.1	-	-	78.7	-	-
70	-	-	-	202	-	-	80.7	-	-	94.8	-	-

Table 5.2: The average time in seconds to optimally partition 48-bit integers 3 through 10 ways using LCS and CIW .

dominates CIW (except for k=7, n=50) and the ratio increases with increasing n, suggesting that LCS is asymptotically faster than CIW. The biggest difference is for $k = 9$ and $n = 68$ where CIW takes 24.7 times as long as LCS on average.

5.7.6 Experimental Results: Memory Usage

Both CIW and LCS use memory to generate subsets and also to cache them. Table 5.3 shows the average memory in GB to partition n integers into $k = 3$ to 6 subsets. There are three columns for each value of k. The first two columns report the average memory required by LCS and CIW respectively. The third column is the ratio of the memory usage CIW to LCS. Table 5.3 shows the same results for $k = 7$ to 10 subsets.

Since LCS only stores low cardinality subsets, the caches tend to be smaller than for CIW. However, LCS uses a variant of EHS to generate subsets while CIW uses ESS. In general, Schroeppel and Shamir requires the square root of the amount of memory that Horowitz and Sahni does. However, since LCS is only generating low cardinality subsets, it uses less memory than standard EHS would.

For $k \leq 7$, CIW uses significantly less memory than EHS. For $k = 8$, the memory usage of the two algorithms is comparable. For $k = 9$ and $k = 10$, LCS uses significantly less memory than CIW. As k increases, for fixed n, the average cardinality of the subsets of a partition goes down. As the average cardinality goes down, low cardinality EHS takes much less memory.

5.8 Summary

This chapter has introduced Cached Iterative Weakening (CIW), a state-of-the-art algorithm for multi-way number partitioning. Previous algorithms for number partitioning had all been either based on branch-and-bound (Chapter 3) or binary-search over bin-packing problems (Chapter 4). The previous algorithms begin by calculating an approximate partition to generate lower and upper bounds. They then shrink these bounds until an optimal partition

$k \rightarrow$	3-Way			4-Way			5-Way			6-Way		
$n \downarrow$	LCS	CIW	R	LCS	CIW	R	LCS	CIW	R	LCS	CIW	R
45	.92	.01	1/139	.74	.00	1/152	.23	.01	1/24	.15	.01	1/13
46	1.24	.01	1/203	.92	.01	1/176	.47	.01	1/41	.17	.01	1/15
47	1.85	.01	1/306	1.36	.01	1/219	.67	.01	1/53	.19	.01	1/14
48	2.44	.01	1/404	1.67	.01	1/242	.80	.01	1/74	.22	.01	1/16
49	3.71	.01	1/557	2.90	.01	1/449	.91	.01	1/86	.51	.02	1/28
50	4.92	.01	1/706	3.63	.01	1/517	1.10	.01	1/90	.60	.02	1/36
51	7.34	.00	1/1475	5.38	.01	1/729	2.70	.01	1/216	.80	.03	1/30
52	9.89	.01	1/1796	6.63	.01	1/855	3.23	.01	1/233	.93	.03	1/34
53	14.8	.01	1/2292	11.5	.01	1/1406	3.73	.01	1/255	1.06	.04	1/25
54	19.0	.01	1/2664	14.4	.01	1/1507	4.54	.02	1/282	1.25	.04	1/33
55	29.3	.01	1/3933	15.4	.01	1/1443	4.17	.02	1/176	2.14	.07	1/30
56	26.1	.01	1/3038	15.8	.01	1/1310	7.59	.02	1/341	2.15	.06	1/35
57	-	.01	-	-	.01	-	9.70	.03	1/314	3.67	.10	1/37
58	-	.01	-	-	.02	-	9.81	.03	1/349	3.69	.10	1/38
59	-	.02	-	-	.02	-	17.0	.04	1/431	4.45	.21	1/21
60	-	.02	-	-	.02	-	17.2	.04	1/411	4.49	.16	1/28

$k \rightarrow$	7-Way			8-Way			9-Way			10-Way		
$n \downarrow$	LCS	CIW	R	LCS	CIW	R	LCS	CIW	R	LCS	CIW	R
45	.09	.02	1/5	.06	.03	1/2	.05	.04	1	.05	.04	1
46	.09	.02	1/5	.06	.03	1/2	.07	.04	1/2	.05	.04	1
47	.12	.03	1/5	.06	.04	1/2	.07	.05	1	.05	.06	1
48	.13	.02	1/6	.07	.04	1/2	.08	.06	1	.05	.07	1
49	.14	.04	1/4	.14	.07	1/2	.08	.09	1	.06	.09	2
50	.30	.03	1/9	.15	.06	1/2	.08	.09	1	.06	.09	2
51	.38	.06	1/6	.16	.10	1/2	.09	.14	2	.09	.15	2
52	.43	.07	1/7	.18	.11	1/2	.09	.12	1	.10	.16	2
53	.48	.10	1/5	.21	.20	1	.10	.25	3	.10	.28	3
54	.56	.10	1/6	.23	.20	1	.11	.30	3	.11	.32	3
55	.73	.18	1/4	.26	.39	1	.27	.40	1	.12	.51	4
56	.87	.17	1/5	.29	.32	1	.30	.42	1	.13	.48	4
57	2.14	.29	1/7	.90	.55	1/2	.36	.81	2	.14	.83	6
58	2.50	.28	1/9	1.03	.56	1/2	.39	.74	2	.15	.90	6
59	3.26	.50	1/6	1.38	.83	1/2	.43	1.22	3	.16	1.16	7
60	3.80	.54	1/7	1.54	.88	1/2	.47	1.31	3	.17	1.29	8
61	2.45	1.01	1/2	1.66	1.66	1	.52	2.16	4	.53	2.21	4
62	2.47	.91	1/3	1.87	1.64	1	.58	2.43	4	.59	2.72	5
63	3.98	1.55	1/3	2.03	2.72	1	.72	3.91	5	.73	4.17	6
64	8.89	1.35	1/7	2.30	2.80	1	2.30	3.99	2	.80	4.87	6
65	14.8	2.92	1/5	7.81	5.60	1	3.00	8.43	3	.85	9.50	11
66	14.9	2.87	1/5	8.93	5.67	1/2	3.34	7.41	2	.93	8.86	10
67	17.5	6.53	1/3	11.8	8.85	1	3.59	14.0	4	1.01	12.5	12
68	17.5	4.38	1/4	13.3	9.16	1	4.01	13.6	3	1.11	14.6	13
69	26.0	-	-	14.4	16.6	1	4.33	-	-	1.42	-	-
70	-	-	-	16.3	-	-	4.86	-	-	1.55	-	-

Table 5.3: The average memory use in GB to optimally partition 48-bit integers 3 through 10 ways using CIW and LCS .

is found. In contrast, CIW uses iterative weakening to search for the optimal partition. It starts by theorizing a perfect partition is possible and sets the upper bound to C_P^*, then calculates a lower bound based on this upper bound. It then iteratively widens this bound until it finds a complete partition. The first complete partition found is optimal.

Along with weakening the bounds, as opposed to performing a branch-and-bound search, CIW also generates subsets only once using ESS and caches them in CIE trees. In contrast, the previous algorithms recursively partition the input set of integers and perform exponential searches at each node of the recursive search tree in order to to generate subsets with sums in range. The iterative weakening along with the caching in concert make CIW orders of magnitude faster than previous algorithms for multi-way partitioning.

We also presented new research on low cardinality search (LCS) which combines the ideas of CIW and the Moffitt partitioning algorithm (MOF) to create an even faster algorithm for some values of n and k. The intuition behind LCS is that given the average cardinality of the subsets in a complete partition $\frac{n}{k}$, it is unlikely that partitions will have many subsets with cardinality much higher than this average. LCS search caches only low cardinality subsets and performs a two-phase algorithm to search for optimal partitions. In the first phase, it searches for an upper bound while populating the low cardinality CIE trees. In the second phase, it uses branch-and-bound along with the CIE trees to either prove the upper bound optimal or find a better optimal partition.

Part III

Experimental Summary, Future Work and Conclusions

CHAPTER 6

High Level Experimental Summary

Throughout this thesis, we have described the following eight algorithms for optimal multi-way number partitioning. The year is when the solver was published. The section is where we discussed the algorithm in this thesis. The citation is for the paper in which the solver was introduced.

Algorithm	Year	Section	Citation
Recursive number partitioning (RNP)	2009	3.4	[Kor09]
Improved Recursive number partitioning (IRNP)	2011	3.4.6	[Kor11]
The Moffitt algorithm	2013	3.5	[Mof13]
Binary-search improved bin completion (BSIBC)	2013	4.5.4	[SK13]
Binary-search branch-and-cut-and-price (BSBCP)	2013 [1]	4.6	[DIM08, SK13]
Sequential Number Partitioning (SNP)	2013	3.7	[KSM13]
Cached iterative weakening (CIW)	2014	5	[SK14]
Low cardinality search (LCS)	2014	5.7	N/A

We have shown experimental data comparing the run times of these algorithms in tables 3.3, 3.4, 3.4, 4.3, 5.1 and 5.2. However, for specific values of n and k, these tables only compare two algorithms. In this chapter, we provide graphs for comparing the run times of all eight algorithms together visually.

The problem instances are the same that we have presented throughout this thesis. The integers are sampled uniformly at random from the range $[1, 2^{48} - 1]$. We generated 100

[1] Though the algorithm we used for BSBCP was created in 2013, Dell'Amico [DIM08] presented a BSBCP algorithm in 2008. However, they only had experimental results for very low precision integers in the range $[1, 10^3]$.

problem instances for each n from $n = 20$ to 60. Figure 6.1 shows results for $k = 3$ and $k = 4$, figure 6.2 for $k = 5$ and $k = 6$, figure 6.3 for $k = 7$ and $k = 8$, and figure 6.4 for $k = 9$ and $k = 10$. All algorithms except for the original RNP algorithm were run against the same problem instances on an Intel Xeon X5680 CPU at 3.33GHz.

We have not run experiments for the RNP algorithm since it is dominated for each n and k by at least one and usually all of the other algorithms. Instead, the RNP data was taken from [Kor09]. The RNP data set was sampled uniformly at random from the range $[1, 2^{31} - 1]$. The algorithm was run on an IBM Intellistation with a two gigahertz AMD Opteron processor. This is a slower machine than the X5680, but since RNP is always multiple orders of magnitude slower than the state of the art, this should not matter.

It is interesting to see how drastically the run times have improved since 2009 when experimental results for RNP were published, the first modern paper on high precision optimal partitioning. In general, the state-of-the-art algorithm is always at least four orders of magnitude faster than RNP for the experiments we ran as n increases.

SNP is the dominant algorithm for $k = 3$ as well as for small values of n for $k = 4$ and $k = 5$. CIW dominates for large values of n for $k = 4$ and $k = 5$ as well as middle values of n for $k = 7$ through $K = 12$. LCS is the dominant algorithm for $k = 6$ through $k = 10$ and large n. MOF is the best for $k = 8$ through $K = 10$ for very small values of n. BSIBC and BSBCP are best for $k = 11$ and $k = 12$ along with small values of n.

The state-of-the-art algorithm for multi-way number partitioning algorithm depends on the size of the input set n, the number of subsets in a partition k and the precision of the input numbers. Table 6.1 shows the algorithm that had the best average run time for each value of n and k for multi-way partitioning of 48-bit integers.

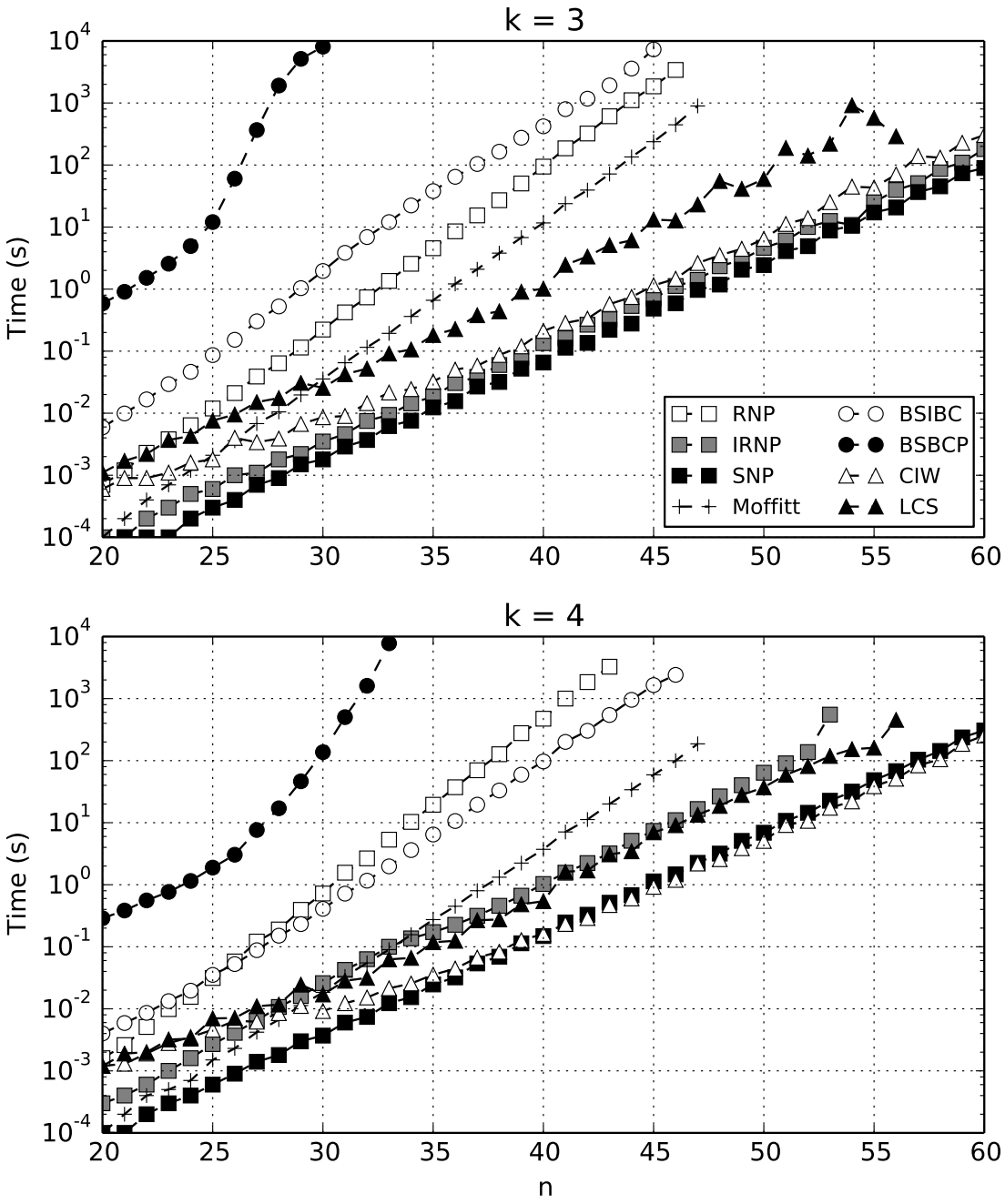


Figure 6.1: The average run time of eight algorithms for three and four-way partitioning.

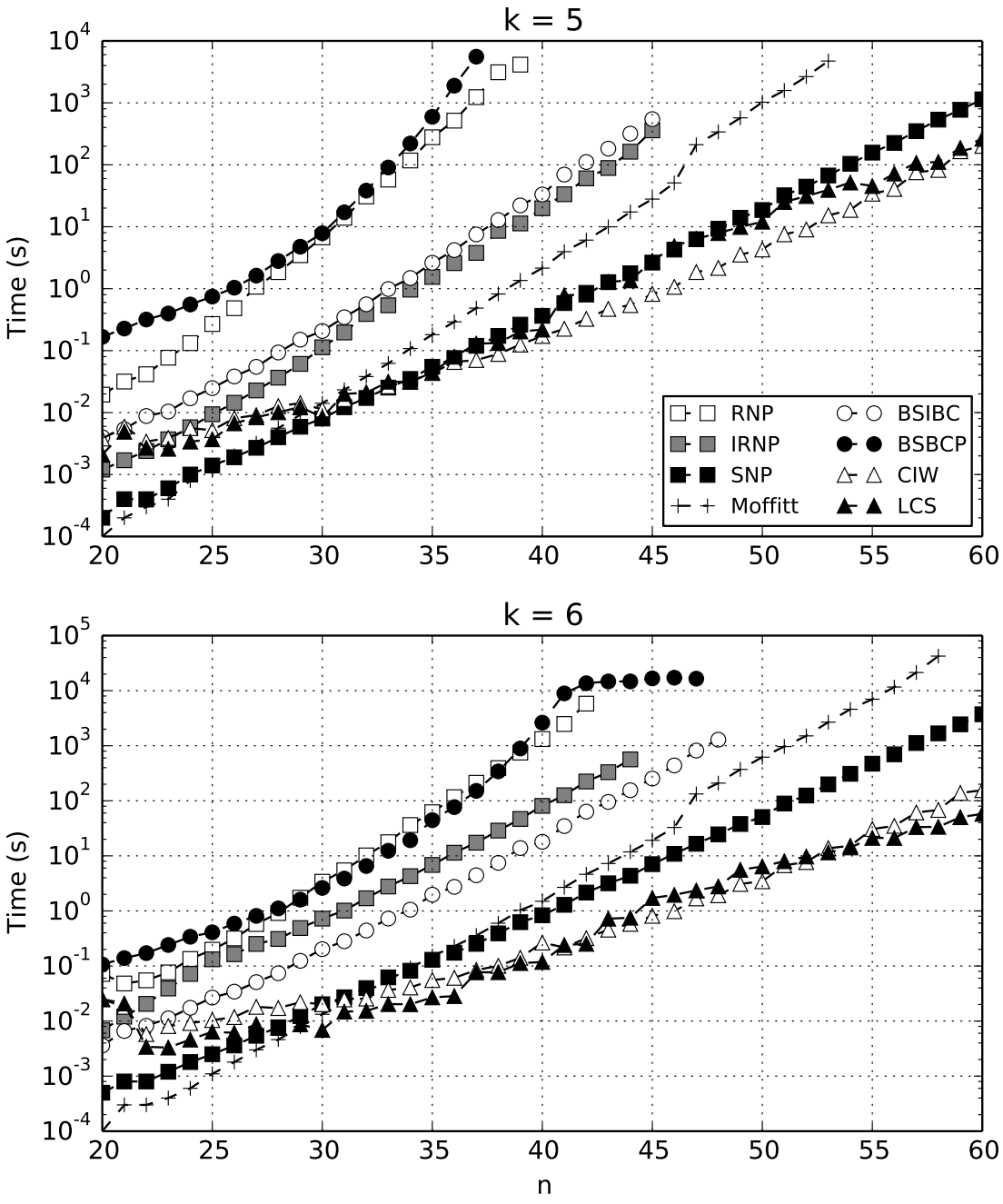


Figure 6.2: The average run time of eight algorithms for five and six-way partitioning.

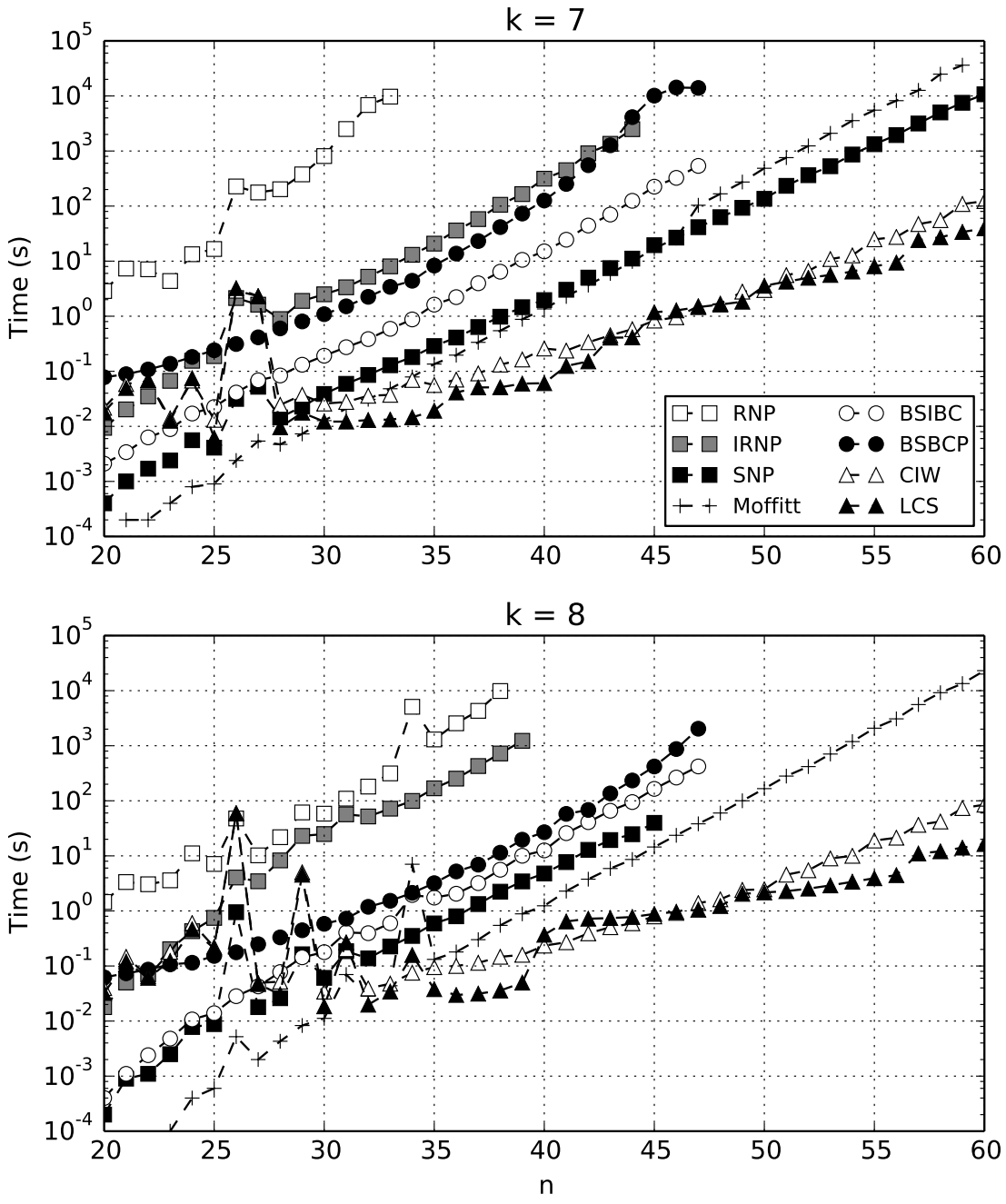


Figure 6.3: The average run time of eight algorithms for seven and eight-way partitioning.

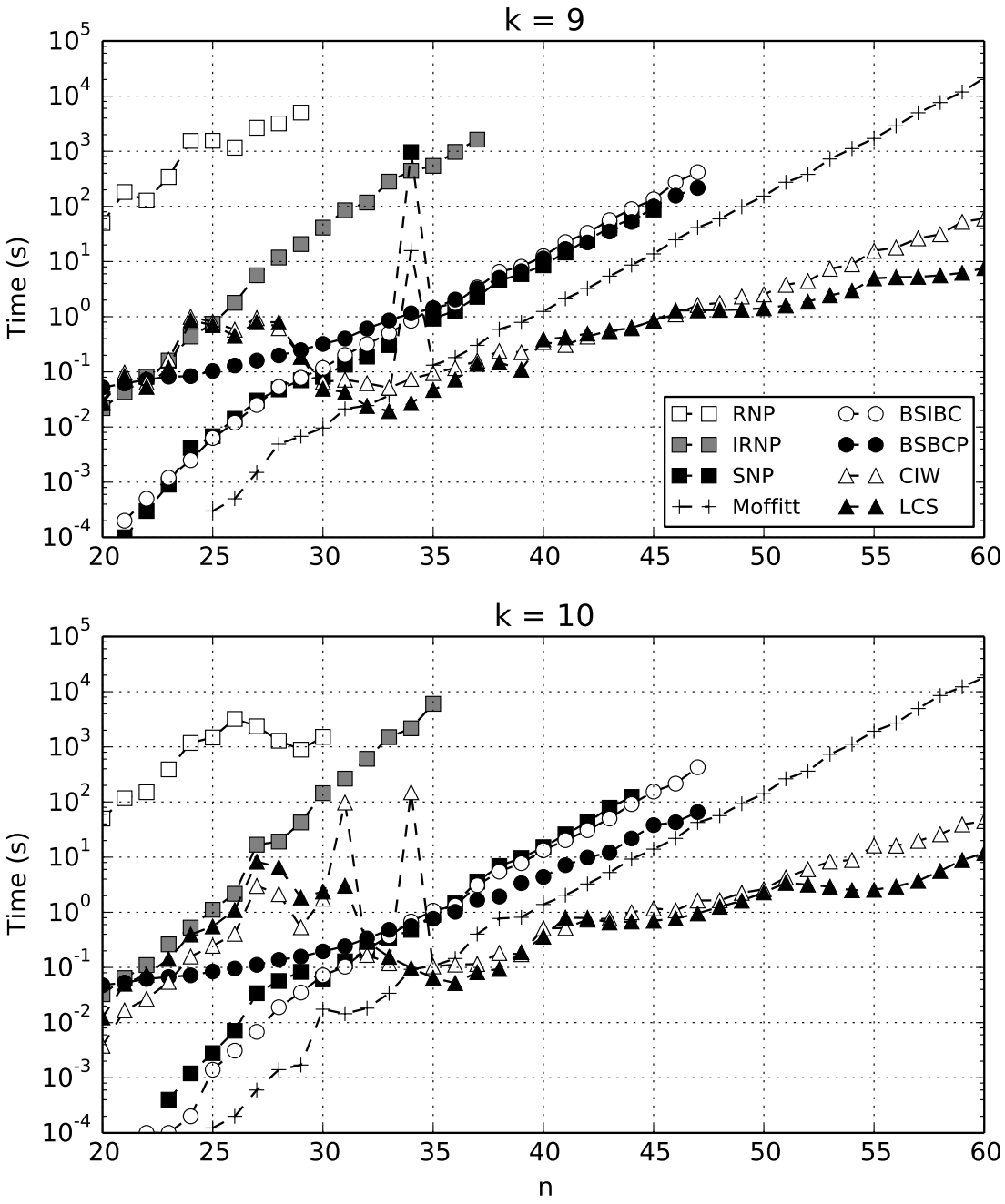


Figure 6.4: The average run time of eight algorithms for nine and ten-way partitioning.

$k \rightarrow$ $n \downarrow$	3-Way	4-Way	5-Way	6-Way	7-Way	8-Way	9-Way	10-Way	11-Way	12-Way
30	SNP	SNP	SNP	LCS	LCS	MOF	MOF	MOF	IBC	IBC
31	SNP	SNP	SNP	LCS	LCS	MOF	MOF	MOF	IBC	IBC
32	SNP	SNP	SNP	LCS	LCS	LCS	LCS	MOF	IBC	IBC
33	SNP	SNP	SNP	LCS	LCS	LCS	LCS	MOF	IBC	IBC
34	SNP	SNP	LCS	LCS	LCS	CIW	LCS	MOF	BCP	IBC
35	SNP	SNP	CIW	LCS	LCS	LCS	LCS	LCS	BCP	BCP
36	SNP	SNP	CIW	LCS	LCS	LCS	LCS	LCS	BCP	BCP
37	SNP	SNP	CIW	LCS	LCS	LCS	LCS	LCS	BCP	BCP
38	SNP	SNP	CIW	LCS	LCS	LCS	LCS	LCS	BCP	BCP
39	SNP	SNP	CIW	LCS	LCS	LCS	LCS	CIW	BCP	BCP
40	SNP	SNP	CIW	LCS	LCS	CIW	CIW	LCS	LCS	LCS
41	SNP	CIW	CIW	CIW	LCS	CIW	CIW	CIW	LCS	LCS
42	SNP	CIW	CIW	LCS	LCS	CIW	CIW	CIW	LCS	LCS
43	SNP	CIW	CIW	CIW	LCS	CIW	LCS	LCS	CIW	LCS
44	SNP	CIW	CIW	CIW	LCS	CIW	CIW	LCS	CIW	LCS
45	SNP	CIW	CIW	CIW	CIW	CIW	LCS	LCS	CIW	LCS
46	SNP	CIW	CIW	CIW	CIW	CIW	CIW	LCS	CIW	CIW
47	SNP	CIW	CIW	CIW	LCS	LCS	LCS	LCS	CIW	CIW
48	SNP	CIW	CIW	CIW	LCS	LCS	LCS	LCS	CIW	CIW
49	SNP	CIW	CIW	CIW	LCS	LCS	LCS	LCS	CIW	CIW
50	SNP	CIW	CIW	CIW	CIW	LCS	LCS	LCS	LCS	LCS
51	SNP	CIW	CIW	CIW	LCS	LCS	LCS	LCS	LCS	LCS
52	SNP	CIW	CIW	CIW	LCS	LCS	LCS	LCS	LCS	LCS
53	SNP	CIW	CIW	LCS	LCS	LCS	LCS	LCS	LCS	LCS
54	SNP	CIW	CIW	LCS	LCS	LCS	LCS	LCS	LCS	LCS
55	SNP	CIW	CIW	LCS	LCS	LCS	LCS	LCS	LCS	LCS
56	SNP	CIW	CIW	LCS	LCS	LCS	LCS	LCS	LCS	LCS
57	SNP	CIW	CIW	LCS	LCS	LCS	LCS	LCS	LCS	LCS
58	SNP	CIW	CIW	LCS	LCS	LCS	LCS	LCS	LCS	LCS
59	SNP	CIW	CIW	LCS	LCS	LCS	LCS	LCS	LCS	LCS
60	SNP	CIW	CIW	LCS	LCS	LCS	LCS	LCS	LCS	CIW

Our Algorithms

LCS	- Low Cardinality Search
CIW	- Cached Iterative Weakening
SNP	- Sequential Number Partitioning
IBC	- Binary-Search Improved Bin Completion

Previous Algorithms

MOF	- Moffitt Partitioning
BCP	- Binary-Search Branch-and-Cut-and-Price

Table 6.1: The algorithm with the fastest average run time for $30 \leq n \leq 60$ and $3 \leq k \leq 12$.

CHAPTER 7

Contributions, Future Work and Conclusions

7.1 Summary of Contributions

This thesis has covered algorithms for computing optimal solutions to both the two-way and multi-way number-partitioning problem. The two-way number-partitioning problem is one of Richard Karp's 21 original NP-complete problems [Kar72] and one of Garey and Johnson's six fundamental NP-complete problems [GJ79]. Multi-way number partitioning is theoretically interesting both because it is NP-complete, but also because it is perhaps the simplest NP-complete problem to describe. Given a set S of n integers, separate S into k subsets such that the largest subset sum is minimized.

Despite the fact that number partitioning is so simple, there have been many algorithms developed since the 1960's which have continuously improved the run times to optimally partition numbers. We have covered both existing algorithms and presented our new algorithms for optimal number partitioning. In this section, we will briefly review these algorithms and highlight our contributions.

7.1.1 Two-Way Number Partitioning

Chapter 2 covers two-way number partitioning. We presented two classic polynomial time approximation algorithms: the greedy algorithm [Gra66] and the Karmarkar-Karp set differencing (KK) Algorithm [KK82]. KK is an important approximation algorithm which is used as an upper bound for many of the optimal algorithms we presented. In practice, it is much more effective than greedy.

We also presented five optimal algorithms. The complete greedy algorithm (CGA) and complete Karmarkar-Karp set differencing algorithm (CKK) are linear-space ($O(n)$) exponential-time ($O(2^n)$) algorithms [Kor98]. When no perfect partition exists, their run times are comparable. However, when many perfect partitions exist, CKK tends to outperform CGA by finding one of these perfect partitions much more quickly.

The Horowitz and Sahni (HS) [HS74] and Schroeppel and Shamir (SH) algorithms [SS81] use exponential memory to improve upon the run time of CGA and CKK. HS uses $O(2^{\frac{n}{2}})$ memory while SS improves upon HS and uses $O(2^{\frac{n}{4}})$ memory. Due to the memory limitations, HS is practical for about $n = 60$ integers and SS for about $n = 120$ integers on a modern desktop computer. They both have time complexity in $O(n \cdot 2^{\frac{n}{2}})$ and hence are much faster than CGA and CKK if no perfect partitions are present and there is sufficient memory to run them. If enough perfect partitions exist, CKK is the dominant algorithm as HS and SS have more overhead to start their search and CGA does not converge on optimal solutions as quickly as CKK does.

There is also a pseudo-polynomial time dynamic programming (DP) algorithm [GJ79] for solving the two-way partition problem. DP is polynomial in the size of S and the magnitude of the input integers. We show a surprising result experimentally both in [KS13] as well as sections 2.5.3 and 2.5.4. Despite being pseudo-polynomial, DP's run time is dominated by CKK and CGA for low precision input and is intractable due to memory limitations for high precision input.

In addition to being one of the most fundamental NP-complete problems, the algorithms used for solving the two-way number-partitioning problem are also indispensable as components of algorithms for solving the multi-way number-partitioning problem. All of the multi-way number-partitioning problems covered in this thesis need to generate subsets with sums in a range. Modified versions of the two-way number-partitioning algorithms are used to generate these subsets.

7.1.2 Multi-Way Number Partitioning

Chapters 3, 4 and 5 cover different classes of algorithms for multi-way number partitioning.

7.1.2.1 Branch-and-Bound Algorithms

Chapter 3 covers branch-and-bound algorithms. We presented two polynomial time approximation algorithms to be used as upper bounds. The multi-way greedy algorithm, also known as longest processing time [Gra66] is the classic upper bound. We also presented the multi-way Karmarkar-Karp algorithm which is used as the upper bound bound for optimal branch-and-bound algorithms [KK82]. Any upper bound is a bound both on overall solution cost and also individual subset sums. However, there are separate lower bounds for each of these values. The lower bound on solution cost is a function of the input set of integers while the lower bound on individual subset sums is a function of the computed upper bound.

Each optimal algorithm for multi-way number partitioning generates a collection of subsets whose sums are all within the lower and upper bounds on individual subsets. We presented three algorithms for this task. Inclusion-exclusion (IE) [Kor09] searches a binary tree of depth n to generate the collection of subsets with time complexity $O(2^n)$ and space complexity $O(n)$. Extended Horowitz and Sahni (EHS) as well as extended Schroeppel and Shamir (ESS) [Kor11] use exponential memory to improve upon the run time of IE. Like their two-way partitioning counterparts, EHS and ESS both have time complexity in $O(n \cdot 2^{\frac{n}{2}})$. HS uses $O(2^{\frac{n}{2}})$ memory while SS improves upon HS and uses $O(2^{\frac{n}{4}})$ memory. In practice, SS tends to be faster. This is likely because it has to sort sets of size $(O^{\frac{n}{4}})$ instead of sets of size $O(2^{\frac{n}{2}})$. This savings exceeds the cost of using heaps to generate the $O(2^{\frac{n}{2}})$ subsets by about a factor of two.

We then presented four optimal branch-and-bound algorithms which recursively partition the input set. RNP uses IE, and IRNP uses ESS to partition the input two-ways, then they both recursively partition the two subsets. MOF uses IE, and SNP uses ESS to generate first subsets, then they recursively partition the remaining integers $k - 1$ ways.

Recursive Number Partitioning (RNP) [Kor09] and Improved Recursive Number Partitioning (IRNP) [Kor11] use the recursive principle of optimality to recursively partition the input set into two sets to be partitioned $\lfloor k/2 \rfloor$ and $\lceil k/2 \rceil$ ways. RNP uses IE to do the two-way partitioning while IRNP uses ESS.

The Moffitt algorithm (MOF) [Mof13] introduced the principle of weakest-link optimality to replace the recursive principle of optimality. Instead of repeatedly searching for balanced two-way partitions, MOF uses IE to search for all first subsets with sums within bounds and then recursively partitions the remaining integers $k-1$ ways.

We present our sequential number partitioning algorithm (SNP) in both [KSM13] and section 3.7. SNP combines the best ideas of IRNP and MOF into a single algorithm. It uses the recursive decomposition of MOF along with ESS from IRNP in one algorithm.

7.1.3 Bin Packing Algorithms

Chapter 4 describes the dual relationship between bin packing and number partitioning. Both algorithms start with an input set S of positive integers. Number partitioning fixes the number of subsets and partitions the integers while minimizing the largest subset sum. In contrast, bin packing fixes the maximum subset sum and minimizes the number of subsets necessary to pack the input integers.

This dual relationship is exploited by the MULTIFIT algorithm [JGJ78] to solve number-partitioning problems using bin-packing algorithms. MULTIFIT first computes lower and upper bounds on the cost of an optimal partition of S into k subsets. It then performs a binary search over the bounds with each probe corresponding to a value in the range $[lb, ub]$ This defines a bin-packing problem with capacity equal to the value of the probe. The binary search continues until the smallest capacity allowing the input integers to be packed into k bins is found. The original MULTIFIT algorithm used first-fit decreasing, an approximation to solve the bin-packing problems. We presented two algorithms for solving the individual bin-packing problems optimally.

It is also possible to solve bin-packing problems using number-partitioning algorithms. One first runs an approximation algorithm to calculate an upper bound on the number of bins in an optimal packing. Then, one tries to solve a number-partitioning problem with k set to one less than this upper bound. If the cost of the solution is less than C, the capacity of the bin, one then tries two less than the upper bound, and so forth. The smallest value of k for which it is possible to partition the input integers into subsets of size C or less is the number of bins in an optimal packing. Solving bin-packing problems with a number-partitioning algorithm is beyond the scope of this thesis and possibly a direction for future research.

Branch-and-cut-and-price (BCP) [BS06] is an operations research algorithm which solves bin-packing problems using linear programming [Chv83, DT97, DT03], branch-and-bound, cutting planes [Gom58] and column generation [DW60, AC05]. Branch-and-cut-and-price was combined with MULTIFIT to solve number-partitioning problems [DIM08]. We also combined the branch-and-cut-and-price solver described in [BS06] with MULTIFIT to create binary-search branch-and-cut-and-price (BSBCP) [SK14].

Bin completion (BC) [Kor02] is an artificial intelligence algorithm for solving the bin-packing problem. We modified BC to create our algorithm, improved bin completion (IBC) [SK13] which was tailored to run with MULTIFIT. IBC added two features to BC: incrementally generated completions and improved variable ordering. We combined MULTIFIT with IBC to create binary-search improved bin completion (BSIBC).

7.1.4 Cached Iterative Weakening

Chapter 5 introduces cached iterative weakening (CIW) [SK14], our current state-of-the-art multi-way number-partitioning algorithm. Before CIW, all previous number partitioning algorithms started by computing upper and lower bounds, and then shrinking those bounds until the optimal solution is found. In contrast, CIW theorizes that a perfect partition is possible and sets the bounds accordingly. It iteratively weakens these bounds until it finds

a feasible solution. This first solution found is guaranteed to be optimal.

Previous algorithms such as RNP, IRNP, MOF, and SNP recursively partition the input set of integers and perform exponential searches at each node of the recursive search tree in order to generate subsets with sums within lower and upper bounds. In contrast, CIW generates subsets once and caches them, thereby avoiding repeated work.

As well as CIW, we also presented new research on low cardinality search (LCS). Low cardinality search generates and caches only low cardinality subsets as opposed to all subsets. For some values of n and k, this is much faster than standard CIW.

7.2 Future Work

The algorithms in this thesis have focused on number partitioning instances with high-precision input integers. The high precision integers decrease the density of subsets with sums close to the cost of the perfect partition. As a result, the algorithms cannot return early before proving that the best solution found so far is optimal. In contrast, with low precision input integers, there tend to be a high density of subsets with sums close to perfect. This leads to many more perfect partitions which if found, can be returned immediately. If perfect partitions exist, the goal becomes finding these partitions as quickly as possible. When the precision of the input integers is low, the values of n and k that become tractable change. Future work involves working with low precision inputs and tailoring algorithms to solve these problems.

Both iterative weakening and caching using cached inclusion-exclusion trees are effective techniques for solving number-partitioning problems. It is worth exploring whether these techniques could be useful in other numeric domains such as knapsack [KPP04], job shop scheduling [CB76], or packing problems in general including rectangle packing and higher dimensional packing [FK07, HK09].

For many of the algorithms in this thesis such as RNP, IRNP, MOF, CIW and LCS, generating subsets with sums in a range is central to the performance of the algorithm. We

have presented three algorithms for this purpose: IE, EHS and ESS. We have also explored an algorithm for generating the m subsets with smallest sums that are greater than C_P^* and a modification to EHS to generate only low cardinality subsets. Any improvement to algorithms solving this problem would improve the performance of many number-partitioning algorithms in this thesis.

The algorithm for generating subsets in sum order (section 5.2) requires a parameter m corresponding to a number of subsets to precompute. This is problematic as it is not obvious what is a good value for m. Finding an algorithm which does not require a parameter, and can efficiently generate subsets in sum order starting with a lower bound could greatly improve CIW and LCS.

The algorithm for generating low cardinality subsets (section 5.7.3) is based on Horowitz and Sahni. It therefore requires a lot more memory than a similar algorithm based on Schroeppel and Shamir or inclusion-exclusion. It is worth exploring if either Schroeppel and Shamir or inclusion-exclusion could be used to efficiently generate low cardinality subsets. Modifying IE to generate low cardinality subsets should be straightforward, the question is whether it would be more efficient. Modifying ESS to generate low cardinality subsets efficiently is not as straightforward. The technique we used for EHS of only generating the low cardinality half sets is not as effective for ESS since it uses two quarter sets to generate the half sets. It is unclear how to efficiently generate only the low cardinality half sets dynamically using the quarter sets of ESS.

In chapter 5, we presented low cardinality search which explores constraining number-partitioning problems using the cardinality of the subsets. The preliminary experiments with LCS look promising. Previous algorithms dealing with number partitioning have focused on bounds on the sums of subsets. However, there are also bounds on the cardinality of subsets as discussed in section 5.3.4. More work needs to be done considering pruning number partitioning problems using bounds on the cardinality of subsets.

7.3 Conclusion

Given an input set of positive integers, separate the integers into k subsets while minimizing the largest subset-sum. This is a simple problem statement that can be described to just about anyone in less than one minute. Yet, given the apparent simplicity, algorithms for solving number-partitioning problems are surprisingly complicated. Depending on the cardinality of the input integers, their precision, the number of subsets to partition into, and the distribution of the input integers, different algorithms dominate.

There are thousands of problems that have been proven to be NP-complete. In some sense, these problems are all equivalent to each other. Number partitioning is one of the simplest of these problems. Other problems such as bin packing, knapsack and rectangle packing are closely related. The greatest hope in studying this problem is that the insights gained can help us to understand not just number partitioning, but other NP-complete problems as well.

Acronyms

BC bin completion. 8, 68, 76–79, 130

BCP branch-and-cut-and-price. 8, 68, 75, 80, 81, 90

BFD best-fit decreasing. 73–76

BSBCP binary-search branch-and-cut-and-price. 75, 90, 91, 106, 120

BSIBC binary-search improved bin completion. 75, 90, 91, 120

CGA complete greedy algorithm. 17–20, 24, 31–34, 36, 55, 127

CIE cached inclusion-exclusion. 99–101, 103–105, 107, 110–113, 117

CIW cached iterative weakening. 8, 93, 94, 97–100, 105–107, 109–113, 115, 117, 120, 130–132

CKK complete Karmarkar-Karp. 19, 20, 24, 31–34, 36, 39, 54, 55, 127

DFS depth-first search. 80

DP dynamic programming. 21–23, 32–34, 36, 127

EHS extended Horowitz and Sahni. 50, 51, 96, 97, 110, 111, 115, 128, 132

ESS extended Schroeppel and Shamir. 50, 51, 53–55, 60, 61, 63, 93, 96, 97, 107, 110, 111, 115, 117, 128, 129, 132

FFD first-fit decreasing. 73–75

HS Horowitz and Sahni. 24, 25, 27, 28, 32, 39, 50, 51, 127, 128

IBC improved bin completion. 75, 76, 79, 130

IE inclusion-exclusion. 47, 48, 55, 57, 59–61, 63, 67, 78–80, 93, 96–100, 110, 112, 128, 129, 132

IRNP improved recursive number partitioning. 51–57, 60, 61, 63, 64, 67, 90–92, 128, 129, 131

KK Karmarkar-Karp. 15, 16, 19, 24, 25, 43–45, 51, 52, 93, 96, 126

LCEHS Low Cardinality Extended Horowitz and Sahni. 110, 111

LCS low cardinality search. 109, 111–113, 115, 117, 120, 131, 132

LP linear Programming. 81

LPT longest processing time. 41

MOF Moffitt algorithm. 8, 56–58, 60, 61, 63, 64, 67, 90–93, 98–100, 106, 107, 109–111, 120, 128, 129, 131

RNP recursive number partitioning. 8, 51, 55, 56, 61, 67, 93, 110, 120, 128, 129, 131

SNP sequential number partitioning. 8, 61, 63, 64, 67, 90, 92, 93, 106, 107, 109–111, 120, 128, 129, 131

SS Schroeppel and Shamir. 27, 28, 32, 39, 50, 51, 54, 55, 67, 127, 128

References

[AC05] Hatem Ben Amor and Jose Valerio de Carvalho. *Cutting stock problems*. Springer, 2005.

[ARS83] Leonard M Adleman, Ronald L Rivest, and Adi Shamir. "Cryptographic communications system and method.", September 20 1983. US Patent 4,405,829.

[BCC93] E. Balas, S. Ceria, and G. Cornuéjols. "A lift-and-project cutting plane algorithm for mixed 0–1 programs." *Mathematical programming*, **58**(1):295–324, 1993.

[Bel57] R. Bellman. "Dynamic Programming, Princeton." *NJ: Princeton UP*, 1957.

[BS06] G. Belov and G. Scheithauer. "A branch-and-cut-and-price algorithm for one-dimensional stock cutting and two-dimensional two-stage cutting." *European Journal of Operational Research*, **171**(1):85–106, May 2006.

[CB76] Edward Grady Coffman and John L Bruno. *Computer and job-shop scheduling theory*. John Wiley & Sons, 1976.

[CGJ96] E.G. Coffman Jr, M.R. Garey, and D.S. Johnson. "Approximation algorithms for bin packing: A survey." *Approximation Algorithms for NP-Hard Problems*, pp. 46–93, 1996.

[Che04] Bo Chen. "Parallel scheduling for early completion." In Joseph YT Leung, editor, *Handbook of scheduling: algorithms, models, and performance analysis*. CRC Press, 2004.

[Chv83] V. Chvatal. *Linear programming*. WH Freeman, 1983.

[Coo71] Stephen A Cook. "The complexity of theorem-proving procedures." In *Proceedings of the third annual ACM symposium on Theory of computing*, pp. 151–158. ACM, 1971.

[CPL09] IBM ILOG CPLEX. "V12. 1: UserâĂŹs Manual for CPLEX." *International Business Machines Corporation*, **46**(53):157, 2009.

[DIM08] M. Dell'Amico, M. Iori, S. Martello, and M. Monaci. "Heuristic and exact algorithms for the identical parallel machine scheduling problem." *INFORMS Journal on Computing*, **20**(3):333–344, 2008.

[DM95] M. Dell'Amico and S. Martello. "Optimal scheduling of tasks on identical parallel processors." *ORSA Journal on Computing*, **7**(2):191–200, 1995.

[DT97] George B Dantzig and Mukund N Thapa. *Linear Programming 1: Introduction*, volume 1. Springer, 1997.

[DT03] George B Dantzig and Mukund N Thapa. *Linear Programming 2: Theory and Extensions*, volume 1. Springer, 2003.

[DW60] George B Dantzig and Philip Wolfe. "Decomposition principle for linear programs." *Operations research*, **8**(1):101–111, 1960.

[EC71] S. Eilon and N. Christofides. "The loading problem." *Management Science*, **17**(5):259–268, 1971.

[FGL94] P.M. França, M. Gendreau, G. Laporte, and F.M. Müller. "A composite heuristic for the identical parallel machine scheduling problem with minimum makespan objective." *Computers & operations research*, **21**(2):205–210, 1994.

[FK05] Alex S. Fukunaga and Richard E. Korf. *Bin completion algorithms for packing and knapsack problems*. University of California at Los Angeles, 2005.

[FK07] Alex S. Fukunaga and Richard E. Korf. "Bin Completion Algorithms for Multicontainer Packing, Knapsack, and Covering Problems." *Journal of Artificial Intelligence Research (JAIR)*, **28**:393–429, 2007.

[FNS04] A. Frangioni, E. Necciari, and M.G. Scutellà. "A multi-exchange neighborhood for minimum makespan parallel machine scheduling problems." *Journal of Combinatorial Optimization*, **8**(2):195–220, 2004.

[GG61] P.C. Gilmore and R.E. Gomory. "A linear programming approach to the cutting-stock problem." *Operations research*, **9**(6):849–859, 1961.

[GGU72] Michael R. Garey, Ronald L. Graham, and Jeffrey D. Ullman. "Worst-case analysis of memory allocation algorithms." In *Proceedings of the fourth annual ACM symposium on Theory of computing*, pp. 143–150. ACM, 1972.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco: W. H. Freeman, 1979.

[GLL79] Ronald L. Graham, Eugene L. Lawler, Jan Karel Lenstra, and A.H.G. Kan. "Optimization and approximation in deterministic sequencing and scheduling: a survey." *Annals of discrete Mathematics*, **5**:287–326, 1979.

[Gom58] R.E. Gomory. "Outline of an algorithm for integer solutions to linear programs." *Bulletin of the American Mathematical Society*, **64**(5):275–278, 1958.

[Gra66] R.L. Graham. "Bounds for certain multiprocessing anomalies." *Bell System Technical Journal*, **45**(9):1563–1581, 1966.

[Hay02] Brian Hayes. "The easiest hard problem." *American Scientist*, **90**(2):113–117, 2002.

[HG95] W.D. Harvey and M.L. Ginsberg. "Limited discrepancy search." In *International Joint Conference on Artificial Intelligence*, volume 14, pp. 607–615. LAWRENCE ERLBAUM ASSOCIATES LTD, 1995.

[HK09] Eric Huang and Richard E Korf. "New Improvements in Optimal Rectangle Packing." In *IJCAI*, pp. 511–516, 2009.

[HS74] E. Horowitz and S. Sahni. "Computing partitions with applications to the knapsack problem." *Journal of the ACM (JACM)*, **21**(2):277–292, 1974.

[HS87] Dorit S. Hochbaum and David B. Shmoys. "Using dual approximation algorithms for scheduling problems theoretical and practical results." *Journal of the ACM*, **34**(1):144–162, January 1987.

[IM08] Manuel Iori and Silvano Martello. "Scatter search algorithms for identical parallel machine scheduling problems." In *Metaheuristics for Scheduling in Industrial and Manufacturing Applications*, pp. 41–59. Springer, 2008.

[JGJ78] Edward G. Coffman Jr, Michael R. Garey, and David S. Johnson. "An application of bin-packing to multiprocessor scheduling." *SIAM Journal on Computing*, **7**(1):1–17, 1978.

[Joh73] David S Johnson. *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

[Kar72] Richard M Karp. *Reducibility among combinatorial problems*. Springer, 1972.

[KK82] N. Karmarkar and R.M. Karp. *The differencing method of set partitioning*. Computer Science Division (EECS), University of California, 1982.

[Kor96] Richard E. Korf. "Improved limited discrepancy search." In *Proceedings of the National Conference on Artificial Intelligence*, pp. 286–291, 1996.

[Kor98] Richard E. Korf. "A complete anytime algorithm for number partitioning." *Artificial Intelligence*, **106**(2):181–203, 1998.

[Kor02] Richard E. Korf. "A new algorithm for optimal bin packing." In *Proceedings of the National conference on Artificial Intelligence*, pp. 731–736, 2002.

[Kor03] Richard E. Korf. "An Improved Algorithm for Optimal Bin Packing." In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03) Acapulco, Mexico*, pp. 1252–1258, 2003.

[Kor09] Richard E. Korf. "Multi-way number partitioning." *Proceedings of the 20nd International Joint Conference on Artificial Intelligence (IJCAI-09)*, pp. 538–543, 2009.

[Kor11] Richard E. Korf. "A Hybrid Recursive Multi-Way Number Partitioning Algorithm." In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI-11) Barcelona, Catalonia, Spain*, pp. 591–596, 2011.

[KPP04] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack problems*. Springer, 2004.

[KS13] Richard E Korf and Ethan L Schreiber. "Optimally Scheduling Small Numbers of Identical Parallel Machines." In *Twenty-Third International Conference on Automated Planning and Scheduling*, 2013.

[KSM13] Richard E Korf, Ethan L Schreiber, and Michael D Moffitt. "Optimal Sequential Multi-Way Number Partitioning." In *International Symposium on Artificial Intelligence and Mathematics (ISAIM-2014)*, 2013.

[Lev73] Leonid A Levin. "Universal search problems." *Problemy Peredachi Informatsii*, **9**(3):115–116, 1973.

[Mer01] Stephan Mertens. "A physicist's approach to number partitioning." *Theoretical Computer Science*, **265**(1):79–108, 2001.

[Mer06] Stephan Mertens. "The easiest hard problem: Number partitioning." *Computational Complexity and Statistical Physics*, **125**(2):125–140, 2006.

[MH78] Ralph Merkle and Martin E Hellman. "Hiding information and signatures in trapdoor knapsacks." *Information Theory, IEEE Transactions on*, **24**(5):525–530, 1978.

[MJG01] Ethel Mokotoff, José Luis Jimeno, and Ana Isabel Gutiérrez. "List scheduling algorithms to minimize the makespan on identical parallel machines." *Top*, **9**(2):243–269, 2001.

[Mof13] Michael D Moffitt. "Search strategies for optimal multi-way number partitioning." In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pp. 623–629. AAAI Press, 2013.

[MT90a] S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations*. Chichester: John Wiley & Sons, 1990.

[MT90b] Silvano Martello and Paolo Toth. "Lower bounds and reduction procedures for the bin packing problem." *Discrete Applied Mathematics*, **28**(1):59–70, 1990.

[Pin12] Michael L Pinedo. *Scheduling: theory, algorithms, and systems*. Springer, 2012.

[Pro93] Foster J Provost. "Iterative weakening: Optimal and near-optimal policies for the selection of search bias." In *AAAI*, pp. 749–755, 1993.

[RNM96] Wheeler Ruml, J Thomas Ngo, Joe Marks, and Stuart M Shieber. "Easily searched encodings for number partitioning." *Journal of Optimization Theory and Applications*, **89**(2):251–291, 1996.

[Sar89] Vivek Sarkar. *Partitioning and scheduling parallel programs for multiprocessors.* MIT press, 1989.

[Sch02] J.E. Schoenfield. "Fast, exact solution of open bin packing problems without linear programming." *Draft, US Army Space & Missile Defense Command*, p. 45, 2002.

[Sha83] Adi Shamir. "A polynomial time algorithm for breaking the basic Merkle-Hellman cryptosystem." In *Advances in Cryptology*, pp. 279–288. Springer, 1983.

[SK13] Ethan L Schreiber and Richard E Korf. "Improved bin completion for optimal bin packing and number partitioning." In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence (IJCAI-13) Beijing, China*, pp. 651–658. AAAI Press, 2013.

[SK14] Ethan L Schreiber and Richard E Korf. "Cached Iterative Weakening for Optimal Multi-Way Number Partitioning." In *Proceedings of the Twenty-Eighth Annual Conference on Artificial Intelligence (AAAI-14) Quebec City, Canada.* AAAI Press, 2014.

[SS81] Richard Schroeppel and Adi Shamir. "A T=O($2^{n/2}$), S=O($2^{n/4}$) Algorithm for Certain NP-Complete Problems." *SIAM Journal of Computing (SICOMP)*, **10**(3):456–464, 1981.

[Tur36] Alan Mathison Turing. "On computable numbers, with an application to the Entscheidungsproblem." *J. of Math*, **58**:345–363, 1936.

[Van99] F. Vanderbeck. "Computational study of a column generation algorithm for bin packing and cutting stock problems." *Mathematical Programming*, **86**(3):565–594, 1999.

[Wal97] Toby Walsh. "Depth-bounded discrepancy search." In *International joint conference on artificial intelligence*, volume 15, pp. 1388–1395. LAWRENCE ERLBAUM ASSOCIATES LTD, 1997.

[Wal09] Toby Walsh. "Where Are the Really Hard Manipulation Problems? The Phase Transition in Manipulating the Veto Rule." In *IJCAI*, pp. 324–329, 2009.