Optimal Multi-Way Number Partitioning

ETHAN L. SCHREIBER and RICHARD E. KORF, University of California, Los Angeles
MICHAEL D. MOFFITT, Google, Austin, Texas

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 runtimes on k identical machines such that the makespan, the elapsed time to complete the schedule, is minimized. The two-way number-partitioning decision problem is one of the original 21 problems that Richard Karp proved NP-complete. It is also one of Garey and Johnson's six fundamental NP-complete problems and the only one based on numbers.

This article explores algorithms for solving multi-way number-partitioning problems optimally. We explore previous algorithms as well as our own algorithms, which fall into three categories: sequential number partitioning (SNP), a branch-and-bound algorithm; binary-search improved bin completion (BSIBC), a bin-packing algorithm; and cached iterative weakening (CIW), an iterative weakening algorithm. We show experimentally that, for large random numbers, SNP and CIW are state-of-the-art algorithms depending on the values of n and k. Both algorithms outperform the previous state of the art by up to seven orders of magnitude in terms of runtime.

Categories and Subject Descriptors: I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Graph and tree search strategies*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Scheduling, search, combinatorial optimization

ACM Reference format:
Ethan L. Schreiber, Richard E. Korf, and Michael D. Moffitt. 2018. Optimal Multi-Way Number Partitioning. *J. ACM* 65, 4, Article 24 (July 2018), 61 pages.
https://doi.org/10.1145/3184400

1 INTRODUCTION

1.1 Problem Description

Given an input multiset $S = \{s_1, s_2, \ldots, s_n\}$ of n positive integers and a number of subsets k, the multi-way number-partitioning problem is to separate the integers of S into k mutually exclusive and collectively exhaustive subsets. Formally, we aim to partition S into $P = \langle S_1, S_2, \ldots, S_k \rangle$ such that $\bigcup_{i=1}^{k} S_i = S$. There are three natural objective functions for P:

(1) Minimize the largest of the k subset sums.
(2) Maximize the smallest of the k subset sums.
(3) Minimize the difference between the largest and smallest of the k subset sums.

Authors' addresses: E. L. Schreiber, 340 Main St, Venice, CA 90291; email: schreiber@google.com; R. E. Korf, Computer Science Department, University of California, Los Angeles, CA, 90095; email: korf@cs.ucla.edu; M. D. Moffitt, 500 W 2nd St, Austin, TX 78701; email: moffitt@google.com.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
2018 Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 0004-5411/2018/07-ART24 $15.00
https://doi.org/10.1145/3184400

We explore algorithms that minimize objective function 1 (minimize the largest subset sum) as it corresponds to minimizing the total time required for a simple scheduling application (see Section 1.3.1).

1.1.1 Example. Consider the input set $S = \{1, 2, 3, 4, 5, 6\}$ with $\text{sum}(S) = 21$ to be partitioned into $k = 3$ subsets. Here are two complete partitions P_A and P_B:

P_A	S_1	S_2	S_3
Subsets:	{1, 5}	{2, 6}	{3, 4}
Sums:	6	8	7
costP_A:	max{6, 8, 7} = 8		

P_B	S_1	S_2	S_3
Subsets:	{1, 6}	{2, 5}	{3, 4}
Sums:	7	7	7
costP_B:	max{7, 7, 7} = 7		

Partition P_A is complete since the integers of the subsets S_1, S_2, and S_3 are mutually exclusive and collectively exhaustive of S. The cost of P_A is the value of the largest subset sum, $\text{cost}(P_A) = \text{sum}(S_2) = 8$.

Partition P_B, like partition P_A, is also complete, with $\text{cost}(P_B) = \text{sum}(S_1) = \text{sum}(S_2) = \text{sum}(S_3) = 7$. No lower-cost partition is possible since decreasing the sum in any of the subsets would increase the sum in another. We call this a *perfect partition* or P^*. If $\text{sum}(S)$ is not divisible by k, the subset sums in a perfect partition will differ by at most 1. The cost of a perfect partition is defined as

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

All perfect partitions are optimal, though not all optimal partitions are perfect.

1.2 Background

We explore algorithms for optimal solutions to the *multi-way number partitioning problem*, also referred to as the *multiprocessor scheduling problem*. The two-way number-partitioning decision problem is one of the original 21 problems that Richard Karp proved NP-complete (Karp 1972). It is also one of Garey and Johnson's six fundamental NP-complete problems (Garey and Johnson 1979) and the only one based on numbers.

Number partitioning is perhaps the easiest NP-complete problem to describe. There is a pseudopolynomial time algorithm for solving it (see Section 2.4.5) 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 large integers, slow even for small integers, and there are many instances for which the exponential number of partitions has a single 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 five decades (Graham 1966, Coffman Jr. et al. 1978, Dell'Amico and Martello 1995, Korf 1998, 2011, Moffitt 2013, and Schreiber and Korf 2013, 2014). This leads us to believe that similar gains may exist for more highly structured NP-complete problems.

We present a number of new algorithms for finding optimal solutions to hard number partitioning problems with large integers having one (or very few) optimal solutions. We show through experiments a progression of asymptotically faster algorithms that each improve performance by orders of magnitude.

1.3 Applications of Multi-Way Number Partitioning

1.3.1 Multiprocessor Scheduling. The multi-way number-partitioning problem is synonymous with the multiprocessor scheduling problem (Garey and Johnson 1979; Sarkar 1989; Pinedo 2012; Dell'Amico et al. 2008; Graham et al. 1979). The input set S of n positive integers correspond to the runtimes 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 the time required 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 in which instead of voting for a candidate, voters veto one candidate, with each voter's veto carrying a different weight (Walsh 2009). 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 its vetoes among the candidates it wishes to lose such that the sum of the veto weights of each of these candidates is larger than the veto weight of the candidate it wishes to win. Multi-way number partitioning with the objective function of maximizing the minimum subset sum can be used to solve this problem.

1.4 Overview

Section 2 discusses both approximate and exact algorithms for solving two-way number partitioning, a special case of multi-way number partitioning. The algorithms in this section form the fundamental basis for multi-way number-partitioning algorithms. All of these algorithms are previous work.

Section 3 covers our branch-and-bound algorithms for solving multi-way number-partitioning problems. It begins with lower bounds and upper bounds, based on approximation algorithms, for multi-way number partitioning. It then describes the complete greedy algorithm (CGA) (Korf 1998), the previous state-of-the-art algorithm. It then covers sequential number partitioning (SNP), which is our branch-and-bound algorithm. We cover two versions of SNP: sequential number partitioning with inclusion-exclusion (SNPIE) (Moffitt 2013; Korf 2009) and sequential number partitioning with extended Schroeppel and Shamir (SNPESS) (Korf et al. 2014; Korf 2011). SNPESS is the state-of-the-art algorithm for multi-way number partitioning for small numbers of subsets.

Section 4 covers solving number-partitioning problems using bin-packing solvers. It discusses the relationship between number partitioning and bin packing, presenting the algorithm MULTIFIT for solving number partitioning problems with any bin-packing solver. It describes bin completion (BC) at a high level and branch-and-cut-and-price (BCP), two optimal bin-packing algorithms that are used with MULTIFIT to optimally solve number-partitioning problems. MULTIFIT with BC is called binary-search improved bin completion (BSIBC) while MULTIFIT with BCP is called binary-search branch-and-cut-and-price (BSBCP). BSIBC (Schreiber and Korf 2013) is our algorithm while BSBCP is previous work. The building blocks for these algorithms—BC, BCP, and MULTIFIT—are all previous works.

Section 5 describes cached iterative weakening (CIW), our iterative weakening number-partitioning algorithm. Branch-and-bound algorithms start with an approximate solution, then search to improve it until a final solution is found and proved optimal. In contrast, an iterative weakening algorithm starts 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.

Table 1. Overview of Notation Used Throughout this Article

Notation	Definition
S	The input multiset of positive integers, $S = \{s_1, s_2, \ldots, s_n\}$.
n	The cardinality of S.
k	The number of subsets in a partition.
P	A partition of S into k subsets, $P = \langle S_1, S_2, \ldots, S_k \rangle$.
P_d	A partial partition of S into d subsets, $P_d = \langle S_1, S_2, \ldots, S_d \rangle$, 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 smallest integer in S.
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.

Note: Some notation localized to only one section is not listed here.

The new work appearing in this article has been presented at the following conferences: IJCAI'09 (Korf 2009), IJCAI'11 (Korf 2011), IJCAI'13 (Schreiber and Korf 2013; Moffitt 2013), ICAPS'13 (Korf and Schreiber 2013), ISAIM'14 (Korf et al. 2014) and AAAI'14 (Schreiber and Korf 2014).[1]

2 TWO-WAY NUMBER PARTITIONING

2.1 Overview

Given an input multiset $S = \{s_1, s_2, \ldots, s_n\}$ of n positive integers, the two-way number-partitioning problem is to separate S into two subsets such that the larger subset sum is minimized. Similar to the multi-way case described in Section 1.1, if a partition exists such that all subset sums differ by at most 1, it is called a *perfect partition*, which we denote as P^*. Its cost is

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

The decision version of this problem, to determine if it is possible to separate S into two subsets such that their sums differ by no more than 1, is one of Karp's original 21 NP-complete problems (Karp 1972). The problem has been called the "easiest hard problem" because, despite being NP-complete, there is an easy-hard-easy transition (see Section 2.5.1) that makes many problem instances trivial to solve, even for large n (Hayes 2002; Korf 1998; Mertens 2006). There is also a pseudopolynomial time dynamic programming algorithm (Garey and Johnson 1979), with runtime polynomial in the sum of the input integers. This is in contrast to polynomial time, which would be polynomial in the number of digits in the input integers.

2.2 Subset-Sum Problem

A problem closely related to the two-way partition problem is the subset-sum problem: Is there a subset of S whose sum equals a particular target value T? Finding a perfect partition for the two-

[1]Each of these papers were authored by a subset of the authors of this article, and the chronological progression of these ideas is somewhat complex. Our presentation here is based on clarity of exposition rather than following the chronological order of these papers.

way number partitioning problem is equivalent to the subset-sum problem with $T = \lceil \frac{\text{sum}(S)}{2} \rceil$. The subset sum problem is also NP-complete (Garey and Johnson 1979).

It can be easily shown that any subset-sum problem reduces to the perfect partition problem. Consider the input set S with target value T. We create input set S' by adding the integer $x = 2T - \text{sum}(S)$ to S. The sum of the input set of the new problem is:

$$\text{sum}(S') = (\text{sum}(S) + x) = (\text{sum}(S) + 2T - \text{sum}(S)) = 2T.$$

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, the subset that does not contain x is a subset of S whose sum is T. If no perfect partition of S' exists, then there is no subset of S whose sum equals T.

This section introduces algorithms for solving two-way partitioning problems both approximately and optimally. All of these 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 multi-way partitioning problems in Sections 3, 4, and 5.

2.3 Polynomial-Time Approximation Algorithms (Upper Bounds)

The following polynomial-time approximation algorithms provide an upper bound for two-way partitioning problems. These bounds are used by exact algorithms to help prune the search space.

2.3.1 Greedy Algorithm. Perhaps the most obvious algorithm for two-way number partitioning is the greedy algorithm (Graham 1966). The greedy algorithm first sorts the input set S into monotonically decreasing order. It then considers the integers one at a time and places them into the subset with the smallest cumulative sum, 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 $O(n \log n + n) = O(n \log n)$ and space $O(n)$. The partition costs are within 4/3 of optimal (Kellerer et al. 2004). The greedy algorithm is optimal for $n \leq 4$ (Korf 2011).

Example 2.1. Consider the input set $S = \{18, 17, 12, 11, 8, 2\}$. The following are the steps the greedy algorithm takes to compute an upper bound on the cost of a two-way partition: $\langle\{18\}, \{\}\rangle$ $\langle\{18\}, \{17\}\rangle\langle\{18\}, \{17, 12\}\rangle\langle\{18, 11\}, \{17, 12\}\rangle\langle\{18, 11, 8\}, \{17, 12\}\rangle\langle\{18, 11, 8\}, \{17, 12, 2\}\rangle$. The upper bound is $\max\langle \sum S_1, \sum S_2\rangle = \max\langle 37, 31\rangle = 37$.

2.3.2 Set Differencing: The Karmarkar-Karp Algorithm (KK). The Karmarkar-Karp (KK) set differencing algorithm (Karmarkar and Karp 1982) provides an alternative to the greedy algorithm. Like the greedy algorithm, 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 only one integer left, which is the difference between the sums of the final sets, $|sum(S_1) - sum(S_2)|$. Given this difference, the larger subset sum is $\frac{\text{sum}(S)+|\text{sum}(S_1)-\text{sum}(S_2)|}{2}$.

Example 2.2. Consider the input set $S = \{18, 17, 12, 11, 8, 2\}$. The 18 and 17 are replaced by their difference of 1, which is inserted into the remaining list, resulting in {12, 11, 8, 2, 1}. Next, the 12 and 11 are replaced by their difference of 1, resulting in {8, 2, 1, 1}. The 8 and 2 are replaced by 6, resulting in {6, 1, 1}. The 6 and 1 are replaced with 5, resulting in {5, 1}. Their difference results in the final subset difference of 4. The cost is $\frac{\text{sum}(S)+\text{difference}}{2} = \frac{68+4}{2} = 36$, corresponding to the partition: $\langle\{18, 12, 2\}, \{17, 11, 8\}\rangle$, with subset sums of 32 and 36, respectively.

To construct the actual subsets, a graph is created with a node for each integer. Whenever two integers are replaced by their difference, an edge between the two nodes is added. The new value

is represented by the node with the larger value. Finally, the resulting graph is two-colored to construct the two subsets.

In general, KK significantly outperforms the greedy algorithm with respect to solution quality. For both algorithms, the difference of the final partition is on the order of the size of the last number to be assigned. For the greedy algorithm, this is the size of the smallest original number. For n integers uniformly distributed between 0 and 1, the greedy method produces a final difference of $O(1/n)$. For KK, repeated differencing operations reduce the size of the final number to be assigned. Yakir (1996) confirmed Karmarkar and Karp's conjecture that the value of the final number to be assigned by KK is $O(1/n^{\alpha \log n})$, for some constant α.

2.4 Optimal Algorithms

2.4.1 Complete Greedy Algorithm (CGA). The CGA (Korf 1998) transforms the greedy algorithm into an optimal algorithm. Like the greedy algorithm, the CGA first sorts S in decreasing order. It then proceeds to partition S into two subsets S_1 and S_2.

The 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 the smaller sum (the greedy choice), and the right branch puts it into the other subset. The CGA keeps track of the sums of both subsets at each node and returns the smallest of the larger subset sums encountered at each leaf node in the tree. The binary tree is searched depth first, visiting left children before right. The 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 next integer of S is only put into one of the subsets since the two trees underneath that node would be identical. This applies to the root node as well, where the sum of each subset is 0, placing the largest integer of S into the first subset only.
(2) If the sum of unassigned integers remaining in S is less than or equal to 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.

2.4.2 Complete Karmarkar-Karp Algorithm (CKK). Like the CGA, the complete Karmarkar-Karp (CKK) algorithm (Korf 1998) transforms the KK approximation algorithm into an optimal algorithm. CKK first sorts S in 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 them with their difference. The alternative is to place the integers 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, which is the KK choice. The right branch puts the two largest remaining integers into the same subset by replacing them with their sum. The difference is inserted into the sorted order, while the sum is prepended to the head of the list. This continues until there is one integer left, which is the difference between the two subset sums on that 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.

Condition	Action
$a+b \leq lb$	: Get next a from A.
$lb < a+b < \text{cost}(P^*)-1$	: Set $lb = a+b$; $ub = \text{sum}(S)-(a+b)$. Get next b from B.
$a+b \in \{\lfloor sum(S)/2 \rfloor, \lceil sum(S)/2 \rceil\}$	: Return $\text{cost}(P^*) = \lceil sum(S)/2 \rceil$.
$\text{cost}(P^*) < a+b < ub$	: Set $ub = a+b$; $lb = \text{sum}(S)-(a+b)$. Get next b from B.
$a+b \geq ub$	: Get next b from B.

Fig. 1. The rules for iterating using the HS algorithm.

Like the 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 $O(n)$.

2.4.3 Horowitz and Sahni (HS). The Horowitz and Sahni (HS) algorithm (Horowitz and Sahni 1974) uses more memory to improve upon the 2^n time complexity of CKK and the CGA. 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 $lb = \text{sum}(S) - ub$. In any solution with a cost better (lower) than the upper bound, the smaller subset sum must exceed this lower bound. It then calculates the perfect partition cost as follows: $\text{cost}(P^*) = \lceil \frac{\text{sum}(S)}{2} \rceil$.

HS sorts the input set S in 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 all $2^{\frac{n}{2}}$ subsets of each half set, including the empty set and the complete half set. Every subset of S can be uniquely represented as the union of a set from A and a set from B. The sets in A and B are sorted by their sums, with A increasing and B decreasing. It then iterates through each set in A and each set in B, starting with the first sets of A and B, respectively. We use a and b to denote the sums of the current sets from A and B, respectively.

If $a+b$ is less than or equal to the lower bound, HS gets the next larger a from A. If $a+b$ is between the lower bound and the perfect partition cost, we have a better solution; the lower bound is set to $a+b$ and the ub to its complement, $\text{sum}(S)-(a+b)$. Also, HS gets the next larger a from A. If $a+b$ is part of a perfect partition P^*, $\text{cost}(P^*)$ is returned immediately. If $a+b$ is between the perfect partition cost 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 smaller b from B. If $a+b$ is greater than or equal to the upper bound, HS gets the next smaller b from B. This process continues until either the A or B lists are exhausted or a perfect partition is found. Figure 1 also lists these rules for iterating through the A and B lists.

For two-way partitioning, there is a simple optimization (Korf and Schreiber 2013). The *largest* integer in S is removed and always assumed to be part of the current set. *largest* is not included in the A or B sets. For each $a \in A$ and $b \in B$, the current subset 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 sets 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 to their size, $2^{\frac{n}{2}}$. Therefore, HS runs in 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 each 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 sorted the input set S into decreasing order and chose the larger integers to populate S_A and the smaller integers to populate S_B. The original HS algorithm does not specify an order for the integers.

2.4.4 Schroeppel and Shamir (SS). The Schroeppel and Shamir (SS) algorithm (Schroeppel and Shamir 1981) 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 increasing 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 the 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 decreasing order as the B list from HS. SS uses these heaps to generate the subsets in sorted order by their sums, 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 $\log 2^{\frac{n}{4}} = \frac{n}{4}$ time. Therefore, the scanning operation has time complexity $O(2 \cdot 2^{\frac{n}{2}} \cdot \frac{n}{4})$. 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 requires only $O(2^{\frac{n}{4}})$ 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 (1974) and Schroeppel and Shamir (1981) claim imprecisely that their algorithms run in time $O(2^{\frac{n}{2}})$, neglecting the factor of n.

2.4.5 Dynamic Programming (DP). A dynamic programming (DP) algorithm (Garey and Johnson 1979; Martello and Toth 1990a; Korf and Schreiber 2013) solves the partition problem in pseudopolynomial time and space. DP allocates a matrix M of bits with n rows and $\text{sum}(S)$ 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 $[0, \text{sum}(S)]$. The bit at row r and column c is set to 1 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 0 except for those in the 0 column, which are all set to 1, corresponding to the empty set. 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 1-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 1, it sets column $c + s_r$ in the current row to 1, where s_r is the integer in S corresponding to the current row. The 1-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 or equal to $\text{cost}(P^*)$, the perfect partition cost, is the optimal subset sum.

There are three simple 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 $\text{cost}(P^*)$ need to be stored. Furthermore, with the exception of the empty set with sum 0, it is impossible to get a value smaller than $\min(S)$, the smallest integer of S. Therefore, DP needs only $\text{cost}(P^*) - min(S)$ columns corresponding to the range

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

- $\underline{\boldsymbol{c + s_r < \text{cost}(P^*)}}$: 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.
- $\underline{\boldsymbol{c + s_r = \text{cost}(P^*)}}$: C^* is set to $\text{cost}(P^*)$ and DP returns immediately.
- $\underline{\boldsymbol{c + s_r \geq C^*}}$: $c + s_r$ is ignored.
- $\underline{\boldsymbol{\text{cost}(P^*) < c + s_r < C^*}}$: C^* is set to $c + s_r$.

Fig. 2. The rules for filling the DP matrix.

$[min(S), \text{cost}(P^*)]$. Column c corresponds to both a subset sum and its complement, $\text{sum}(S) - c$. Since the complement subsets are not stored in the matrix, DP keeps track of C^*, the larger sum in the lowest cost partition found so far, which is initialized to $sum(S)$. Figure 2 lists the rules for scanning the row and updating the matrix and C^*.

Since only 1 row is observed at a time, DP needs to store just 1 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 multiple times, the scan is done in reverse from the last column to the first. When the scan is complete, the bit corresponding to column s_r is set to 1.

The third optimization recognizes that only partitions with sums 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 last column, the scan begins with the largest column c having $c + s_r < C^*$.

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

2.5 Experimental Results

We now present a series of experiments we ran to compare the two-way partitioning algorithms described in this section. Depending on the magnitude of the input integers and the number of integers n, different algorithms are preferable. The size of the input integers is measured by the number of bits b needed to represent them. 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 at most $n \times (2^b - 1)$ 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. Furthermore, the partitions with extreme sums are the rarest and those with roughly equal sums the most numerous. If b is small and n is large, 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 that determines the difficulty of a problem instance. Two-way partitioning has an easy-hard-easy transition for input integers of a particular magnitude. When no perfect partitions exist, as n increases, the problems tend to get more difficult since there are more partitions. However, at some n, perfect

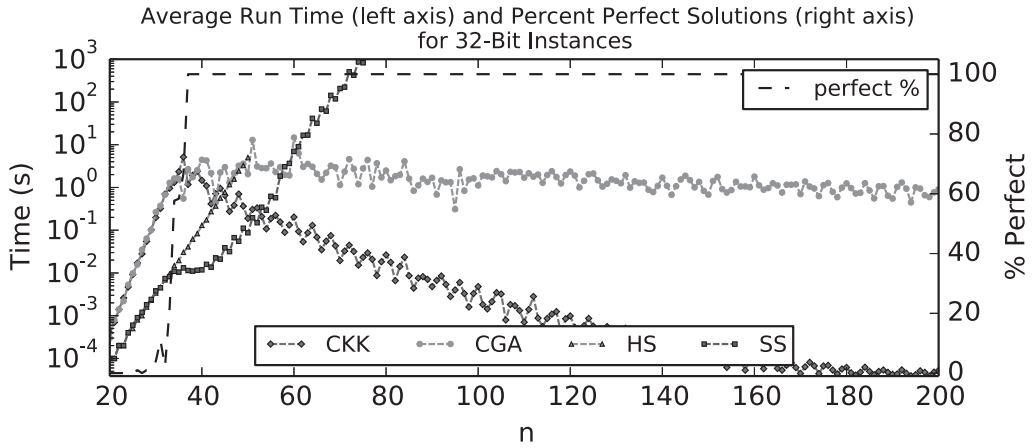


Fig. 3. The average runtimes of the CGA, CKK, HS, and SS as well as the percent of problems with perfect partitions for 32-bit partition instances.

partitions start to appear and the problems eventually get easier again as the number of perfect partitions grows exponentially (Korf 1998; Mertens 2006).

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 runtimes for each value of n. We chose 32-bit integers in order to show the easy-hard-easy transition of two-way number partitioning.

Figure 3 shows both the average runtime of the CGA, CKK, HS, and SS on this dataset (left axis) and the percentage of the 100 problem instances that have a perfect partition (right axis). For n from 20 to 29, none of the optimal 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 the 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 than CGA. On the other hand, while HS and SS are faster than the CGA and CKK for small n, the problems do not get easier for them at $n = 40$. This is because they have to completely generate their "half" sets, which takes $O(2^{n/2})$ time, before they begin searching for optimal partitions.

2.5.2 48-Bit Experiments. We then 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 runtimes for each value of n. We chose 48-bit integers in order to generate hard instances without perfect partitions (Korf 2011).

We ran complete Karmarkar-Karp (CKK), the complete greedy algorithm (CGA), Horowitz and Sahni (HS), and Schroeppel and Shamir (SS) on the benchmark set described above. For 48-bit integers 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 the CGA would speed up since there would be so many perfect partitions, but we were not able to reach this point owing to time limitations. Note that neither HS nor SS display the easy-hard-easy transition, even with perfect partitions.

Figure 4 shows the timing results for the four algorithms. All four algorithms have an exponential explosion in runtime. Both CKK and the CGA, with time complexity in $O(2^n)$, follow a very

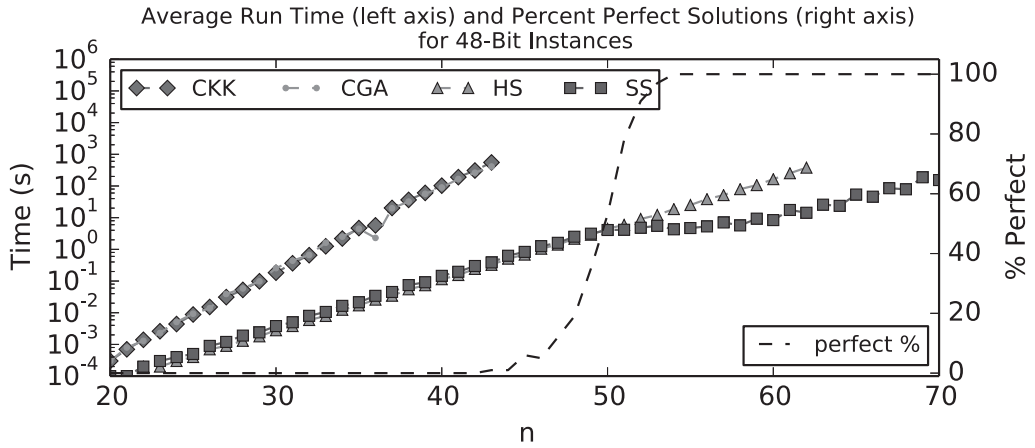


Fig. 4. The average runtimes of the CGA, CKK, HS, and SS as well as the percent of problems with perfect partitions for 48-bit partition instances. The CGA and CKK lines overlap and thus appear as only 1 line.

similar line. Both HS and SS, with time complexity in $O(2^{\frac{n}{2}})$, are significantly faster than both CKK and the 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 problems have perfect partition, as can be seen by the dashed line in Figure 4. 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 increasingly easier to find perfect partitions and the time to generate the complete sets dominates the total runtime of HS and SS. This initial overhead explains the difference between the average runtimes of the two algorithms.

Both CKK and the CGA use memory linear in n. As such, memory usage is not an issue for these algorithms. 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. For 48-bit problem instances, while HS maxes out our memory (48GB) by $n = 60$, SS can solve problems of size $n = 80$ with less than 100 MB of memory. For these problem instances, SS is time bound since there is more than enough memory to solve all problems that can be solved optimally within a reasonable time frame.

2.5.3 Dynamic Programming Results on 16-Bit Instances. Section 2.4.5 covers the pseudopolynomial time DP algorithm for solving two-way number-partitioning problems. The time complexity is $O(n \cdot \text{sum}(S))$, and the space complexity is $O(\text{sum}(S))$. Given this complexity, 48-bit integers 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 sampled uniformly at random from the range $[1, 2^{16} - 1]$. We generated 100 problem instances for each n from $n = 20$ to $n = 100$.

Given the small magnitude of the input integers, 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, the CGA, and DP on this benchmark set. Figure 5 shows the timing results for the three algorithms. For 16-bit integers and n from 20 to 100, both CKK and the 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.

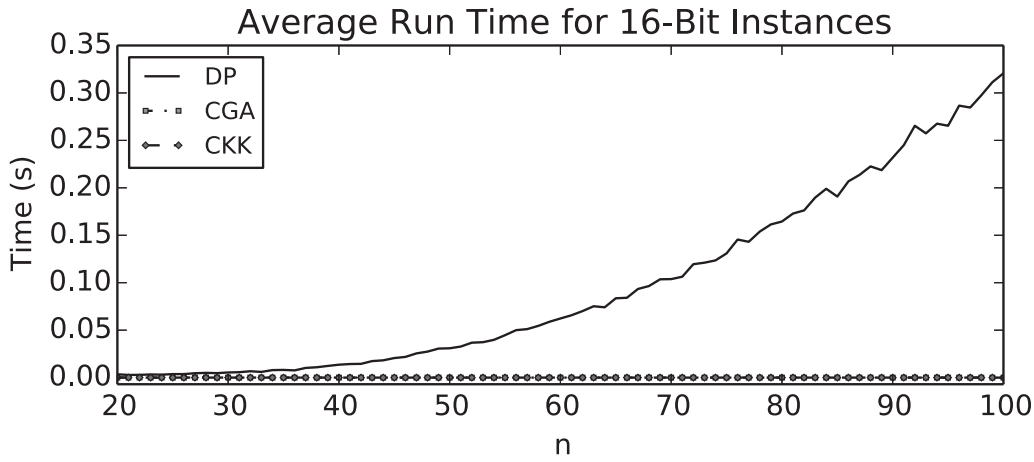


Fig. 5. The average runtimes of CGA, CKK, and DP for solving 16-bit partition instances. The CGA and CKK lines overlap and thus appear as only one line.

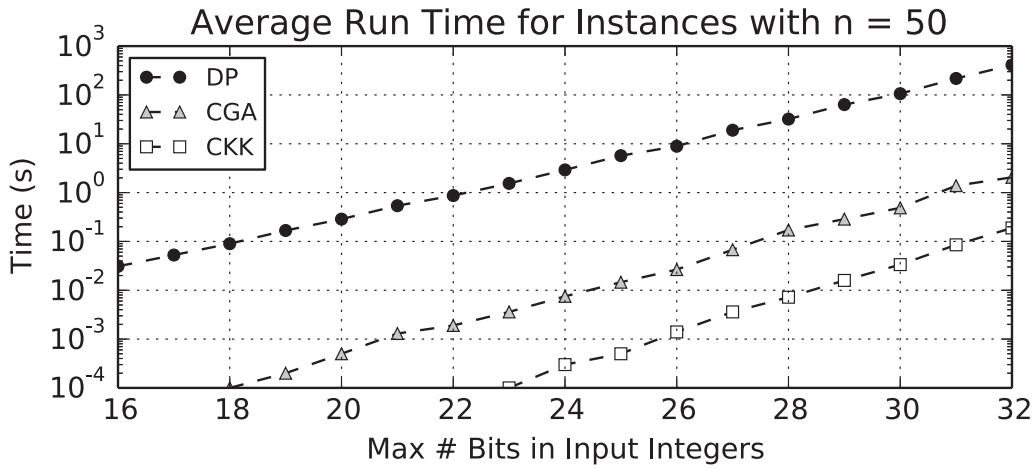


Fig. 6. The average runtimes of the CGA, CKK, and DP for solving 16-bit partition instances.

The perceived wisdom is that, given enough memory, DP — a pseudopolynomial time algorithm — should be the fastest. The 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 (Korf and Schreiber 2013).

2.5.4 Dynamic Programming Results Varying Magnitude. 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 6 shows the average runtimes of CKK, the CGA, and DP as b ranges from 16 to 32. For all of these relatively small-magnitude input sets, CKK and the CGA solve all instances almost

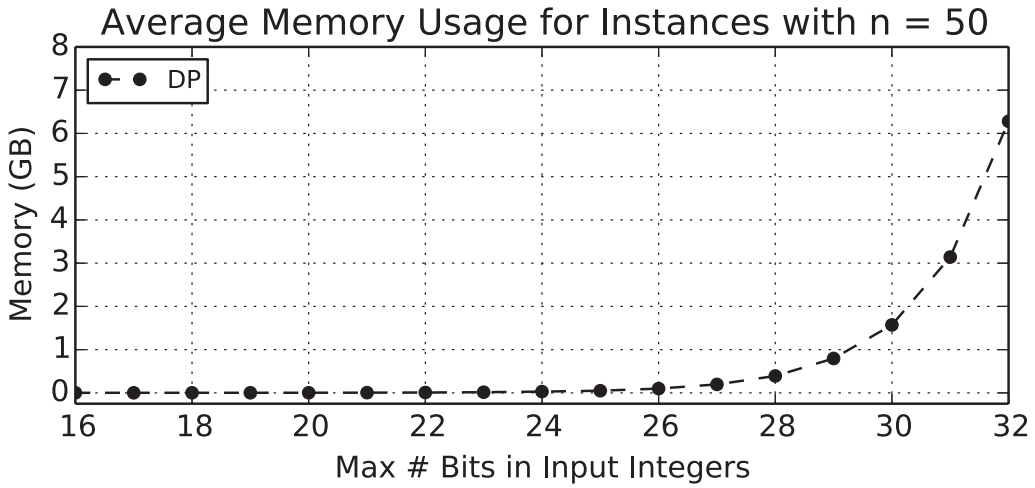


Fig. 7. The memory usage of dynamic programming for solving instances with $n = 50$ as the magnitude of the input integers is varied.

Table 2. The Smallest Value of b for Each n in Which SS Dominates CKK

n	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
b	17	16	17	17	17	18	20	19	19	20	21	22	21	23	24	24	22	25	26	26
n	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
b	26	27	29	29	28	31	31	32	31	32	32	33	33	34	35	35	35	36	37	37
n	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
b	37	39	39	40	39	41	42	42	41	43	44	44	44	46	46	47	46	47	48	48

instantaneously. DP is approximately three orders of magnitude slower than the CGA and four orders of magnitude slower than CKK.

Figure 7 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 sum(S). Since sum(S) grows exponentially with b, the memory usage of DP also grows exponentially. In contrast, both the CGA and CKK use memory linear in n, the cardinality of the input set S; hence, their memory usage is negligible.

Given that DP is much slower than CKK and the CGA for 16-bit integers, is intractable for 48-bit integers, and is dominant for all magnitude datasets from 2^{16} to 2^{32}, it appears not to be a useful algorithm for two-way number partitioning. As we will see, its memory usage makes it even more prohibitive for multi-way partitioning.

2.5.5 Dominant Algorithm as a Function of Cardinality and Magnitude. The state-of-the-art algorithm for two-way number partitioning depends on the size of the input set n and the size of the input integers. For each value of n in the range [20, 79] and b in the range [8, 48], we generated 100 instances uniformly at random from within the range $[1, 2^b]$. For each n, CKK is the dominant algorithm for small b, presumably because of the initial overhead of CKK. For large b, SS is the dominant algorithm. Table 2 lists the smallest value of b for each n for which SS outperforms CKK.

3 BRANCH-AND-BOUND ALGORITHMS

The multi-way number-partitioning problem is to separate a multiset $S = \{s_1, s_2, \ldots, s_n\}$ into k mutually exclusive and collectively exhaustive subsets defining partition $P = \langle S_1, S_2, \ldots, S_k \rangle$ such that the largest subset sum in P, $\max_i(\text{sum}(S_i))$, is minimized (see Section 1.1). This section introduces algorithms for solving this problem approximately, then uses these approximations to solve the problem optimally using branch-and-bound algorithms. The approximate algorithms are used as upper bounds on each subset sum S_i for the optimal algorithms. Lower bounds both on solution cost and the sum of each individual subset S_i are also introduced. The approximations and lower bounds are from previous work, while the branch-and-bound algorithms are our contributions.

3.1 Polynomial-Time Approximation Algorithms (Upper Bounds)

There is a significant amount of literature on approximation algorithms for the multi-way partition problem. See, for example, Mokotoff and Gutiérrez (2001), Iori and Martello (2008), Chen (2004), and Dell'Amico et al. (2008). Nonetheless, we found the following two simple approximation algorithms 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. It is also known as longest processing time (LPT) (Graham 1966). The algorithm first sorts the integers of S in decreasing order. It then considers the integers one at a time and places them into a subset S_i, $1 \leq i \leq k$, with the smallest sum. If two or more subsets both have the same smallest sum, one is chosen arbitrarily.

Graham proved that LPT achieves an upper bound no worse than $(\frac{4}{3} - \frac{1}{3k}) \times optimal$ (Graham 1966). The greedy algorithm is optimal for $n \leq k + 2$ (Korf 2011).

3.1.2 Multi-Way Karmarkar-Karp (KK). The multi-way KK algorithm (Karmarkar and Karp 1982) extends the two-way Karmarkar-Karp 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 relative sums of each of the k sets the tuple represents. For example, the tuple (35, 33, 32) may correspond to three subsets with sums of 35, 33, and 32. Since only the relative sums matter, the tuple can be normalized by subtracting the minimum value. The tuple above is normalized to $(35 - 32, 33 - 32, 32 - 32) = (3, 1, 0)$. Each normalized 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 integer itself and the remaining integers are set to 0. Each of these tuples corresponds to putting the integer from S into one subset and nothing in the other subsets.

At each step of the KK algorithm, the first two tuples in the list, those whose largest integers are greatest, are combined. Given the tuples $A = (a_1, \ldots, a_k)$ and $B = (b_1, \ldots, b_k)$, both sorted in decreasing order, A is combined with B to form the new tuple C as $C = (a_1 + b_k,\ a_2 + b_{k-1}, \ldots, a_k + b_1)$.

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. The normalized C replaces both A and B; this combination process continues until only one tuple is left.

Example 3.1. 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 of these integers.

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 $(a_1, a_2, a_3) + \text{reverse}(b_1, b_2, b_3) = (a_1 + b_3, a_2 + b_2, a_3 + b_1)$. 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 $i = 4$, $(17, 12, 0) + \text{reverse}(11, 0, 0) = (17 + 0, 12 + 0, 0 + 11) = (17, 12, 11)$. 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. In this case, KK has found a perfect partition with cost 38.

3.2 Lower Bounds

We define 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 Largest Subset Sum. Dell'Amico and Martello (1995) define three lower bounds on solution cost.

- L_0: Sum(S), divided by k, and rounded up.
- L_1: The greater of L_0 and 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}. L_2 is the max of L_1 and the sum of these two integers.

Assuming that $S = \{s_1, s_2, \ldots, s_n\}$ 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 even moderate values of n.

3.2.2 On Any Subset Sum. The lower bound on any subset sum is the smallest sum for one subset such that if the remaining integers were partitioned perfectly into $k - 1$ subsets, their largest sum would be less than the upper bound ub (Korf 2009). 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 the 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 a sum smaller than this lb, then one of the remaining $k - 1$ subsets would be forced to have a sum greater than or equal to ub and thus could not lead to a better solution.

3.3 Complete Greedy Algorithm (CGA)

We now turn to algorithms that find optimal solutions to multi-way number partitioning. The simplest such algorithm is an extension of the (CGA for two-way partitioning (described in Section 2.4.1) to k-way partitioning. We consider this the obvious algorithm for finding optimal solutions and the previous state of the art before we began our work.

The CGA first sorts the integers of S into decreasing order. For k-way partitioning, the CGA searches a k-ary tree, where each level of the tree corresponds to a particular number. The k branches at each node assign the current number alternately to each of the different subsets. These k partial subsets are kept sorted in increasing order of current sum so that each number is assigned first to the subset with the smallest sum, then to the subset with the next larger sum, etc. Thus, the first complete solution found down the leftmost branch is the greedy solution.

If there are two subsets with the same sum, the current number is assigned to only one. For example, initially, all of the subsets are empty with zero sums, so the largest number is assigned just to the first subset. If an assignment to a subset creates a subset sum that equals or exceeds the largest subset sum in the best complete solution found so far, that branch is pruned from the tree. Finally, if the sum of the remaining unassigned integers plus the smallest current subset sum is less than or equal to the largest subset sum, all remaining integers are assigned to the subset with the smallest sum, terminating that branch of the tree.

3.4 Generating Complete Subsets

The CGA takes the input integers of S one at a time in decreasing order and alternately assigns them to one of the k subsets. By contrast, the remaining algorithms in this section construct complete subsets of the partition one at a time. We first construct all the first subsets S_1 that could possibly lead to a better overall partition; then, for each such subset, we recursively partition the remaining integers not assigned to the first subset $k - 1$ ways into $\langle S_2, \ldots, S_k \rangle$.

This guarantees an optimal solution for the following reason. For any given first subset, optimally partitioning the remaining integers $k - 1$ ways results in a complete partition whose largest subset sum is less than or equal to that of any complete partition that includes that first subset. Thus, applying this algorithm to all possible first subsets that could be part of an optimal partition results in an overall optimal partition. In fact, as we will see below, we do not need to optimally partition the remaining integers $k - 1$ ways.

Since we are constructing entire subsets at once, we need an algorithm that will generate all possible subsets whose sums lie within a given range.

3.5 Generating Subsets with Sums within a Given Range

When performing a branch-and-bound search, we generate the subsets of the partition $P = \langle S_1, S_2, \ldots, S_k \rangle$ one at a time. When generating candidates for subset S_i, given a set of remaining integers $S_i^R = S - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})$, a lower bound ($lb$) and upper bound ($ub$) on individual subset sums, we generate all subsets of S_i^R with sums within the range $[lb, ub - 1]$. We use $ub - 1$ since ub represents the cost of the best solution found so far, and we are only interested in better solutions.

Sections 3.5.1, 3.5.2, and 3.5.3 introduce our algorithms for generating all subsets with sums within a range. These are extensions of previously existing algorithms for solving the subset-sum problem. We use the notation S to denote the input set. Since we use these algorithms for generating candidate subsets for S_i, S will be some S_i^R for our purposes.

3.5.1 Inclusion-Exclusion (IE) Binary Tree Search. Perhaps the most straightforward way to generate subsets of S with sums within a range is using an inclusion-exclusion (IE) algorithm (Korf 2009). IE traverses a binary tree with each node representing a collection of subsets and the leaves correspond to individual complete 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 and 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 nonassigned integers is less than ub, we include all the remaining integers.

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 binary tree.

3.5.2 Extended Horowitz and Sahni (EHS). Given an input multiset $S = \{s_1, s_2, \ldots, s_n\}$ and a *target* value, the HS algorithm, described in Section 2.4.3, searches for a subset of S whose sum is as close as possible to *target*. The extended Horowitz and Sahni (EHS) algorithm (Korf 2011) 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 1.

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 the set pointed to by a unioned with each subset between those pointed to by b_1 and b_2 have sums within the bounds $[lb, ub - 1]$.

3.5.3 Extended Schroeppel and Shamir (ESS). The SS algorithm, described in Section 2.4.4, is based upon the HS algorithm but uses less memory. The extended Schroeppel and Shamir (ESS) algorithm (Korf 2011) 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 A in increasing order, while B_1 and B_2 are combined in a max heap to generate the subsets in B in decreasing order.

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 subset is generated from the min heap.

Since the subsets of the B set are generated from a max heap that 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 $sum(a \cup b) \geq lb$ 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 removed 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 $a \cup b$.

3.6 Sequential Number Partitioning (SNP)

SNP constructs the subsets of the complete partition one at a time. It first constructs all possible first subsets that could be part of a complete partition better than the best so far, then recursively partitions the remaining unassigned integers $k - 1$ ways.

3.6.1 Initial Upper Bound. The initial upper bound for SNP is computed using the KK polynomial-time approximation algorithm and its value is stored in the variable ub. SNP then seeks to improve this bound until it finds and verifies an optimal partition.

3.6.2 Eliminating Duplicate Partitions. Two partitions that differ only in the order of the subsets are obviously equivalent. For example, the 2-way partition $\langle\{8, 7\}, \{4, 5, 6\}\rangle$ is equivalent to the partition $\langle\{4, 5, 6\}, \{8, 7\}\rangle$. The subsets of any k-way partition can be permuted $k!$ ways. In order to prevent separately computing each of these permutations, we require that each subset constructed contain the largest of the remaining unassigned integers.

3.6.3 Weakest-Link Optimality. Given a first subset, optimally partitioning the remaining unassigned integers $k - 1$ ways will result in an optimal partition among those that include that first subset, but it can do more work than necessary. Since the cost of a complete solution is the largest subset sum, all we need to do to ensure an optimal partition among those including the first subset is to partition the remaining integers $k - 1$ ways such that each of the subset sums is less than or equal to the sum of the first subset. More generally, we have to partition only the remaining unassigned integers so that the largest subset sum of the remaining integers is less than or equal to the maximum sum of the subsets already constructed.

3.6.4 Constructing the Individual Subsets. Each recursive call to the partitioning algorithm has five parameters: S, the set of remaining integers to be partitioned; k, the number of subsets to partition S into; lb, the lower bound on the subset sums; ub, the upper bound on the subset sums, which is equal to the best complete partition found so far; and max, the maximum subset sum of the sets already constructed in the current partial partition. It returns the maximum subset sum of the best complete partition that it finds.

Each recursive call constructs each possible subset that contains the largest number in S and whose sum is greater than or equal to lb and strictly less than ub. For each such subset, it recursively calls itself with the remaining unassigned integers, one fewer subset, lb, ub, and the larger of max and the sum of the current subset. If the recursive call successfully completes a new partition with maximum subset sum less than ub, it updates ub to the new largest subset sum and lb correspondingly, then continues with the next subset within the new range. If it fails to complete a full partition with a cost less than ub, it also continues with the next subset but with the current bounds.

The basic subroutine of SNP is constructing a subset from a set of integers whose sum lies within a given range. As described above, there are three ways to do this. The simplest way, described in Section 3.5.1, is to search an IE binary tree of the remaining integers. A more complex algorithm is to use the EHS described in Section 3.5.2. Finally, we have the ESS, described in Section 3.5.3. ESS dominates EHS both in terms of runtime and memory used. However, neither IE nor ESS dominate the other.

The worst-case runtime of IE is $O(2^n)$ for n integers, but its simplicity means that it has a very small constant factor. The runtime of ESS is $O(n \cdot 2^{n/2})$, but its relative complexity results in a larger constant factor. Thus, for small values of n, IE is faster, whereas for larger values of n, ESS is faster. Since IE is a linear-space algorithm, it uses almost no memory, whereas ESS requires $O(2^{n/4})$ memory. This makes it practical up to about $n = 120$, beyond which we would have to revert back to IE, but we never approach this limit in our experiments.

We refer to SNP using IE to generate the subsets as SNPIE, and using ESS to generate the subsets as SNPESS.

3.6.5 Dominance Pruning for SNPIE. When running IE to generate subset S_i of the current partition $\langle S_1, S_2, \ldots, S_k \rangle$, the sum of the integers of S_i must be within the range $[lb, ub - 1]$. The remaining integers 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 they 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; thus, IE now excludes 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, however, since in any partition that contains $S_1 = \{15, 4, 2\}$, the 4 and 2 could be swapped for the 6 without changing the partition cost. Any place where 6 could fit in subsequent subsets $\langle S_2, S_3, \ldots, S_k \rangle$, 4 and 2 could also fit. Therefore, since all partitions that include $\{15, 6\}$ have already been considered, $S_1 = \{15, 4, 2\}$ is dominated and does not need to be considered. In general, once a number is excluded from a set, the sum of the integers included in that set must exceed the excluded number.

More formally, let x be 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 a 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 (Martello and Toth 1990a, 1990b).

Note that this dominance pruning is not used with SNPESS, since the individual subsets are generated in a completely different way.

3.7 Recursive Number Partitioning

SNPIE and SNPESS are based on two of our earlier algorithms called recursive number partitioning (RNP) (Korf 2009) and improved recursive number partitioning (IRNP) (Korf 2011). While the SNP

Table 3. Difference Between Bin Packing and Number Partitioning

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

algorithms construct subsets one at a time, RNP and IRNP recursively partition the original set into two partitions. RNP uses IE to do the recursive two-way partitions while IRNP uses ESS. While RNP and IRNP have been important for motivating further research, they are dominated by SNPIE and SNPESS; thus, we do not describe them here. See the conference papers for further details.

3.8 Experimental Results: CGA, SNPIE, and SNPESS

To show the empirical differences between CGA, SNPIE, and SNPESS, 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 $n = 20$ to 45. We chose to generate 48-bit integers in order to generate hard instances without perfect partitions (Korf 2011). All experiments were run on an Intel Xeon X5680 CPU running at 3.33GHz.

Figure 8 reports the average runtimes for CGA, SNPIE, and SNPESS to partition each of the problem instances into $k = 3$ to 12 subsets. CKK is not compared, as the CGA dominates CKK for multi-way partitioning (Korf 2009). SNPESS outperforms SNPIE for $k \leq 6$ while SNPIE outperforms SNPESS for $k \geq 7$.

For $k = 3$ to 5, SNPESS dominates SNPIE and the ratio of their runtimes increases with n. For $k = 6$, SNPIE is fastest for small n, but then SNPESS dominates for $n \geq 34$. For $k = 7$, $k = 8$, and $k = 9$, SNPIE dominates for the integers reported, but as n increases, the ratio decreases, suggesting that if n were to get large enough, SNPESS would dominate. For $k = 10$, 11, and 12, SNPIE dominates.

Note that, for $n = 34$ and $k = 9$, the average times for both SNPESS and SNPIE do not follow a trend with the rest of the series for $k = 9$. This is due to the results for one problem that took 96455.9 seconds for SNPESS and 1572.27 seconds for SNPIE. This shows how sensitive these algorithms can be to individual problem instances.

The average runtime of the CGA, the previous state of the art, is dominated by both SNPIE and SNPESS for all n and k. Depending on n and k, there is between two and six orders of magnitude speedup. As n increases, the ratio of the runtimes increases, strongly suggesting that both SNPIE and SNPESS are asymptotically faster.

4 BIN-PACKING ALGORITHMS FOR NUMBER-PARTITIONING PROBLEMS

Given an input multiset $S = \{s_1, s_2, \ldots, s_n\}$ of n positive integers representing item weights and a positive integer capacity C, the bin-packing problem is to 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 article uses the terminology "subsets" and "subset sums" for both bin packing and number partitioning.

This section describes the relationship between bin packing and number partitioning, focusing on solving multi-way partition problems (see Section 1.1) using bin-packing algorithms. We first describe 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.

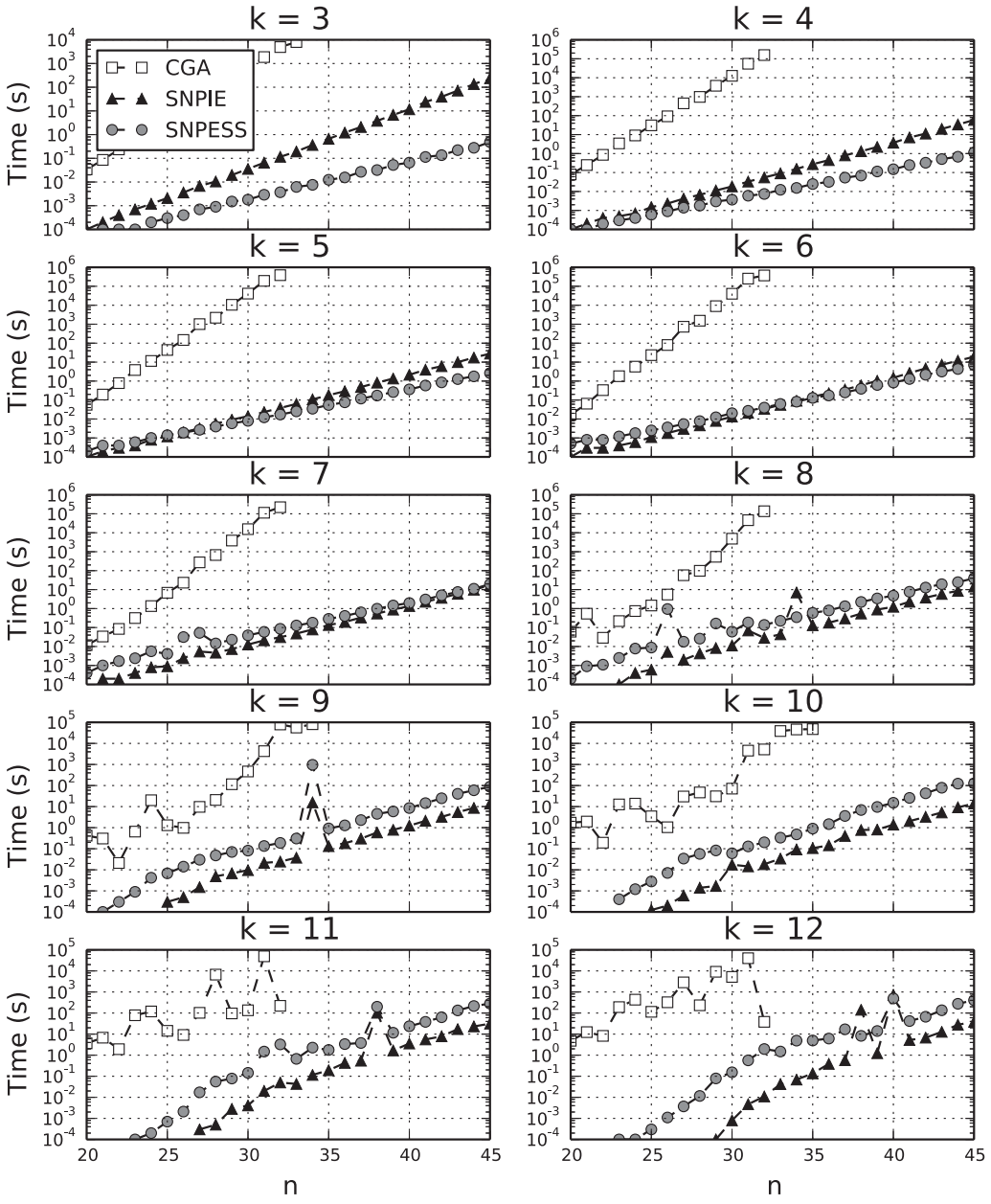


Fig. 8. The average time in seconds to optimally partition 48-bit integers 3 through 12 ways using the CGA, SNPIE, and SNPESS.

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, highlighting the differences:

Multi-way Number Partitioning
Given an input multiset $S = \{s_1, s_2, \ldots, s_n\}$ of positive integers and a **number of subsets k**, separate the integers of S into k mutually exclusive and collectively exhaustive subsets, **minimizing the largest subset sum**.

Bin Packing
Given an input multiset $S = \{s_1, s_2, \ldots, s_n\}$ 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**.

In some sense, bin packing and number partitioning are dual problems. Bin packing fixes the maximum subset sum and minimizes the number of subsets, while number partitioning fixes the number of subsets and minimizes the largest subset sum.

4.2 Lower Bounds

A lower bound (lb) function for bin packing calculates the minimum number of subsets with sums 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 (1990b) introduced two bin packing lower bounds which they refer to as L_1 and L_2. We cover L_1 here and refer the reader to the original paper for an explanation of L_2. Both algorithms first sort the input set S in decreasing order.

4.2.1 L_1 Lower Bound. The lower bound L_1 relaxes 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 4.1. Consider the input set $S = \{99, 97, 94, 91, 8, 5, 4\}$ and the bin capacity $C = 100$. (This example is taken from Korf (2002).) The L_1 bound is $L1 = \lceil \frac{\text{sum}(S)}{C} \rceil = \lceil \frac{398}{100} \rceil = 4$.

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 (Johnson 1973; Garey et al. 1972). 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 Coffman Jr et al. (1997).

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 which it fits. If it does not fit into any of the bins in the list, it is placed in an empty bin and appended to the list. The number of bins used is returned. Yue (1991) proved that FFD achieves an upper bound no worse than $11/9\ OPT + 1$. Dósa (2007) later proved a tighter bound of $11/9OPT + 6/9$.

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 with the least remaining capacity in which it fits. If it does not fit into any of the bins, it is placed in an empty bin and appended to the list. The number of bins used is returned. BFD has the same solution quality guarantee as FFD, but for some problem instances produces a better bound on the number of bins required (Johnson et al. 1974).

4.4 MULTIFIT: Solving Number-Partitioning Problems with Bin-Packing

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 (Coffman Jr et al. 1978). 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_C) and upper (ub_C) bounds on C^*, it performs a binary search over the bin capacities between lb_C and ub_C. Each probe of the binary search sets C to a value within the range $[lb_C, ub_C - 1]$ and uses FFD (see Section 4.3.1) to approximate the number of subsets needed to pack S into subsets with a 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. We are aware of two bin-packing algorithms: our algorithm, called improved bin completion (IBC), and the operations research algorithm BCP, which have been used with MULTIFIT to optimally solve number-partitioning problems. In the next two sections, we describe these algorithms at a high level. For information on other optimal bin-packing algorithms in general, see Fukunaga and Korf (2005), Schoenfield (2002), and Martello and Toth (1990a).

Algorithm 1 shows the source code for MULTIFIT. The function packBins is a call to either FFD, IBC, or BCP depending on which version of the algorithm is being run.

ALGORITHM 1: MULTIFIT

Input: Input S, number of subsets k, lower bound lb_C, upper bound ub_C
Output: The partition cost

```
while ub_C > lb_C do
    C = (ub_C + lb_C)/2;
    numBins =packBins(S, C);
    if numBins > k then
        lb_C = C + 1;
    else
        ub_C = C;
return max;
```

4.5 Bin Completion (BC and BSIBC)

Classic optimal bin-packing algorithms such as those of Eilon and Christofides (1971) or Martello and Toth (1990a) consider input integers one at a time and assign them to bins. These are item-oriented branch-and-bound algorithms beyond the scope of this article.[2] In contrast, bin completion (BC) (Korf 2002, 2003; Schreiber and Korf 2013) considers bins one at a time and assigns a complete set of integers to them. This is a bin-oriented branch-and-bound algorithm.

BC computes its initial upper bound on the number of bins required using BFD (see Section 4.3.2) and lower bound using Martello and Toth's L_2 wasted space heuristic (Martello and Toth 1990b). If these bounds are equal, the BFD solution is returned as optimal. Otherwise, a tree search is performed to either prove that the initial upper bound is optimal or to find the optimal solution with a smaller value.

[2]Korf found bin completion to be asymptotically faster than item-oriented approaches (Korf 2002, 2003).

A *feasible* set is a set of input integers witha sum less than or equal to the bin capacity 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, and second, to shrink the remaining capacity of the bin, which results in fewer feasible bin completions to consider.

At each node of the search tree, BC generates all feasible completions containing the largest remaining integer and branches on each. Branches can be pruned when the cost of a solution through the current node is guaranteed to be at least as large as the best solution found so far. Furthermore, some completions are dominated by other completions. We refer the reader to the papers cited above for the details of these pruning and dominance rules.

In 2013, we improved BC for use with MULTIFIT, calling our new algorithm improved bin completion (IBC) (Schreiber and Korf 2013). The optimal number-partitioning algorithm using MULTIFIT and IBC is called binary search improved bin completion (BSIBC). We refer readers to Schreiber and Korf (2013) for further details.

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

BCP (Padberg and Rinaldi 1991; Belov and Scheithauer 2006) is an operations research algorithm for solving an integer linear program. The core of BCP is linear programming (Chvatal 1983; Dantzig and Thapa 1997, 2003), a technique for solving a maximization or minimization problem given a set of constraints in the form of linear inequalities. The time complexity for 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 integers. 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 (Dantzig and Wolfe 1960; Amor and de Carvalho 2005) is used. Finally, there is an optimization for integer linear programming that adds constraints to cut down the feasible region for optimal solutions, called cutting planes (Gomory 1958).

Dell'Amico et al. (2008) proposed an optimal number-partitioning algorithm using BCP as the bin-packing solver for MULTIFIT. Schreiber and Korf (2013) also used MULTIFIT with Belov's BCP algorithm to solve number-partitioning problems.

4.7 Experimental Results: BSBCP, BSIBC, SNPIE, and SNPESS

We ran the MULTIFIT algorithms binary-search branch-and-cut-and-price (BSBCP) and binary-search improved bin completion (BSIBC) against the same dataset used to test the branch-and-bound algorithms, SNPIE and SNPESS (see Section 3.8). 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 value of n, which ranges from $n = 20$ to 45. All experiments were run on an Intel Xeon X5680 CPU at 3.33GHz.

Figure 9 reports the average runtimes for BSBCP, BSIBC, SNPIE, and SNPESS to partition each of the problem instances into $k = 3$ to 12 subsets. BSIBC outperforms BSBCP for $k \leq 8$. Both branch-and-bound algorithms outperform both MULTIFIT algorithms for $k \leq 8$. For $k = 9$ through 12, BSIBC outperforms BSBCP for small n while the results swap for larger n. SNPIE is the dominant algorithm for $k = 9$ and $k = 10$. For $k = 11$ and $k = 12$, BSBCP is the best algorithm for high n.

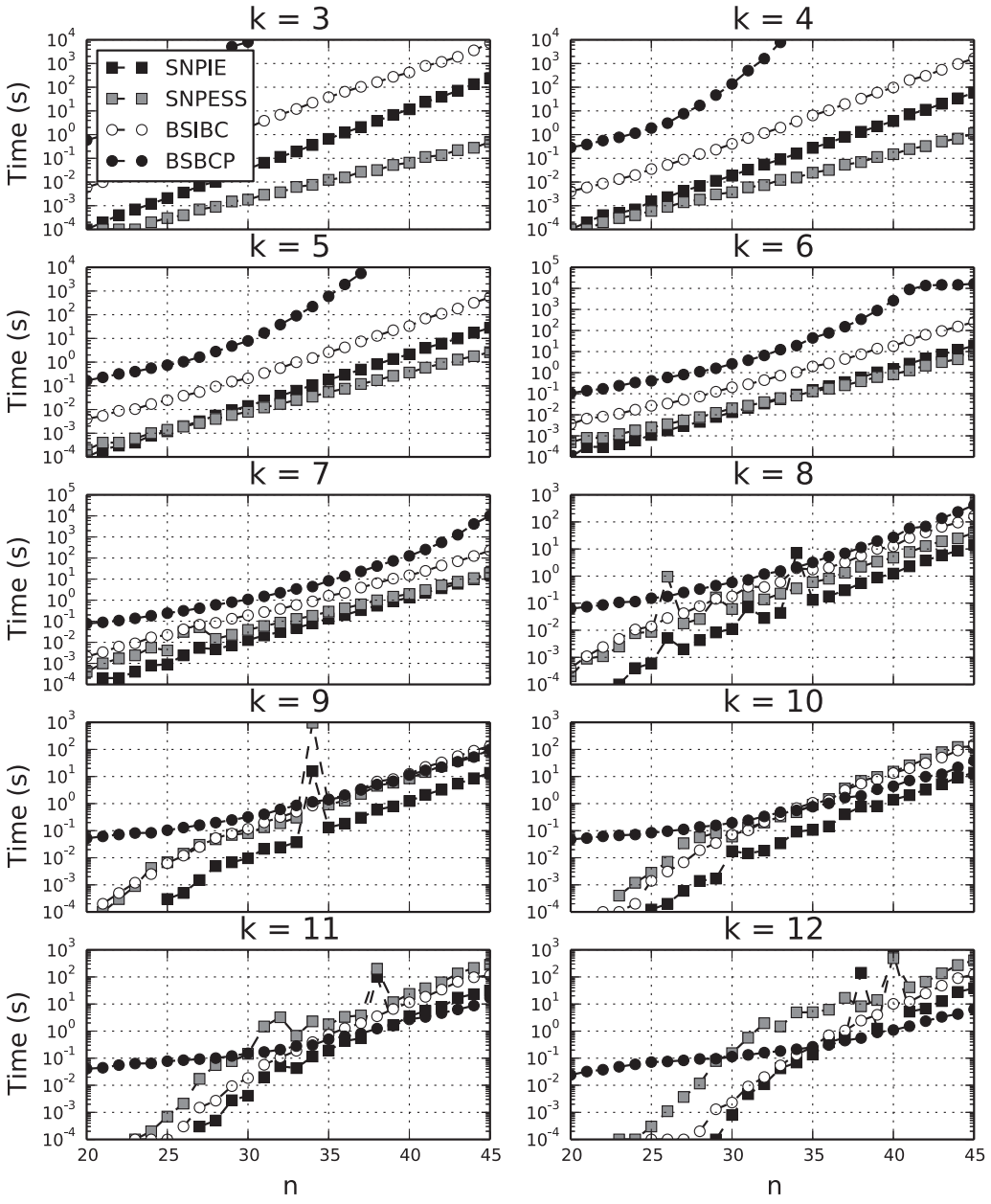


Fig. 9. The average time in seconds to optimally partition 48-bit integers 3 through 12 ways using BSIBC, BSBCP, SNPIE, and SNPESS.

4.8 Solving Bin-Packing Problems with Number-Partitioning Solvers

It is also possible to solve bin-packing problems using number partitioning algorithms. Given an input set S and bin capacity C, we use an approximation such as FFD or BFD to generate an upper bound on the number of bins in an optimal packing. We then use a number-partitioning algorithm with k set to this value. If this algorithm finds a partition with a value less than or equal to C, we

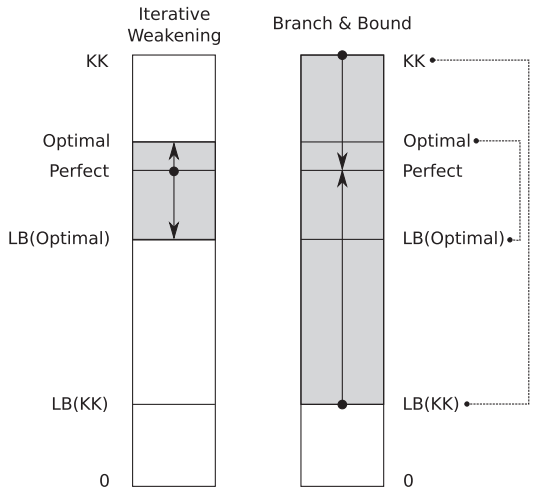


Fig. 10. A comparison of the search spaces of iterative weakening and branch-and-bound. Iterative weakening starts by theorizing a perfect partition and increases this upper bound while decreasing the implied lower bound until an optimal partition is found. Branch-and-bound calculates an upper bound 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.

then try packing the integers into one fewer bins. This continues until the number-partitioning algorithm finds an optimal partition with a cost greater than C. The fewest number of bins for which the number-partitioning algorithm can find a partition with a 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 article.

5 CACHED ITERATIVE WEAKENING

Before cached iterative weakening (CIW), work on multi-way number partitioning fit into one of two classes. The CGA (see Section 3.3) and SNP (see Section 3.6), including both SNPIE and SNPESS, are branch-and-bound algorithms. BSIBC and BSBCP (see Section 4) 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 to be optimal. In contrast, CIW starts with a lower bound and iteratively increases it until an optimal partition is found. The first complete partition found is optimal.

Let C^* be 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 (see 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 optimality by proving that there is no partition with all subsets having sums less than C^*. In contrast, CIW considers only partitions with a cost less than or equal to C^*.

For each partial partition, the previous algorithms generate the next subsets using exponential algorithms. SNPESS uses ESS to generate the next subsets while SNPIE uses IE. In contrast, CIW usually 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 P^*, $\text{cost}(P^*) = \lceil \text{sum}(S)/k \rceil$, 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 $\text{cost}(P^*)$.

In order to consider only subsets with sums less than or equal to C^*, CIW uses 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 $\text{cost}(P^*)$ and calculates lb_1 based on ub_1. (We will discuss how to find this smallest existing subset in Section 5.2.) 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 (Provost 1993). 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 a cost within the range $[\text{cost}(P^*), C^* - 1]$. Iterative weakening, on the other hand, **only** explores partial partitions with costs between $\text{cost}(P^*)$ and C^*.

Suppose that we could efficiently generate subsets one by one in sum order starting with $\text{cost}(P^*)$. 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 to 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.

5.2 Precomputing: Generating Subsets in Sum Order

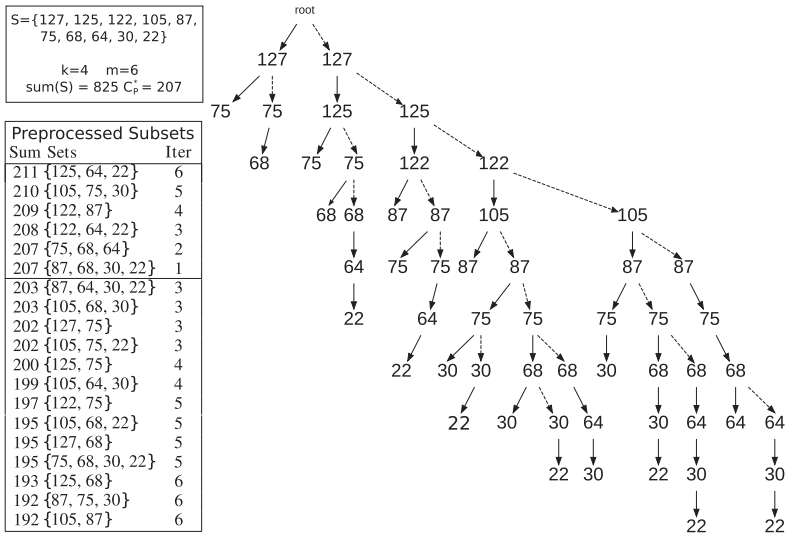
We are not aware of an efficient algorithm for generating subsets one by one in sum order. Instead, we describe an algorithm for efficiently generating and storing the m subsets with the smallest sums greater than or equal to $\text{cost}(P^*)$, the cost of a perfect partition, for any constant m.

Let max be the m^{th} smallest subset sum greater than or equal to $\text{cost}(P^*)$. 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 $[\text{cost}(P^*), max]$ and all subsets with sums within the range $[min, \text{cost}(P^*) - 1]$.

Section 3.5 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 $[min, max]$. We do not know the values of min and max before generating the m subsets with sums greater than or equal to $\text{cost}(P^*)$.

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 3.5 for this purpose. We put each subset found with a sum within the range $[min, \text{cost}(P^*) - 1]$ into the min-heap and those in the range $[\text{cost}(P^*), max]$ 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 then continue searching for all subsets with sums in the new range $[min, max]$. Each time a subset with a sum greater than or equal to $\text{cost}(P^*)$ 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 a sum less than min from the min-heap.



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

Fig. 11. A complete cached inclusion-exclusion tree for the input set S and the bounds $lb_6 = 192, ub_6 = 211$ containing all of the subsets listed in the table on the bottom left.

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 cost(P^*). If m iterations are performed without finding an optimal partition, the algorithm is run again from scratch to generate the next $2m$ subsets. If the $2m$ subsets are exhausted without finding an optimal partition, then $4m$ subsets are generated, then $8m$, etc. Thus, m is a parameter of CIW. We discuss setting m experimentally in Section 5.6.

ESS has a runtime that is the square root of the runtime of IE and memory usage that 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 5.1. Consider the example number-partitioning problem with $S = \{127, 125, 122, 105, 87, 75, 68, 64, 30, 22\}$ and $k = 4$. Figure 11 shows 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 cost(P^*) = 207. There are 13 sets with sums within the range $[min, \text{cost}(P^*) - 1]$ shown below the horizontal line and 6 sets with sums within the range $[\text{cost}(P^*), max]$ 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 $[lb_{it}, ub_{it}]$. The smallest subset sum greater than or equal to cost(P^*) is 207. There are two subsets with this value; we arbitrarily choose to examine $\{87, 68, 30, 22\}$ first. We set $ub_1 = 207$ and $lb_1 = 825 - 3 \times 207 = 204$. There are no subsets with sums less than cost(P^*) within this range; thus, the iteration terminates. Iteration would consider both subsets having sum 207. However, since they overlap, this iteration also terminates. The third iteration has $ub_3 = 208$ and $lb_3 = 825 - 3 \times 208 = 201$ with seven sets in range: namely, the rows with $Iter$ equal to 1, 2, or 3. 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 a sum at least as large as cost(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 $\langle S_2, \ldots, S_k \rangle$.

This task is very similar to the search of the recursive partitioning tree used by SNP (both SNPIE and SNPESS), described in Section 3.6. The difference is that CIW is searching for any complete partition since it is guaranteed to be optimal, while SNP must search for the lowest-cost complete partition.

However, using SNP 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, SNPIE repeatedly calls IE, and SNPESS repeatedly calls ESS, 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 that contain all the integers of S. For each first subset S_1 in A, copy all subsets of A that do not contain any integers in S_1 into a new array B. Then, recursively try to select $k-1$ disjoint subsets from B that contain the remaining integers of $S - S_1$. If $k = 0$, return true; 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 that performs the same function.

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 slightly more complicated.

CIE trees are similar to IE trees (see Section 3.5.1). IE searches an implicit tree that represents 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 $[lb_{it}, ub_{it}]$. Each node of the CIE tree corresponds to one of the integers in S. The integer is included on the left branch and excluded on the right.

Figure 11 depicts the CIE tree for iteration six of an example four-way partitioning problem of the set $S = \{127, 125, 122, 105, 87, 75, 68, 64, 30, 22\}$. For iteration six, $S_1 = \{125, 64, 22\}$, $ub_6 = \text{sum}(S_1) = 211$, and $lb_6 = 192$. The list of all 19 subsets of S with sums in the range $[lb_{it}, ub_{it}]$ is shown at the bottom left. The 19 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, the leftmost branch follows 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, the rightmost branch, $root \dashrightarrow 127 \dashrightarrow 125 \dashrightarrow 122 \dashrightarrow 105 \dashrightarrow 87 \rightarrow 75 \rightarrow 68 \dashrightarrow 64 \rightarrow 30 \rightarrow 22$, corresponds to the subset $\{75, 68, 30, 22\}$.

Given this CIE tree, we can use the SNP algorithm as described in Section 3.6 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 using IE binary-tree search or ESS to construct subsets, we search the CIE tree instead. Since the CIE tree contains only subsets that are within the bounds, it is much smaller and thus faster to search.

5.3.3 Cached Inclusion-Exclusion (CIE) Trees. We now discuss the full version of 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 that 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, are found in one tree. In contrast, there is one CIE tree for each unique cardinality of complete subsets. The distribution of the cardinalities of the subsets with sums in the range $[lb_{it}, ub_{it}]$ is not uniform. The average cardinality of a subset in any complete partition is n/k. Typically, most subsets in an optimal partition have cardinalities 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 that CIW never has to examine the higher cardinality subsets. In Figure 12, there are separate trees for subsets of cardinality 2, 3, and 4 storing all subsets with sums within the range $[192, 211]$, which is every subset in iteration 1 through 6.

IE searches an implicit tree, meaning that only the recursive stack of IE is stored in memory. In contrast, the 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; thus, 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 precomputed 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 that 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. Let $card$ be the cardinality of the tree that CIE is searching to generate S_d in partial partition $\langle S_1, \ldots, S_d \rangle$. 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 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 $k-d$ disjoint subsets, each with cardinality $\geq card$.

5.3.5 Avoiding Duplicates. Choosing subsets in cardinality order avoids many duplicate partitions that differ only by a permutation of the subsets. 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 5 of Iterative Weakening

Figure 13 shows the recursive partitioning search tree for iteration 5 of our running example. The S_is are generated using the iteration 5 CIE trees from Figure 12. In this example, when we search

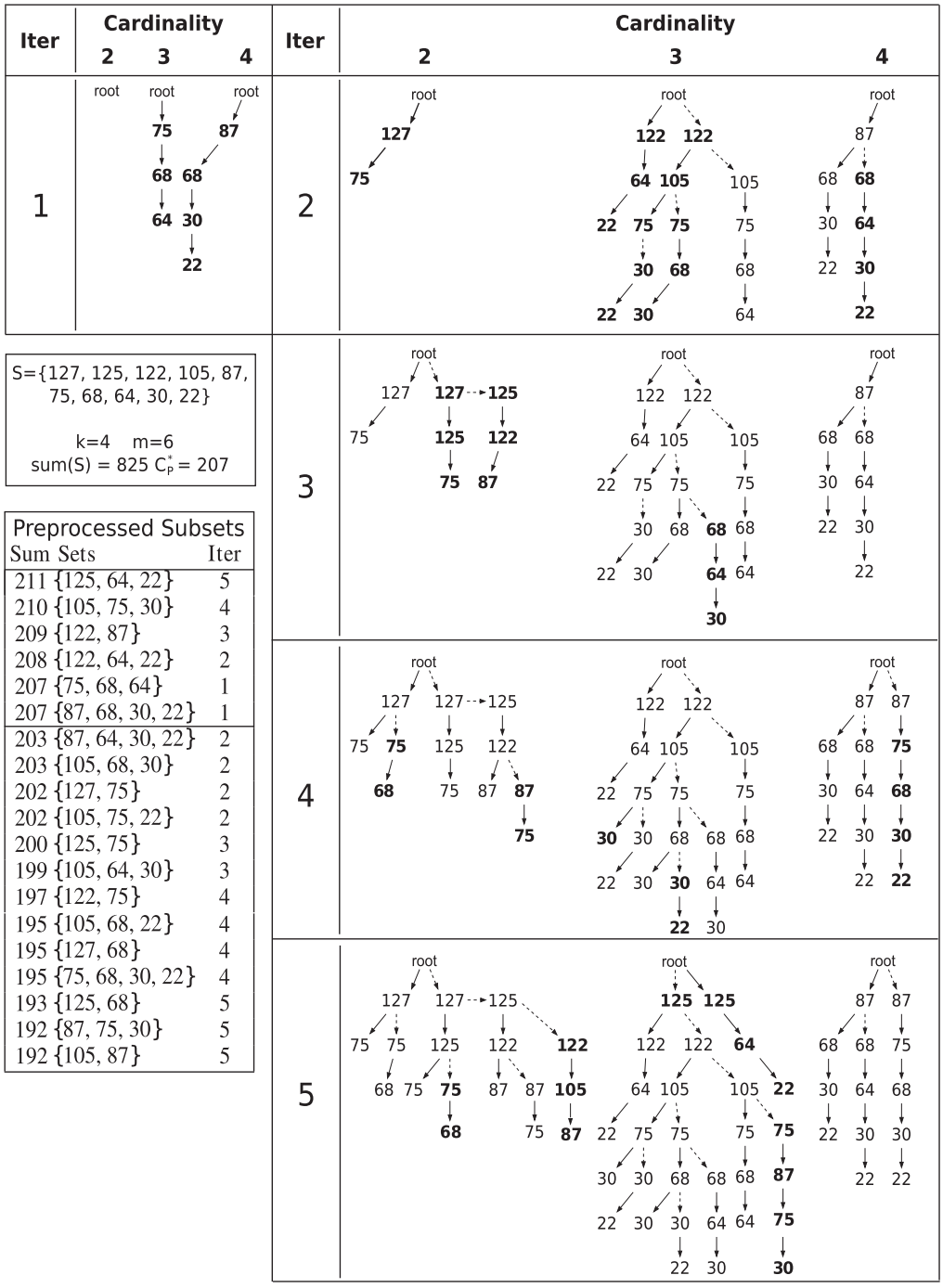


Fig. 12. CIW example with the list of preprocessed subsets sorted by sum and cached inclusion-exclusion trees for cardinalities 2, 3, and 4 during iterations 1, 2, 3, 4, 5 and 6. The bold integers were added during that iteration.

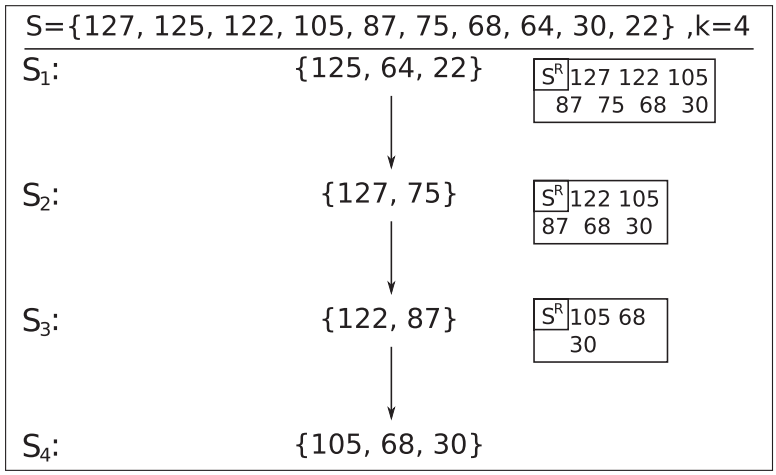


Fig. 13. The recursive partitioning search tree for iteration 5, described in Section 5.4.

CIE trees, refer to Figure 12. When we refer to subset S_i or the remaining integers S^R, refer to Figure 13.

The root of the tree is the subset $S_1 = \{105, 75, 30\}$, whose sum 210 is the upper bound ub_5 for iteration 5 of iterative weakening. The search of candidate subsets for S_2 begins in the cardinality 2 CIE tree of Figure 12 at iteration 5, 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; thus, we exclude 127 and then include 125. Next, we cannot include 75 since it is not in S^R; therefore, 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_5; thus, we prune. Note that the set {125, 64, 22} is not in the cardinality 3 CIE tree since its sum is greater than ub_5. 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 and 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 3 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 then 75 is not in S^R; therefore, 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 since there is no exclusion branch to exclude 122, we must 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.

5.5 Example: Iteration 6 of Iterative Weakening

In iteration 6, iterative weakening sets $S_1 = \{125, 64, 22\}$ with $ub_6 = \text{sum}(S_1) = 211$ and adds the four rows with $Iter = 6$ from the table in Figure 12 to the CIE trees. Figure 14 shows the recursive partitioning search tree for iteration 6. CIW partitions S into $k = 4$ subsets all with

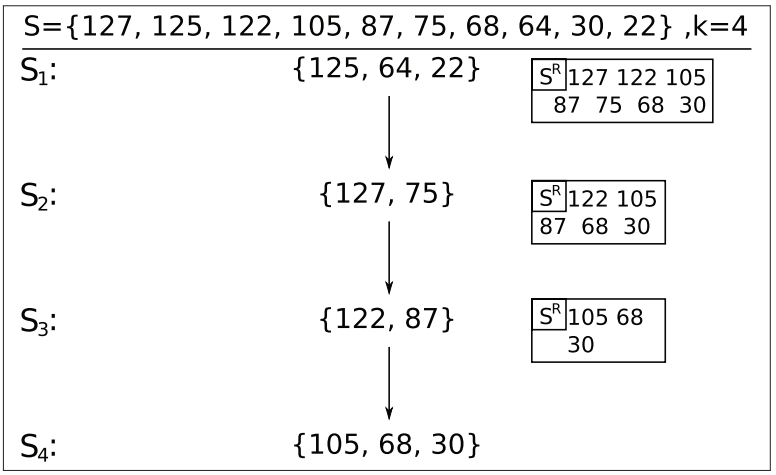


Fig. 14. The recursive partitioning search tree for iteration 6, described in Section 5.5.

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 compared to the branch-and-bound and MULTI-FIT algorithms empirically, we ran CIW against the same dataset used to test the other 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 value of n, but this time, because of the improved performance of CIW, n ranges from $n = 20$ to 60 as opposed to $n = 20$ to 45. All experiments were run on an Intel Xeon X5680 CPU at 3.33GHz.

Generally, for $n = 20$ to 60, among the previous algorithms discussed, SNPESS (see Sections 3.5.3 and 3.6) was the best performer for $k = 3$ to 7, SNPIE (see Sections 3.5.1 and 3.6) for $k = 8$ to 10, and BSBCP (see Section 4.4) for $k = 11$ and 12. Thus, we compare CIW to SNPESS for $k \leq 7$, to SNPIE for k from 8 to 10, and to BSBCP for $k = 11$ and 12. These results are presented in Figure 15.

For a particular k, since the performance depends on n, one might infer that a hybrid recursive algorithm that uses one of SNPESS, SNPIE, or BSBCP for recursive calls depending on n and k would outperform the individual algorithms. However, Korf et al. (2014) show that such a hybrid algorithm in fact does not significantly outperform the best of the individual algorithms.

For CIW, we must choose a value for m, the number of subsets to initially generate during the precomputing phase (see Section 5.2). We want m to be the number of subsets with sums within the range $[\text{cost}(P^*), C^*]$, but we do not know this number in advance. If m is too small, CIW has to run ESS multiple times. If m is too large, CIW wastes 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. Let m_i be the number of subsets with sums within the range $[\text{cost}(P^*), C^*]$ 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 $(n = 59; k = 12)$.

CIW is faster than SNPESS for $k \geq 4$ and $n > 40$. SNPESS is faster than CIW for $k = 3$. For fixed n, SNPESS 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 runtimes of SNPESS to CIW tend to grow as n gets larger, suggesting

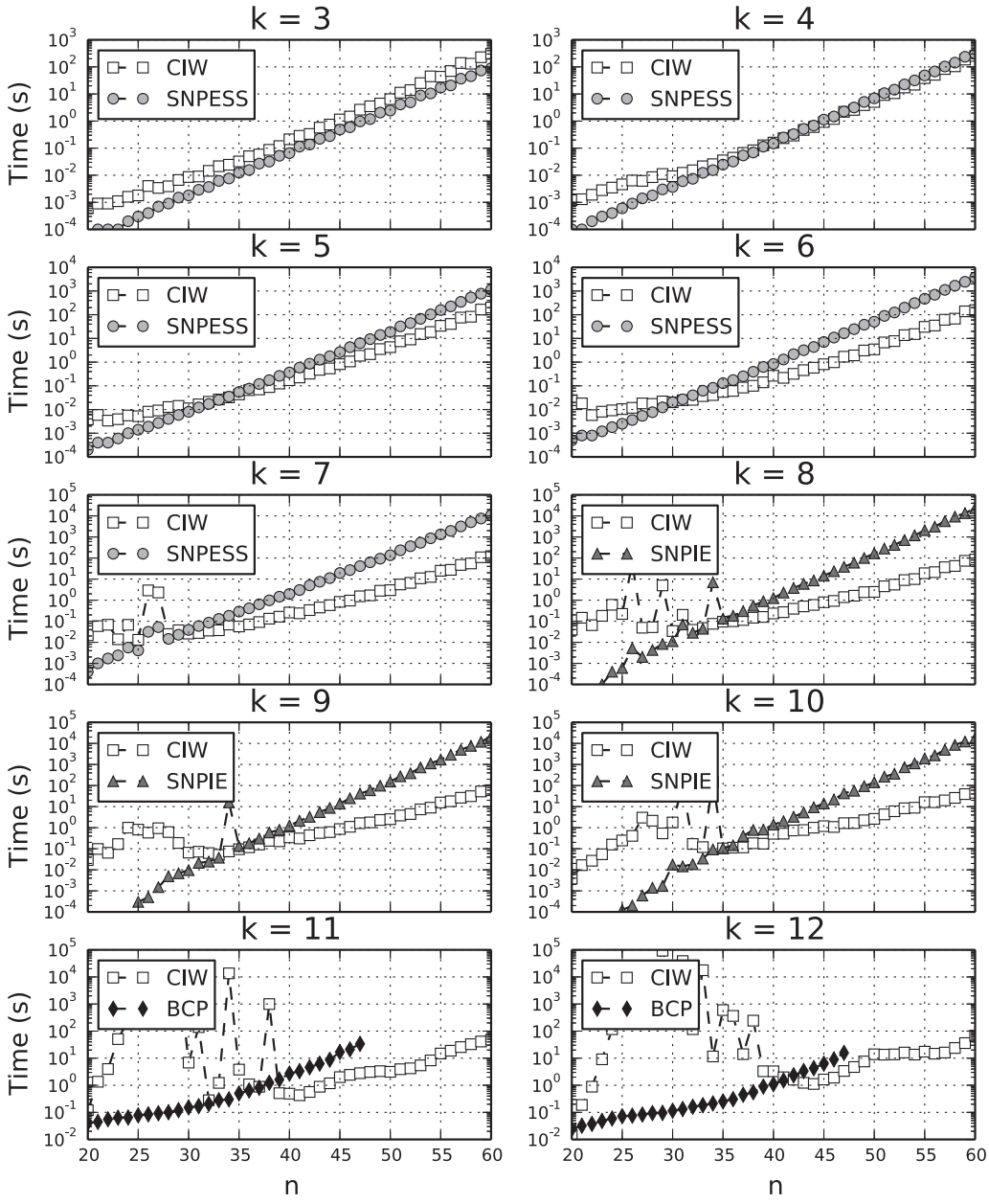


Fig. 15. The average time in seconds to optimally partition 48-bit integers 3 through 12 ways using CIW. For each k, CIW is compared to the previous best algorithm at the time that CIW was invented.

that CIW is asymptotically faster than SNPESS. For $k = 3$ and $k = 4$, there is no clear trend. The biggest difference in the average runtimes is for $(n = 58; k = 7)$, in which SNPESS takes 91 times longer than CIW.

For k from 8 to 10, we compare CIW to SNPIE and for 11 and 12, we compare CIW to BSBCP. CIW outperforms both SNPIE and BSBCP for all k and $n > 40$ in these ranges. The ratios of the

runtimes of SNPIE to CIW and BSBCP to CIW grow as n gets larger for all k, again suggesting that CIW is asymptotically faster than SNPIE and BSBCP. The biggest difference in the average runtimes of SNPIE and CIW is for $(n = 60; k = 10)$, in which SNPIE takes 400 times longer than CIW. The biggest difference for BSBCP is for $(n = 60; k = 11)$, in which BSBCP takes 250 times longer than CIW.

For very small n, CIW is not as fast as the other algorithms. For small k, this is owing to the overhead of Schroeppel and Shamir. However, the difference between CIW and the other algorithms is a tiny fraction of a second; thus, it is immaterial. For $k \geq 10$, there is an exponential explosion using CIW to solve some instances with small n, leading to a very large average runtime. For these instances, the optimal solution cost turns out to be much greater than the perfect partition cost. This leads to many iterative weakening steps and a slow runtime for instances that are trivial with the other algorithms. However, as n grows, CIW dominates. Future work will explore how to mitigate this issue with iterative weakening and instances that are trivial with the other algorithms.

There is memory overhead for CIW owing to both ESS and the CIE trees proportional to the number of subsets with sums in the range $[\text{cost}(P^*), C^*]$. All of the experiments require less than 4.5GB of memory and 95% require less than 325MB. However, with increased n, memory will become a limiting factor as well as time. Better understanding of and reducing memory usage is the subject of future work.

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

5.7 Summary

This section has introduced CIW, a state-of-the-art algorithm for multi-way number partitioning. Other algorithms for number partitioning are either based on branch-and-bound (see Section 3) or binary-search over bin-packing problems (see Section 4). These algorithms begin by calculating an approximate partition to generate lower and upper bounds. They then shrink these bounds until the optimal partition is found. In contrast, CIW uses iterative weakening to search for the optimal partition. It starts by theorizing that a perfect partition is possible and sets the upper bound to $\text{cost}(P^*)$, 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 other algorithms 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 in range. The iterative weakening along with the caching in concert makes CIW orders of magnitude faster than the other algorithms for multi-way partitioning.

6 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 the Moffitt algorithm (MOF). We start by describing the weaknesses of the previous algorithms and then describe how LCS improves upon them.

6.1 Weakness of CIW

While performing cached iterative weakening, each subset with a sum in the range $[\text{cost}(P^*), C^*]$ is considered in ascending sum order as the first subset S_1 of partitions $\langle S_1, \ldots, S_k \rangle$. 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 the cardinalities of each 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 the remaining subsets to have relatively low cardinality. In general, it is harder to find low cardinality subsets with sums in a particular range since there are many fewer such subsets.

While performing iterative weakening, many of the S_1 subsets whose sum is within the range $[\text{cost}(P^*), ub_{it}]$ 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 that 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 toward the leaves where they often never have to be considered because of the cardinality pruning rule discussed in Section 5.3.4.

6.2 Weakness of Branch-and-Bound Algorithms

A weakness of branch-and-bound algorithms such as SNPIE and SNPESS 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 sums 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 precomputed CIE trees is much faster than using ESS or IE to generate 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.

6.3 Low-Cardinality Horowitz and Sahni

The average cardinality of any complete partition of n integers into k subsets is n/k. We arbitrarily 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. The algorithm performs about the same with other similar definitions of low cardinality, such as $\lceil n/k \rceil$.

In order to perform a low-cardinality search, we must be able to generate low-cardinality subsets with sums within the range $[lb_{it}, ub_{it}]$. Section 3.5.2 describes the EHS algorithm for generating all subsets with sums within this range. Low Cardinality Horowitz and Sahni (LCHS) modifies EHS to generate subsets within the range $[lb_{it}, ub_{it}]$ but 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 $n/2$ integers in S while S_B consists of all subsets of the smallest $n/2$ numbers in S. LCHS 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 LCHS 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.

6.4 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 step used to choose subset S_1 considers only low-cardinality subsets generated using LCHS. 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 1

Phase 1 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 cost(P^*). In contrast, LCS generates m subsets as well, but it only generates subsets with cardinality less than or equal to $MC = \lceil n/k \rceil + 1$. Because LCS considers only low-cardinality subsets, for the same dataset the value of m can be much smaller than for CIW since there will be many fewer low-cardinality subsets with sums in the range $[\text{cost}(P^*), C^*]$. 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 for subsets S_2 through S_k, as described in Section 5.3.4, CIE trees are used to search for low-cardinality subsets with cardinality $\leq MC$ while IE binary tree search is used to search for high-cardinality subsets with cardinality $> MC$.

For our example with $n = 50$ and $k = 10$, CIW would first search for cardinality two through six subsets in CIE trees. The subsets are always considered in nondecreasing cardinality order. Consider the partial partition of d subsets with $d < k$, $P = \langle S_1, S_2, \ldots, S_d \rangle$. If there are no subsets remaining in the CIE trees that are mutually exclusive of $S_1 \cup S_2 \cup \cdots \cup S_d$, then CIW will attempt to complete the partition using IE to search for $k - d$ subsets with cardinalities greater than or equal to 7. Recall that 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 thus the search will be relatively inexpensive.

Phase 1 continues iterative weakening until the first complete partition is found. Since phase 1 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 CIW, the S_1 subset is always the subset with the greatest sum. If it turns out that the subset with the greatest sum in the optimal partition has a high cardinality, then phase 1 will not find an optimal partition.

Phase 2

After phase 1 is complete, phase 2 either proves that the partition found in phase 1 is optimal or it finds an optimal partition. Phase 2 performs a branch-and-bound recursive partitioning using the CIE trees created in phase 1. However, unlike phase 1, it does not force the S_1 subset using iterative weakening. Instead, the subsets S_1 through S_k with sums in the range $[lb, ub]$ are generated using CIE trees for low-cardinality subsets and IE binary tree search for high-cardinality subsets. The low-cardinality subsets are all generated in nondecreasing cardinality before any high-cardinality subsets are generated.

The first complete partition is not returned. Instead, the recursive partitioning continues until a partition is found with a cost equal to the lower bound on solution cost or until the search space is exhausted. Since no assumption is made about the cardinality of the subset with the largest sum, phase 2 guarantees an optimal partition.

Discussion

Phase 2 is typically more computationally expensive than phase 1. Since the iterative weakening happens only over low-cardinality subsets and the CIE trees contain only low-cardinality subsets, phase 1 is significantly faster than the iterative weakening of CIW.

Phase 1 accomplishes two things. First, it creates a better upper bound for phase 2. Since a complete partition is found in phase 1, the optimal partition is guaranteed to have a cost no greater than of the best partition found in phase 1. Second, it populates the CIE trees with all low-cardinality subsets with sums less than or equal to the upper bound. These CIE trees are therefore ready to use for phase 2.

6.5 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 owing to time or memory constraints are shown with a hyphen.

Figure 16 reports the average runtimes for LCS and CIW to partition the input sets into $k = 3$ to 10 subsets. For $k \leq 4$, CIW outperforms LCS and there is no clear trend for the ratio of the runtimes of the two algorithms as a function of n. For $k = 5$, CIW outperforms LCS for the values of n that we tested. However, as n increases, the ratio of the runtime of CIW to LCS also increases. The trend suggests that, with high enough n, LCS would 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 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 $n = 68$, in which CIW takes 24.7 times as long as LCS, on average.

6.6 Experimental Results: Memory Usage

Both CIW and LCS use memory to generate subsets and to cache them. Table 4 shows the average memory in gigabytes to partition n integers into $k = 3$ to 10 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 of CIW to LCS.

Since LCS stores only 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, SS requires the square root of the amount of memory that HS does. However, since LCS is generating only low-cardinality subsets, it uses less memory than standard EHS would.

For $k \leq 7$, CIW uses significantly less memory than LCS. 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.

6.7 Summary

This section has introduced 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 (Section 3) or binary search over bin-packing problems (Section 4). The previous algorithms begin by calculating an approximate partition to generate lower and upper bounds. They then shrink these bounds until the optimal partition is found. In contrast, CIW uses iterative weakening to search for the optimal partition. It starts by theorizing that a perfect partition is possible and sets the upper bound to $\text{cost}(P^*)$, 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

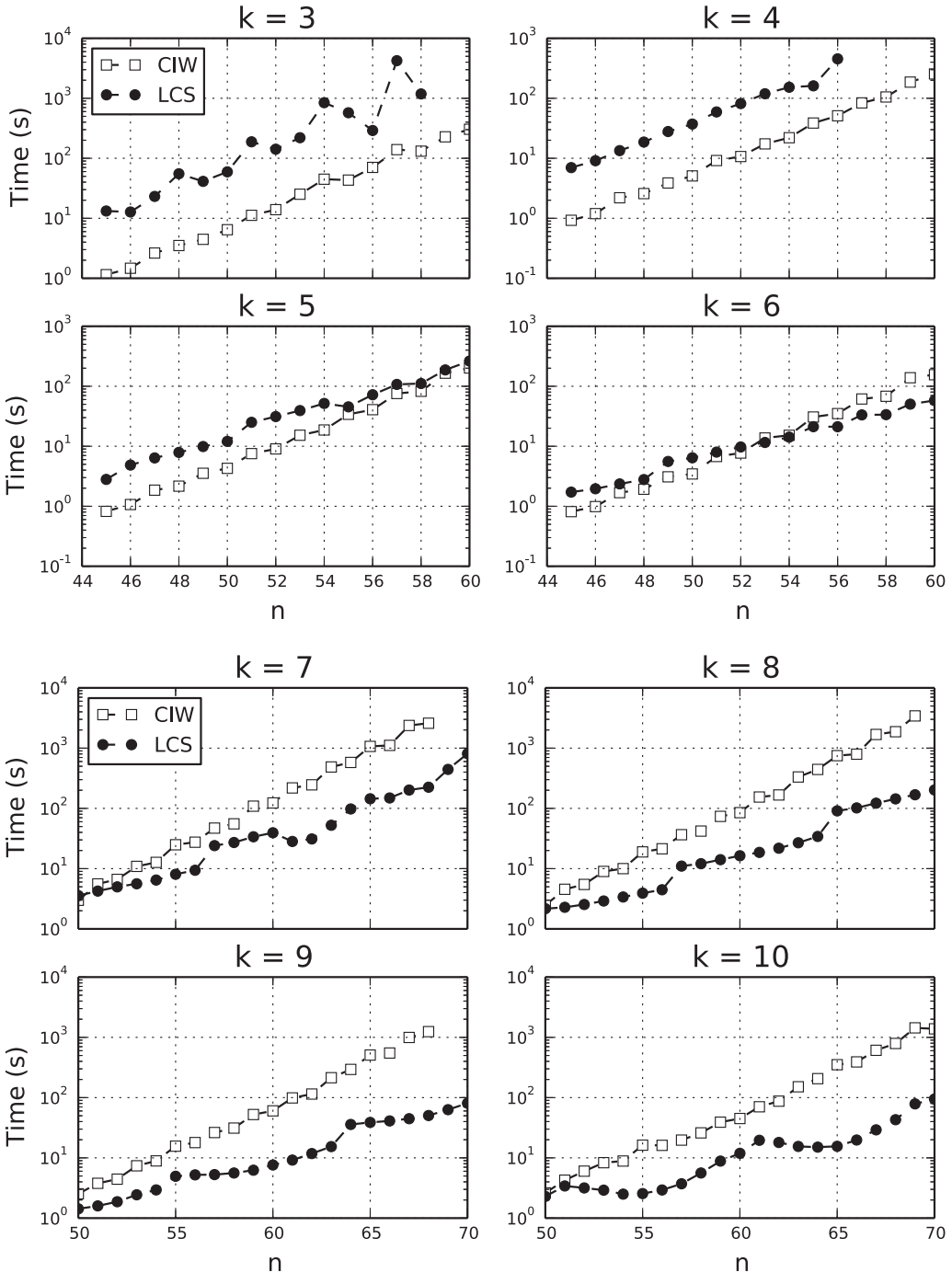


Fig. 16. The average time in seconds to optimally partition 48-bit integers 3 through 10 ways using LCS and CIW.

Table 4. Average Memory Use in GB to Optimally Partition 48-bit Integers 3 Through 10 Ways Using CIW and 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	.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. Listing of Multi-way Partitioning Algorithms Described in This Article

Algorithm	Abbr.	Year	Section	Citation
Complete greedy algorithm	CGA	1998	3.3	(Korf 1998)
Complete Karmarker Karp	CKK	2009	3.1.2	(Korf 2009)
Recursive number partitioning	RNP	2009	3.7	(Korf 2009)
Improved recursive number partitioning	IRNP	2011	3.7	(Korf 2011)
Sequential number partitioning with inclusion-exclusion	SNPIE	2013	3.5.1, 3.6	(Moffitt 2013)
Binary-search improved bin completion	BSIBC	2013	4	(Schreiber and Korf 2013)
Binary-search branch-and-cut-and-price	BSBCP	2013[3]	4	(Dell'Amico et al. 2008; Schreiber and Korf 2013)
Sequential Number Partitioning with extended Schroeppel and Shamir	SNPESS	2013	3.5.3, 3.6	(Korf et al. 2014)
Cached iterative weakening	CIW	2014	5	(Schreiber and Korf 2014)

algorithms 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 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 LCS, which combines the ideas of CIW and the 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.

7 HIGH-LEVEL EXPERIMENTAL SUMMARY

Throughout this article, we have described the following eight algorithms in Table 5 for optimal multi-way number partitioning. The year is when the algorithm was published. The section is where we discussed the algorithm in this article. The citation is for the article in which the algorithm was introduced. All of the articles were written by authors of this article except for Dell'Amico et al. (2008).

We have shown experimental data comparing the runtimes of these algorithms throughout this article. However, for specific values of n and k, these graphs compare only a few algorithms. In this section, we provide graphs comparing the runtimes of all eight algorithms.

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

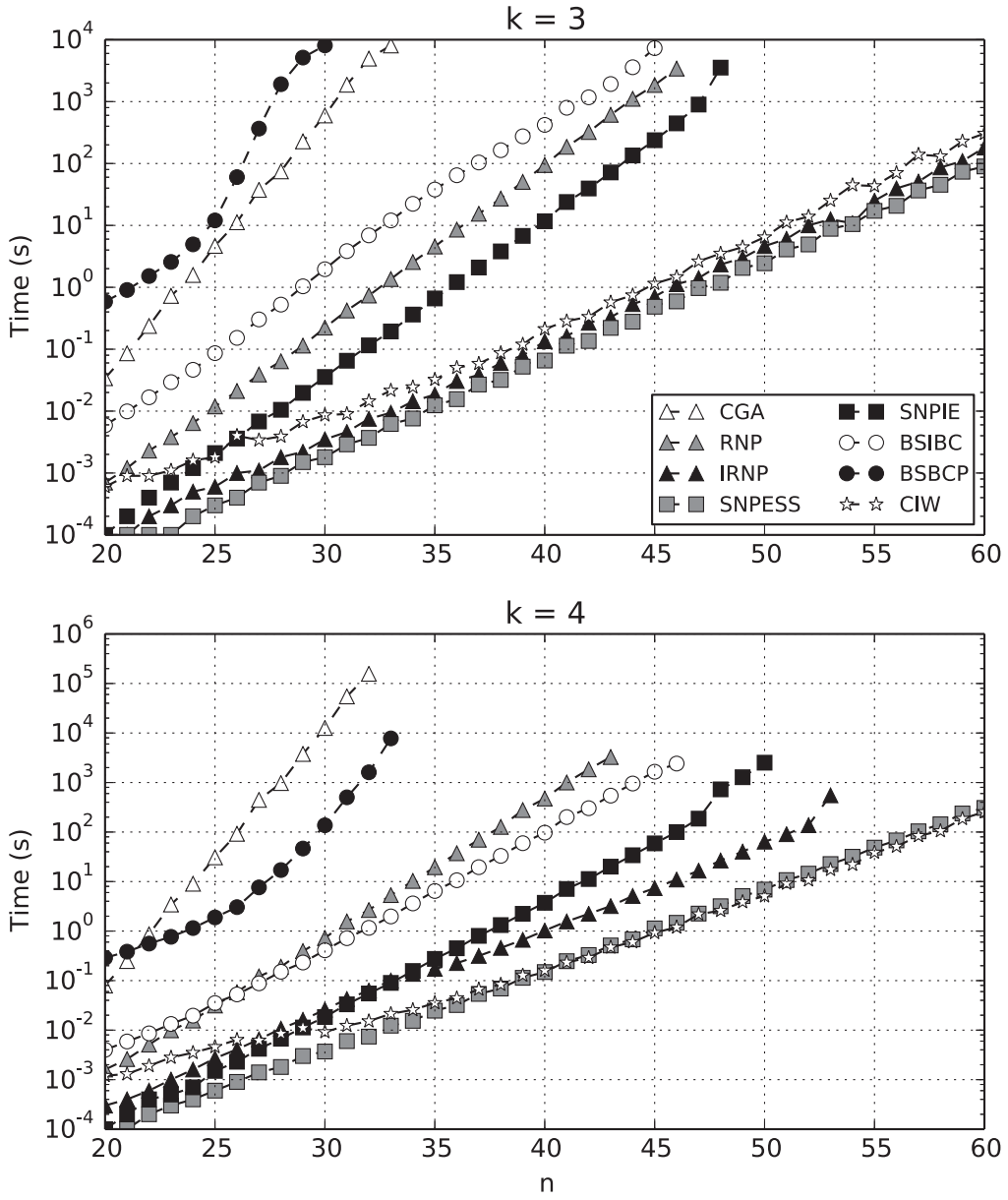


Fig. 17. The average runtime of seven algorithms for three- and four-way partitioning.

The problem instances are the same that we have presented throughout this article. The integers are sampled uniformly at random from the range $[1, 2^{48} - 1]$. We generated 100 problem instances for each n from $n = 20$ to 60. Figure 17 shows results for $k = 3$ and $k = 4$, Figure 18 for $k = 5$ and $k = 6$, Figure 19 for $k = 7$ and $k = 8$, Figure 20 for $k = 9$ and $k = 10$, and Figure 21 for $k = 11$ and $k = 12$. All algorithms were run on the same problem instances on an Intel Xeon X5680 CPU at 3.33GHz.

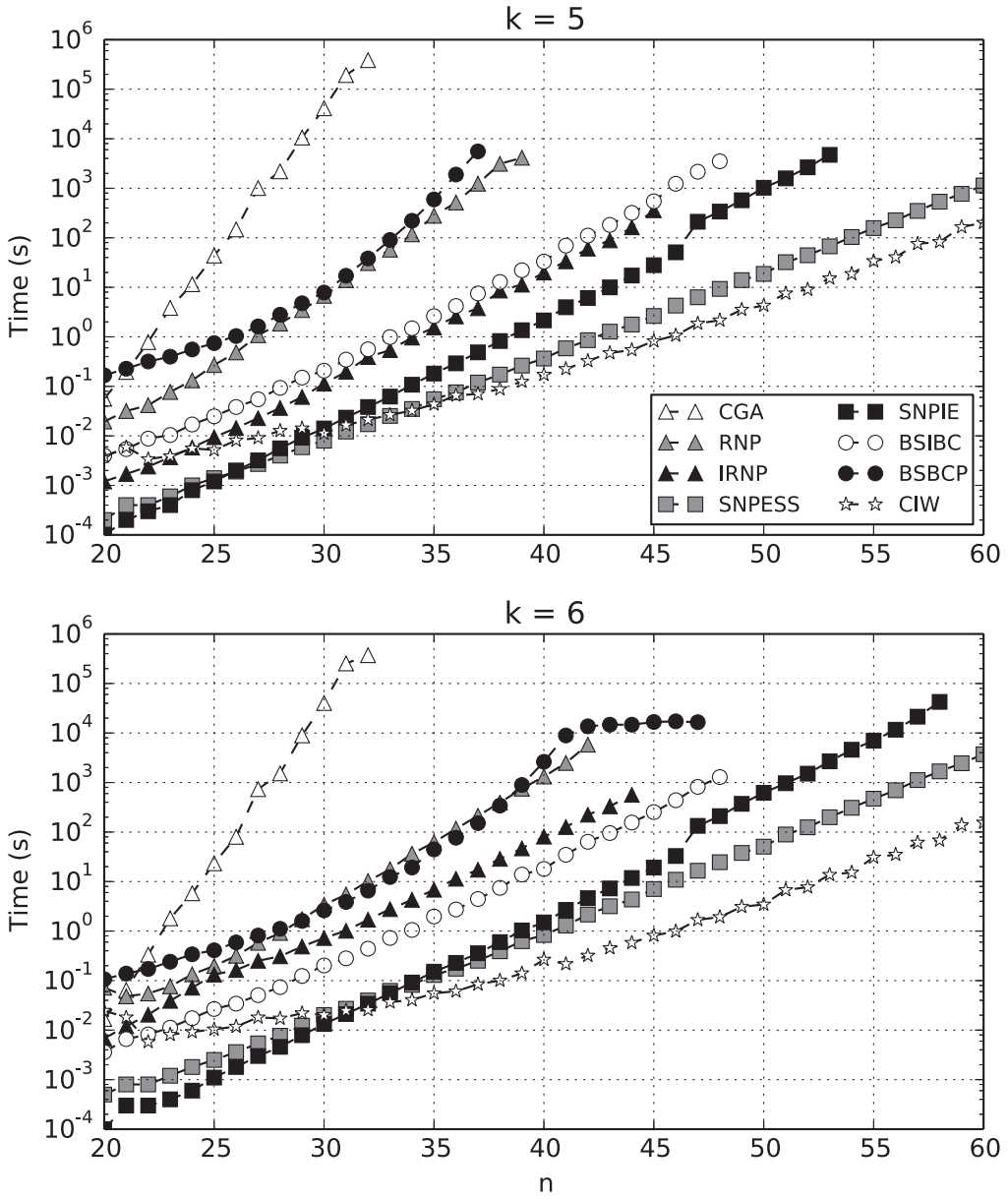


Fig. 18. The average runtime of seven algorithms for five- and six-way partitioning.

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 Korf (2009). The RNP dataset was sampled uniformly at random from the range $[1, 2^{31} - 1]$. The algorithm was run on an IBM Intellistation with a 2GHz AMD Opteron processor. This machine runs at 2GHz, compared to 3.3GHz for the X5680, but since RNP is always multiple orders of magnitude slower than the state of the art, this does not make a significant difference.

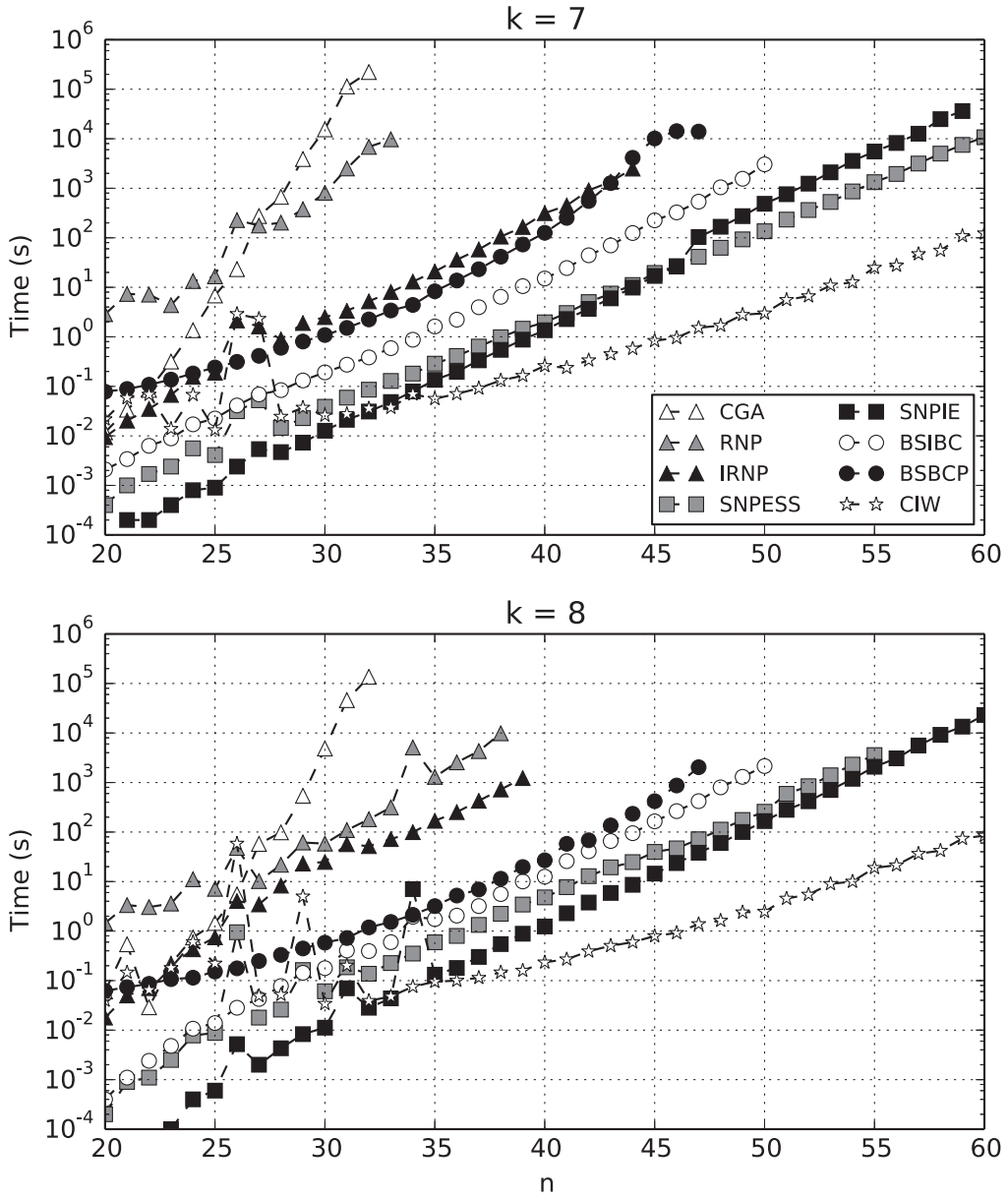


Fig. 19. The average runtime of seven algorithms for seven- and eight-way partitioning.

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

The improvement over CGA from 1998 is even more impressive. Since multi-way experiments beyond $k = 3$ were not run in Korf (1998), we ran the CGA against our benchmark sets. For $n = 32$, the largest problem instances for which the CGA was able to complete our benchmarks, CIW

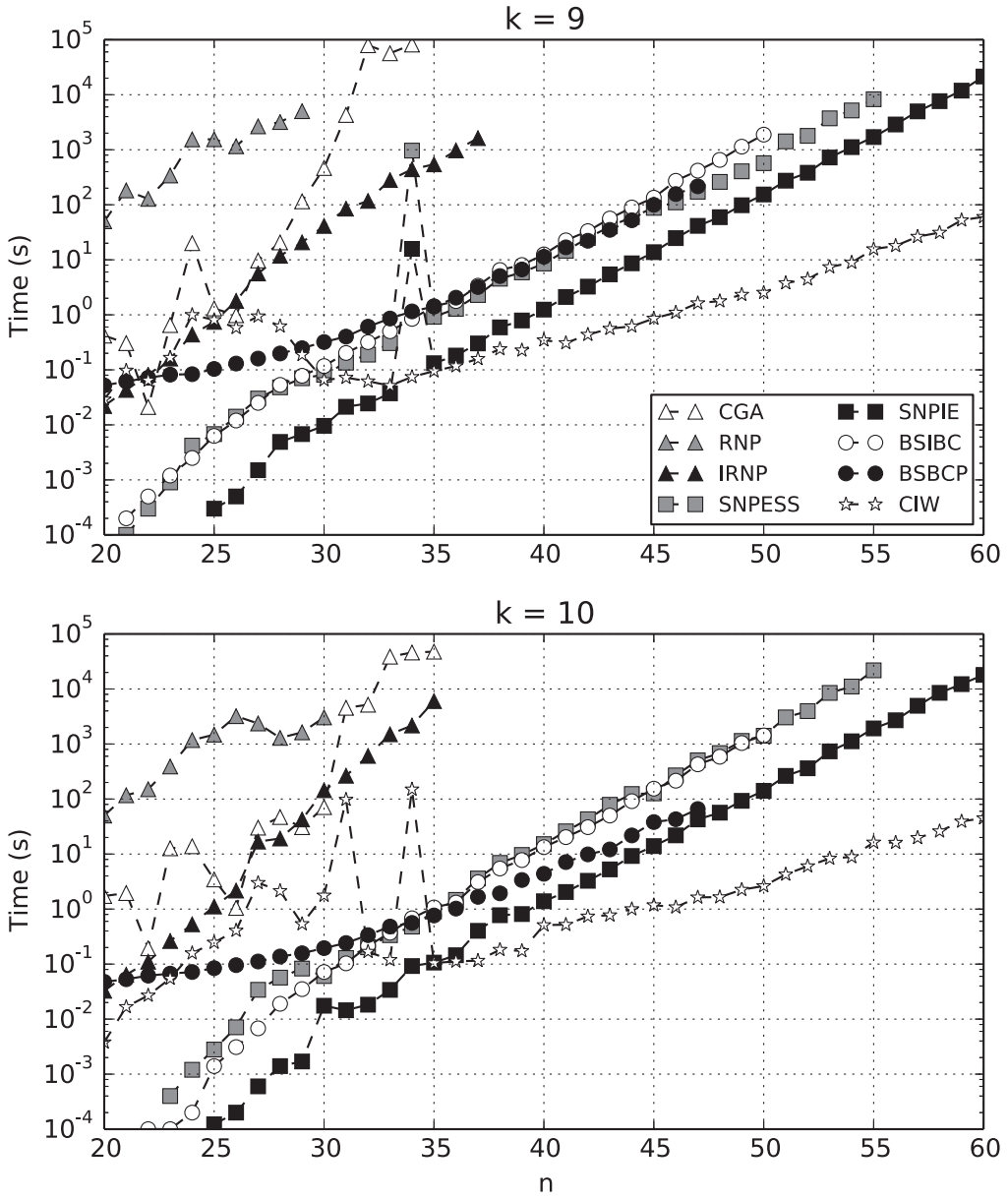


Fig. 20. The average runtime of seven algorithms for nine- and ten-way partitioning.

was between five and seven orders of magnitude faster, that is, up to ten million times faster! This speaks to how far we have come between 1998 and today in solving multi-way number-partitioning problems.

SNPESS is the dominant algorithm for $k = 3$ and for small values of n for $k = 4$ and $k = 5$. CIW dominates for large values of n for $k = 4$ through $k = 12$. SNPIE is the best for $k = 8$ through $k = 10$ for very small values of n. BSIBC and BSBCP are best for $k = 11$ and $k = 12$ and small values of n.

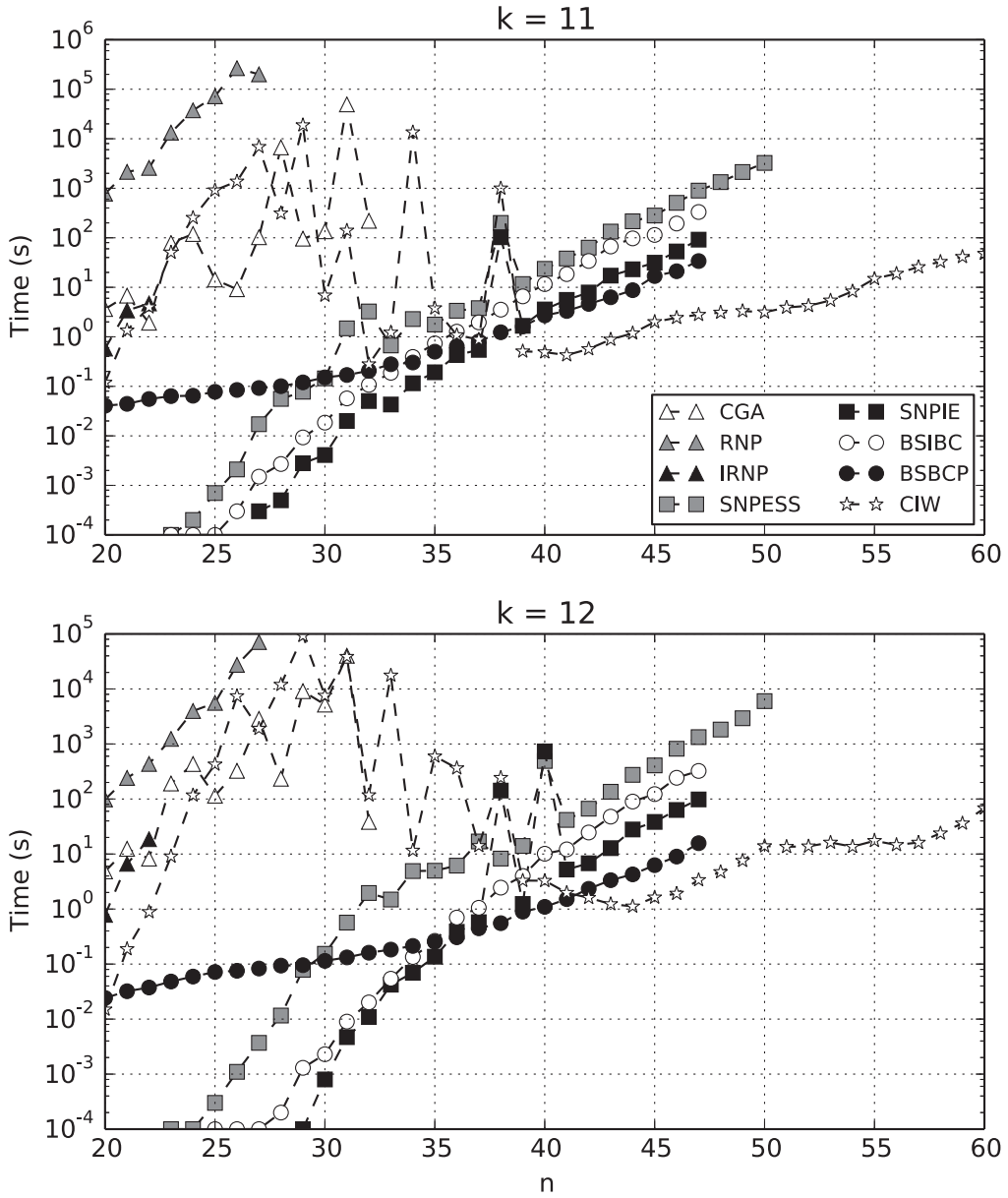


Fig. 21. The average run-time of seven algorithms for 11- and 12-way partitioning.

The state-of-the-art algorithm for multi-way number partitioning depends on the size of the input set n, the number of subsets in a partition k and the magnitude of the input numbers. Table 6 shows the algorithm that had the best average runtime for each value of n and k for multi-way partitioning of 48-bit integers.

In general, the magnitude of the input integers affects the runtime only by determining if there will be perfect partitions. We chose 48-bit integers because it eliminates perfect partitions for the values of n and k that we ran, thus making the problem instances harder. As long as there are no

Table 6. Algorithm with the Fastest Average Runtime to Partition 48-bit Integers for $30 \leq n \leq 60$ and $3 \leq k \leq 12$

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

CIW	- Cached Iterative Weakening	IBC	- Binary-Search Improved Bin Completion
SNP	- Sequential Number Partitioning	BCP	- Binary-Search Branch-and-Cut-and-Price (previous work)
MOF	- Moffitt Partitioning		

perfect partitions, the magnitude of the integers will not significantly affect the runtimes of the algorithms.

8 CONTRIBUTIONS, FUTURE WORK, AND CONCLUSIONS

8.1 Summary of Contributions

This article has covered algorithms for computing optimal solutions to both the two-way and multi-way number-partitioning problems. The two-way number-partitioning problem is one of Richard Karp's 21 original NP-complete problems (Karp 1972) and one of Garey and Johnson's six

fundamental NP-complete problems (Garey and Johnson 1979). Multi-way number partitioning is theoretically interesting both because it is NP-complete and 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 1960s that have continuously improved the runtimes to optimally partition numbers. We have covered both existing algorithms and presented our new algorithms for optimal number partitioning broken up into three classes.

Section 3 presented our two optimal branch-and-bound algorithms, SNPIE and SNPESS, which recursively partition the input set. SNPESS is state of the art for small k.

Section 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. We showed that any bin-packing solver can be used with the MULTIFIT algorithm to solve number-partitioning problems using bin-packing algorithms. While MULTIFIT is not our algorithm, using improved bin completion with MULTIFIT is our algorithm, which is state of the art for large k and small n.

Section 5 introduced cached iterative weakening (CIW) (Schreiber and Korf 2014). This algorithm hypothesizes that a perfect partition is possible and sets the bounds accordingly. It then iteratively weakens these bounds until it finds a feasible solution. This first solution found is guaranteed to be optimal. CIW outperforms both the branch-and-bound and MULTIFIT algorithms for most values of n and k that we tested and is state of the art for $4 \leq k \leq 12$ and $n \geq 41$.

RNP, SNPIE, and SNPESS 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.

In terms of memory usage, SNPIE, BSIBC, and BSBCP all use memory linear in n, the size of the input set. SNPESS and CIW use the Schroeppel and Shamir algorithm and, therefore, use $O(2^{n/4})$ memory.

8.2 Future Work

The algorithms in this article have focused on number-partitioning instances with very-large-magnitude input-positive integers. Large-magnitude integers eliminate perfect partitions. As a result, the algorithms cannot return before proving that the best solution found so far is optimal. In contrast, with small-magnitude input integers, there tends to be a high density of perfect partitions, which when found can be returned immediately. When perfect partitions exist, the goal becomes finding these partitions as quickly as possible. When the magnitude of the input integers is low, the values of n and k that become tractable change. Future work involves working with moderate-magnitude inputs and tailoring algorithms to solve these problems.

Both iterative weakening and caching IE 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 (Kellerer et al. 2004), job shop scheduling (Coffman Jr. and Bruno 1976), or packing problems in general, including rectangle packing and higher-dimensional packing (Fukunaga and Korf 2007; Huang and Korf 2009).

For many of the algorithms in this article. such as SNPIE and SNPESS, 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 cost(P^*). Any improvement to algorithms for solving this problem would improve the performance of many algorithms in this article.

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 that does not require a parameter and can efficiently generate subsets in sum order starting with a lower bound could greatly improve CIW.

8.3 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 1 minute. Yet, given the apparent simplicity, number partitioning is surprisingly complicated. Depending on the cardinality of the input integers, their magnitude, and the number of subsets to partition into, 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. 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.

APPENDIX

The appendix shows the same experimental results used throughout the article. Here, we present the data as tables so that the results can be seen more precisely.

Again, the datasets used are composed of problem instances with integers sampled uniformly at random from the range $[1, 2^{48} - 1]$. There are 100 problem instances for each value of n. We show results for n ranging from $n = 40$ to 60. All experiments were run on an Intel Xeon X5680 CPU at 3.33GHz.

Table 7. Average Time in Seconds to Partition $n = 40$ to 60 Integers Three Ways

$k \rightarrow$	3-Way							
$n \downarrow$	CGA	RNP	IRNP	SNPIE	SNPESS	BSIBC	BSBCP	CIW
20	.03	.00	.00	.00	.00	.01	.59	.00
21	.09	.00	.00	.00	.00	.01	.91	.00
22	.24	.00	.00	.00	.00	.02	1.52	.00
23	.73	.00	.00	.00	.00	.03	2.57	.00
24	1.58	.01	.00	.00	.00	.05	4.94	.00
25	4.67	.01	.00	.00	.00	.09	12.0	.00
26	11.2	.02	.00	.00	.00	.15	60.3	.00
27	37.5	.04	.00	.01	.00	.30	366	.00
28	74.6	.06	.00	.01	.00	.53	1910	.00
29	227	.11	.00	.02	.00	1.04	5141	.01
30	597	.22	.00	.04	.00	1.96	8088	.01
31	1871	.42	.00	.07	.00	3.84	-	.01
32	4946	.74	.01	.12	.00	6.90	-	.01
33	8062	1.36	.01	.19	.01	12.0	-	.02
34	-	2.56	.01	.36	.01	22.3	-	.02
35	-	4.56	.02	.66	.01	38.0	-	.03
36	-	8.54	.03	1.21	.02	64.6	-	.05
37	-	15.4	.04	2.09	.03	104	-	.06
38	-	27.1	.06	3.80	.03	164	-	.09
39	-	50.4	.08	6.77	.05	275	-	.12
40	-	94.3	.13	11.7	.07	421	-	.21
41	-	186	.16	23.9	.11	796	-	.28
42	-	322	.27	39.5	.14	1173	-	.33
43	-	611	.33	71.8	.22	1922	-	.57
44	-	1108	.54	135	.28	3589	-	.74
45	-	1854	.71	238	.49	7308	-	1.15
46	-	3410	1.12	445	.59	-	-	1.47
47	-	-	1.41	896	.97	-	-	2.63
48	-	-	2.34	3538	1.18	-	-	3.51
49	-	-	2.97	-	2.06	-	-	4.46
50	-	-	4.63	-	2.44	-	-	6.44
51	-	-	6.05	-	4.10	-	-	11.2
52	-	-	10.0	-	4.93	-	-	13.9
53	-	-	12.5	-	8.81	-	-	25.2
54	-	-	10.8	-	10.5	-	-	44.8
55	-	-	25.1	-	17.2	-	-	43.3
56	-	-	39.6	-	20.7	-	-	70.5
57	-	-	51.0	-	36.7	-	-	139
58	-	-	86.3	-	45.4	-	-	131
59	-	-	110	-	73.6	-	-	228
60	-	-	181	-	89.6	-	-	304

Table 8. Average Time in Seconds to Partition $n = 40$ to 60 Integers Four Ways

$k \rightarrow$	4-Way							
$n \downarrow$	CGA	RNP	IRNP	SNPIE	SNPESS	BSIBC	BSBCP	CIW
20	.08	.00	.00	.00	.00	.00	.29	.00
21	.25	.00	.00	.00	.00	.01	.38	.00
22	.86	.01	.00	.00	.00	.01	.56	.00
23	3.44	.01	.00	.00	.00	.01	.77	.00
24	9.03	.02	.00	.00	.00	.02	1.15	.00
25	30.8	.03	.00	.00	.00	.04	1.89	.00
26	92.0	.06	.00	.00	.00	.05	3.04	.01
27	446	.12	.01	.00	.00	.09	7.62	.01
28	974	.19	.01	.01	.00	.15	17.1	.01
29	3771	.39	.02	.01	.00	.23	46.4	.01
30	12627	.72	.03	.02	.00	.41	137	.01
31	54739	1.56	.04	.03	.01	.72	502	.01
32	156905	2.66	.06	.05	.01	1.16	1609	.02
33	-	5.33	.10	.09	.01	1.98	7781	.02
34	-	10.3	.14	.16	.02	3.62	-	.03
35	-	19.5	.17	.27	.02	6.45	-	.04
36	-	37.3	.23	.45	.03	10.6	-	.04
37	-	70.0	.32	.80	.05	19.5	-	.07
38	-	127	.46	1.32	.07	33.0	-	.08
39	-	278	.67	2.23	.11	59.5	-	.13
40	-	474	1.03	3.72	.15	97.1	-	.16
41	-	1001	1.56	7.09	.25	200	-	.23
42	-	1845	2.23	11.2	.33	303	-	.29
43	-	3266	3.20	20.0	.51	544	-	.47
44	-	-	5.11	34.0	.68	955	-	.60
45	-	-	7.37	59.0	1.13	1651	-	.92
46	-	-	11.0	99.8	1.47	2414	-	1.20
47	-	-	16.6	187	2.26	-	-	2.20
48	-	-	26.5	728	3.18	-	-	2.58
49	-	-	40.2	1280	5.09	-	-	3.87
50	-	-	63.9	2516	6.89	-	-	5.08
51	-	-	90.5	-	10.7	-	-	9.17
52	-	-	137	-	14.4	-	-	10.7
53	-	-	551	-	22.6	-	-	17.4
54	-	-	-	-	31.7	-	-	22.0
55	-	-	-	-	48.4	-	-	38.7
56	-	-	-	-	67.8	-	-	51.1
57	-	-	-	-	103	-	-	83.8
58	-	-	-	-	143	-	-	105
59	-	-	-	-	237	-	-	186
60	-	-	-	-	307	-	-	254

Table 9. Average Time in Seconds to Partition $n = 40$ to 60 Integers Five Ways

$k \rightarrow$	5-Way							
$n \downarrow$	CGA	RNP	IRNP	SNPIE	SNPESS	BSIBC	BSBCP	CIW
20	.06	.02	.00	.00	.00	.00	.17	.00
21	.20	.03	.00	.00	.00	.01	.23	.01
22	.78	.04	.00	.00	.00	.01	.32	.00
23	3.84	.08	.00	.00	.00	.01	.40	.00
24	11.5	.13	.01	.00	.00	.02	.56	.01
25	43.8	.27	.01	.00	.00	.02	.75	.01
26	147	.48	.01	.00	.00	.04	1.04	.01
27	1011	1.08	.02	.00	.00	.05	1.62	.01
28	2194	1.85	.04	.01	.00	.09	2.79	.01
29	10558	3.47	.06	.01	.01	.15	4.76	.01
30	41157	6.72	.11	.01	.01	.21	7.85	.01
31	193051	13.9	.20	.02	.01	.35	17.1	.02
32	386011	30.8	.39	.04	.02	.56	38.3	.02
33	-	57.6	.54	.06	.03	.99	90.4	.03
34	-	117	.96	.11	.03	1.49	221	.03
35	-	278	1.55	.18	.05	2.62	595	.04
36	-	516	2.58	.29	.08	4.19	1891	.07
37	-	1235	3.78	.49	.12	7.51	5553	.07
38	-	3114	8.59	.82	.17	12.8	-	.09
39	-	4134	11.3	1.36	.26	22.1	-	.12
40	-	-	19.9	2.15	.36	33.1	-	.17
41	-	-	33.4	3.96	.59	69.5	-	.23
42	-	-	60.4	6.08	.86	111	-	.33
43	-	-	88.8	9.97	1.27	182	-	.47
44	-	-	163	17.3	1.77	319	-	.54
45	-	-	359	27.9	2.64	546	-	.82
46	-	-	-	50.9	4.27	1233	-	1.07
47	-	-	-	211	6.30	2159	-	1.85
48	-	-	-	339	9.30	3504	-	2.16
49	-	-	-	572	14.0	-	-	3.54
50	-	-	-	1023	18.6	-	-	4.28
51	-	-	-	1585	32.2	-	-	7.54
52	-	-	-	2647	44.4	-	-	9.02
53	-	-	-	4727	67.1	-	-	15.2
54	-	-	-	-	104	-	-	18.7
55	-	-	-	-	157	-	-	34.1
56	-	-	-	-	225	-	-	40.7
57	-	-	-	-	349	-	-	75.7
58	-	-	-	-	536	-	-	82.5
59	-	-	-	-	771	-	-	165
60	-	-	-	-	1140	-	-	201

Table 10. Average Time In Seconds to Partition $n = 40$ to 60 Integers Six Ways

$k \rightarrow$	6-Way							
$n \downarrow$	CGA	RNP	IRNP	SNPIE	SNPESS	BSIBC	BSBCP	CIW
20	.02	.07	.01	.00	.00	.00	.11	.02
21	.06	.05	.01	.00	.00	.01	.14	.02
22	.33	.05	.02	.00	.00	.01	.17	.01
23	1.81	.08	.04	.00	.00	.01	.24	.01
24	5.79	.13	.07	.00	.00	.02	.34	.01
25	23.4	.20	.13	.00	.00	.03	.41	.01
26	79.9	.32	.16	.00	.00	.03	.59	.01
27	734	.58	.25	.00	.01	.05	.81	.02
28	1529	.92	.31	.00	.01	.07	1.11	.02
29	8943	1.74	.49	.01	.01	.12	1.60	.02
30	40323	3.39	.72	.01	.02	.20	2.62	.02
31	253915	5.46	1.01	.02	.03	.28	3.87	.02
32	372155	10.2	1.70	.03	.04	.44	6.55	.03
33	-	17.7	2.78	.06	.06	.73	12.4	.04
34	-	36.3	4.28	.09	.08	1.05	19.2	.04
35	-	62.0	6.84	.15	.13	1.97	44.5	.06
36	-	116	11.4	.23	.17	2.73	76.8	.06
37	-	212	17.3	.36	.25	4.44	152	.08
38	-	392	28.8	.60	.39	7.45	342	.10
39	-	750	46.8	1.03	.62	13.8	891	.14
40	-	1321	80.8	1.50	.83	18.0	2623	.27
41	-	2480	126	2.68	1.29	34.7	8922	.22
42	-	5815	224	4.63	2.16	63.6	13632	.32
43	-	-	331	7.26	3.17	95.4	14701	.46
44	-	-	565	11.8	4.35	156	14717	.58
45	-	-	-	19.2	7.07	253	16689	.81
46	-	-	-	32.8	10.9	439	17194	.99
47	-	-	-	133	16.6	817	16518	1.69
48	-	-	-	210	24.6	1287	-	1.92
49	-	-	-	372	37.9	-	-	3.09
50	-	-	-	620	50.7	-	-	3.45
51	-	-	-	968	89.2	-	-	6.75
52	-	-	-	1509	125	-	-	7.70
53	-	-	-	2679	199	-	-	13.7
54	-	-	-	4597	309	-	-	15.2
55	-	-	-	6999	472	-	-	30.7
56	-	-	-	11639	698	-	-	35.0
57	-	-	-	21347	1119	-	-	61.4
58	-	-	-	42433	1677	-	-	68.1
59	-	-	-	-	2448	-	-	139
60	-	-	-	-	3728	-	-	156

Table 11. Average Time in Seconds to Partition $n = 40$ to 60 Integers Seven Ways

$k \rightarrow$	7-Way							
$n \downarrow$	CGA	RNP	IRNP	SNPIE	SNPESS	BSIBC	BSBCP	CIW
20	.01	2.87	.01	.00	.00	.00	.08	.02
21	.03	7.33	.02	.00	.00	.00	.09	.06
22	.08	7.11	.03	.00	.00	.01	.11	.07
23	.32	4.36	.07	.00	.00	.01	.14	.01
24	1.36	13.3	.16	.00	.01	.02	.18	.07
25	6.84	16.6	.19	.00	.00	.02	.24	.01
26	23.2	227	2.14	.00	.03	.04	.31	2.90
27	274	177	1.62	.01	.05	.07	.41	2.31
28	670	202	.89	.00	.01	.08	.60	.02
29	3884	376	1.90	.01	.02	.13	.80	.04
30	15457	810	2.50	.01	.04	.19	1.09	.03
31	113607	2510	3.35	.02	.06	.27	1.51	.03
32	222113	6842	5.21	.03	.09	.38	2.25	.04
33	-	9720	8.04	.05	.13	.59	3.41	.04
34	-	-	13.0	.08	.18	.87	4.40	.07
35	-	-	20.8	.14	.29	1.62	8.31	.06
36	-	-	36.0	.20	.41	2.22	13.7	.07
37	-	-	58.0	.34	.64	3.96	23.2	.09
38	-	-	106	.55	.98	6.43	41.5	.13
39	-	-	165	.88	1.46	10.6	73.1	.16
40	-	-	314	1.36	1.96	15.1	126	.26
41	-	-	448	2.28	3.02	24.5	252	.24
42	-	-	908	3.67	5.03	44.3	552	.34
43	-	-	1351	5.99	7.47	70.2	1269	.45
44	-	-	2492	9.86	11.1	125	4123	.58
45	-	-	-	17.0	19.5	225	10134	.83
46	-	-	-	26.3	26.7	325	14241	.96
47	-	-	-	104	41.4	537	13972	1.52
48	-	-	-	167	62.9	1045	-	1.69
49	-	-	-	274	92.9	1555	-	2.79
50	-	-	-	486	135	3042	-	2.99
51	-	-	-	758	233	-	-	5.57
52	-	-	-	1239	364	-	-	6.63
53	-	-	-	2098	529	-	-	10.9
54	-	-	-	3566	864	-	-	12.7
55	-	-	-	5524	1328	-	-	24.9
56	-	-	-	8213	1945	-	-	27.5
57	-	-	-	12634	3158	-	-	47.1
58	-	-	-	24877	5015	-	-	55.4
59	-	-	-	36207	7520	-	-	109
60	-	-	-	-	10715	-	-	123

Table 12. Average Time in Seconds to Partition $n = 40$ to 60 Integers Eight Ways

$k \rightarrow$	8-Way							
$n \downarrow$	CGA	RNP	IRNP	SNPIE	SNPESS	BSIBC	BSBCP	CIW
20	.06	1.45	.02	.00	.00	.00	.06	.04
21	.54	3.35	.05	.00	.00	.00	.07	.15
22	.03	3.05	.07	.00	.00	.00	.09	.06
23	.22	3.59	.20	.00	.00	.00	.11	.18
24	.74	11.1	.43	.00	.01	.01	.11	.60
25	1.45	7.13	.75	.00	.01	.01	.15	.22
26	5.51	47.7	4.04	.01	.95	.03	.18	58.6
27	57.5	10.2	3.45	.00	.02	.04	.25	.05
28	98.5	21.8	8.27	.00	.03	.08	.33	.05
29	540	61.4	23.0	.01	.16	.14	.45	5.01
30	4850	58.4	24.9	.01	.06	.18	.58	.03
31	46021	109	56.7	.07	.19	.41	.72	.20
32	136426	181	51.9	.03	.14	.39	1.19	.04
33	-	314	72.0	.04	.23	.60	1.53	.05
34	-	5097	99.4	7.04	.35	1.95	2.13	.08
35	-	1300	169	.13	.59	1.74	3.19	.10
36	-	2554	253	.18	.80	2.05	5.21	.10
37	-	4301	428	.30	1.33	3.20	6.92	.11
38	-	9850	723	.55	2.24	5.60	11.4	.15
39	-	-	1228	.89	3.44	10.0	19.7	.16
40	-	-	-	1.24	4.79	12.6	26.9	.23
41	-	-	-	2.29	7.70	25.8	57.9	.27
42	-	-	-	3.77	12.9	40.9	68.2	.39
43	-	-	-	5.89	19.4	65.9	136	.51
44	-	-	-	8.60	24.7	94.8	234	.60
45	-	-	-	14.5	39.7	165	420	.79
46	-	-	-	23.6	46.8	264	871	.92
47	-	-	-	38.2	71.5	421	2037	1.39
48	-	-	-	60.0	112	798	-	1.62
49	-	-	-	100	175	1302	-	2.40
50	-	-	-	166	257	2149	-	2.47
51	-	-	-	281	592	-	-	4.54
52	-	-	-	418	841	-	-	5.47
53	-	-	-	707	1398	-	-	8.98
54	-	-	-	1189	2297	-	-	10.0
55	-	-	-	2082	3584	-	-	19.0
56	-	-	-	3078	-	-	-	21.4
57	-	-	-	5589	-	-	-	36.6
58	-	-	-	9182	-	-	-	42.0
59	-	-	-	13441	-	-	-	73.6
60	-	-	-	23085	-	-	-	84.9

Table 13. Average Time in Seconds to Partition $n = 40$ to 60 Integers Nine Ways

$k \rightarrow$	9-Way							
$n \downarrow$	CGA	RNP	IRNP	SNPIE	SNPESS	BSIBC	BSBCP	CIW
20	.42	50.1	.02	.00	.00	.00	.05	.03
21	.31	182	.04	.00	.00	.00	.06	.10
22	.02	128	.08	.00	.00	.00	.07	.06
23	.65	341	.16	.00	.00	.00	.08	.17
24	20.1	1543	.44	.00	.00	.00	.08	1.00
25	1.30	1550	.74	.00	.01	.01	.10	.81
26	.98	1156	1.79	.00	.01	.01	.13	.58
27	9.71	2663	5.64	.00	.03	.03	.16	.94
28	20.7	3187	11.8	.00	.05	.05	.20	.62
29	115	5021	20.7	.01	.07	.08	.25	.19
30	465	-	41.3	.01	.08	.12	.32	.07
31	4284	-	85.0	.02	.13	.20	.40	.07
32	79225	-	118	.02	.19	.32	.61	.06
33	56622	-	281	.04	.31	.50	.86	.05
34	81550	-	442	15.8	965	.84	1.15	.07
35	-	-	543	.13	.92	1.43	1.41	.09
36	-	-	973	.18	1.29	1.77	2.05	.12
37	-	-	1626	.30	2.29	3.38	3.20	.16
38	-	-	-	.59	4.53	6.52	5.07	.24
39	-	-	-	.79	5.91	8.04	6.68	.23
40	-	-	-	1.24	8.55	12.5	11.4	.35
41	-	-	-	2.11	14.6	22.6	16.8	.31
42	-	-	-	3.26	24.9	33.2	22.2	.44
43	-	-	-	5.44	40.4	56.3	35.0	.56
44	-	-	-	8.63	59.6	89.3	52.3	.62
45	-	-	-	13.8	87.9	135	99.7	.86
46	-	-	-	24.8	111	273	156	1.10
47	-	-	-	41.4	173	417	216	1.63
48	-	-	-	59.4	260	654	-	1.78
49	-	-	-	98.0	405	1139	-	2.32
50	-	-	-	154	570	1883	-	2.53
51	-	-	-	274	1420	-	-	3.78
52	-	-	-	382	1790	-	-	4.43
53	-	-	-	724	3723	-	-	7.39
54	-	-	-	1116	5179	-	-	8.90
55	-	-	-	1701	8280	-	-	15.6
56	-	-	-	2878	-	-	-	17.9
57	-	-	-	4983	-	-	-	26.3
58	-	-	-	7636	-	-	-	31.2
59	-	-	-	11874	-	-	-	52.4
60	-	-	-	21414	-	-	-	60.3

Table 14. Average Time in Seconds to Partition $n = 40$ to 60 Integers Ten Ways

$k \rightarrow$	10-Way							
$n \downarrow$	CGA	RNP	IRNP	SNPIE	SNPESS	BSIBC	BSBCP	CIW
20	1.73	50.1	.03	.00	.00	.00	.05	.00
21	1.95	117	.06	.00	.00	.00	.05	.02
22	.19	151	.11	.00	.00	.00	.06	.03
23	12.7	390	.26	.00	.00	.00	.07	.05
24	13.8	1177	.53	.00	.00	.00	.07	.16
25	3.45	1478	1.11	.00	.00	.00	.08	.25
26	1.05	3205	2.17	.00	.01	.00	.10	.41
27	30.2	2363	16.8	.00	.03	.01	.11	3.00
28	47.0	1291	19.2	.00	.06	.02	.14	2.14
29	30.8	1628	42.5	.00	.08	.04	.16	.54
30	71.3	3043	144	.02	.06	.07	.20	1.77
31	4545	-	265	.01	.13	.10	.24	97.6
32	5174	-	607	.02	.20	.21	.34	.17
33	38654	-	1501	.03	.34	.37	.48	.12
34	45955	-	2169	.09	.48	.68	.56	150
35	47638	-	6027	.11	.90	1.07	.77	.10
36	-	-	-	.15	1.47	1.31	1.02	.11
37	-	-	-	.40	3.60	3.09	1.67	.12
38	-	-	-	.77	6.91	5.46	1.93	.18
39	-	-	-	.81	9.63	7.77	3.38	.17
40	-	-	-	1.39	15.2	13.3	4.37	.51
41	-	-	-	2.04	25.9	20.3	7.14	.52
42	-	-	-	3.27	42.9	30.8	9.87	.74
43	-	-	-	5.26	78.6	50.1	12.1	.77
44	-	-	-	9.22	123	91.4	21.8	1.00
45	-	-	-	14.0	127	154	38.2	1.18
46	-	-	-	21.9	269	215	42.9	1.08
47	-	-	-	42.6	505	427	65.8	1.63
48	-	-	-	56.5	684	587	-	1.65
49	-	-	-	92.8	1137	1041	-	2.25
50	-	-	-	141	1413	1433	-	2.62
51	-	-	-	263	3043	-	-	4.26
52	-	-	-	362	3961	-	-	6.02
53	-	-	-	735	8486	-	-	8.31
54	-	-	-	1120	11085	-	-	8.86
55	-	-	-	1920	21829	-	-	16.3
56	-	-	-	2700	-	-	-	16.1
57	-	-	-	4950	-	-	-	19.7
58	-	-	-	8508	-	-	-	26.0
59	-	-	-	12249	-	-	-	39.1
60	-	-	-	18036	-	-	-	45.0

Table 15. Average Time in Seconds to Partition $n = 40$ to 60 Integers 11 Ways

$k \rightarrow$	11-Way							
$n \downarrow$	CGA	RNP	IRNP	SNPIE	SNPESS	BSIBC	BSBCP	CIW
20	3.65	777	.58	.00	.00	.00	.04	.12
21	6.78	2142	3.40	.00	.00	.00	.04	1.35
22	1.92	2577	4.72	.00	.00	.00	.06	3.98
23	78.5	13252	-	.00	.00	.00	.06	50.6
24	119	37467	-	.00	.00	.00	.06	254
25	14.3	71827	-	.00	.00	.00	.08	913
26	9.13	265922	-	.00	.00	.00	.08	1375
27	103	199694	-	.00	.02	.00	.09	6868
28	6676	-	-	.00	.06	.00	.10	312
29	95.9	-	-	.00	.08	.01	.12	18839
30	135	-	-	.00	.14	.02	.15	6.87
31	49343	-	-	.02	1.48	.06	.17	140
32	220	-	-	.05	3.25	.11	.21	.28
33	-	-	-	.04	.67	.19	.28	1.22
34	-	-	-	.12	2.28	.39	.30	13555
35	-	-	-	.19	1.77	.74	.50	3.80
36	-	-	-	.43	3.36	1.27	.64	1.08
37	-	-	-	.54	3.79	1.96	.81	.88
38	-	-	-	103	199	3.54	1.24	997
39	-	-	-	1.67	11.6	6.56	1.62	.51
40	-	-	-	3.57	23.6	11.6	2.70	.49
41	-	-	-	5.57	38.2	18.5	3.32	.43
42	-	-	-	7.85	63.2	34.0	4.62	.57
43	-	-	-	17.2	133	66.8	6.29	.88
44	-	-	-	23.1	215	97.3	8.74	1.18
45	-	-	-	31.4	281	114	16.9	2.00
46	-	-	-	52.9	508	194	21.1	2.43
47	-	-	-	91.6	897	331	33.9	2.79
48	-	-	-	-	1331	-	-	3.10
49	-	-	-	-	2128	-	-	3.32
50	-	-	-	-	3249	-	-	3.14
51	-	-	-	-	-	-	-	3.91
52	-	-	-	-	-	-	-	4.23
53	-	-	-	-	-	-	-	5.44
54	-	-	-	-	-	-	-	8.30
55	-	-	-	-	-	-	-	14.9
56	-	-	-	-	-	-	-	18.9
57	-	-	-	-	-	-	-	25.7
58	-	-	-	-	-	-	-	32.8
59	-	-	-	-	-	-	-	41.3
60	-	-	-	-	-	-	-	47.8

Table 16. Average Time in Seconds to Partition $n = 40$ to 60 Integers 12 Ways

$k \rightarrow$	12-Way							
$n \downarrow$	CGA	RNP	IRNP	SNPIE	SNPESS	BSIBC	BSBCP	CIW
20	4.83	98.2	.78	.00	.00	.00	.02	.02
21	12.4	241	6.63	.00	.00	.00	.03	.19
22	8.25	434	18.6	.00	.00	.00	.04	.89
23	191	1239	-	.00	.00	.00	.05	9.08
24	432	3970	-	.00	.00	.00	.06	117
25	115	5586	-	.00	.00	.00	.07	426
26	325	27382	-	.00	.00	.00	.08	7428
27	2857	71124	-	.00	.00	.00	.08	1896
28	234	-	-	.00	.01	.00	.09	11977
29	9109	-	-	.00	.08	.00	.10	92996
30	5192	-	-	.00	.15	.00	.11	7556
31	40079	-	-	.00	.57	.01	.13	38493
32	38.3	-	-	.01	1.96	.02	.16	118
33	-	-	-	.04	1.49	.05	.18	17636
34	-	-	-	.07	4.90	.13	.21	11.6
35	-	-	-	.14	4.97	.26	.25	597
36	-	-	-	.39	6.16	.70	.31	363
37	-	-	-	.58	16.9	1.05	.45	14.1
38	-	-	-	142	8.27	2.45	.55	243
39	-	-	-	1.24	14.0	3.99	.90	3.29
40	-	-	-	733	491	10.1	1.10	3.27
41	-	-	-	5.23	41.7	12.2	1.52	1.98
42	-	-	-	6.79	66.7	24.6	2.33	1.60
43	-	-	-	12.8	135	47.8	3.35	1.23
44	-	-	-	28.0	275	89.9	4.27	1.12
45	-	-	-	38.3	408	123	6.27	1.63
46	-	-	-	63.2	820	243	8.99	1.91
47	-	-	-	98.2	1332	324	15.7	3.38
48	-	-	-	-	1830	-	-	4.72
49	-	-	-	-	2937	-	-	7.56
50	-	-	-	-	6011	-	-	13.9
51	-	-	-	-	-	-	-	13.5
52	-	-	-	-	-	-	-	13.8
53	-	-	-	-	-	-	-	16.2
54	-	-	-	-	-	-	-	13.6
55	-	-	-	-	-	-	-	17.4
56	-	-	-	-	-	-	-	14.6
57	-	-	-	-	-	-	-	16.2
58	-	-	-	-	-	-	-	23.8
59	-	-	-	-	-	-	-	36.1
60	-	-	-	-	-	-	-	68.2

REFERENCES

H. B. Amor and J. V. de Carvalho. 2005. *Cutting Stock Problems*. Springer.

G. Belov and G. Scheithauer. 2006. 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.

B. Chen. 2004. Parallel scheduling for early completion. In *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, Joseph Y. T. Leung (Ed.). CRC Press, Boca Raton, FL.

V. Chvatal. 1983. *Linear Programming*. WH Freeman, San Francisco, CA.

E. G. Coffman Jr. and J. L. Bruno. 1976. *Computer and Job-shop Scheduling Theory*. John Wiley & Sons, Hoboken, NJ.

E. G. Coffman Jr, M. R. Garey, and D. S. Johnson. 1978. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing* 7, 1, 1–17.

E. G. Coffman Jr, M. R. Garey, and D. S. Johnson. 1997. Approximation algorithms for bin packing: A survey. *Approximation Algorithms for NP-Hard Problems*, Dorit S. Hochbaum (Ed.). PWS Publishing Co., 46–93.

G. B. Dantzig and M. N. Thapa. 1997. *Linear Programming 1: Introduction*. Vol. 1. Springer.

G. B. Dantzig and M. N. Thapa. 2003. *Linear Programming 2: Theory and Extensions*. Vol. 1. Springer.

G. B. Dantzig and P. Wolfe. 1960. Decomposition principle for linear programs. *Operations Research* 8, 1, 101–111.

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

M. Dell'Amico and S. Martello. 1995. Optimal scheduling of tasks on identical parallel processors. *ORSA Journal on Computing* 7, 2, 191–200.

G. Dósa. 2007. The tight bound of first fit decreasing bin-packing algorithm is FFD (I) 11/9OPT (I)+ 6/9. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*. Springer, 1–11.

S. Eilon and N. Christofides. 1971. The loading problem. *Management Science* 17, 5, 259–268.

A. S. Fukunaga and R. E. Korf. 2005. *Bin Completion Algorithms for Packing and Knapsack Problems*. PhD Dissertation. University of California at Los Angeles.

A. Fukunaga and R. E. Korf. 2007. Bin completion algorithms for multicontainer packing, knapsack, and covering problems. *Journal of Artificial Intelligence Research* 28, 393–429.

M. R. Garey, R. L. Graham, and J. D. Ullman. 1972. Worst-case analysis of memory allocation algorithms. In *Proceedings of the 4th Annual ACM Symposium on Theory of Computing*. ACM, 143–150.

M. R. Garey and D. S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, CA.

R. E. Gomory. 1958. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society* 64, 5, 275–278.

R. L. Graham. 1966. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal* 45, 9, 1563–1581.

R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Kan. 1979. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics* 5, 287–326.

B. Hayes. 2002. The easiest hard problem. *American Scientist* 90, 2, 113–117.

E. Horowitz and S. Sahni. 1974. Computing partitions with applications to the knapsack problem. *Journal of the ACM* 21, 2, 277–292.

E. Huang and R. E. Korf. 2009. New improvements in optimal rectangle packing. In *IJCAI*. 511–516.

M. Iori and S. Martello. 2008. Scatter search algorithms for identical parallel machine scheduling problems. In *Metaheuristics for Scheduling in Industrial and Manufacturing Applications*. Springer, 41–59.

D. S. Johnson. 1973. *Near-optimal Bin Packing Algorithms*. Ph.D. dissertation. Massachusetts Institute of Technology, Cambridge, MA.

D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham. 1974. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing* 3, 4, 299–325.

N. Karmarkar and R. M. Karp. 1982. *The Differencing Method of Set Partitioning*. Computer Science Division (EECS), University of California.

R. M. Karp. 1972. *Reducibility Among Combinatorial Problems*. Springer.

H. Kellerer, U. Pferschy, and D. Pisinger. 2004. *Knapsack Problems*. Springer.

R. E. Korf. 1998. A complete anytime algorithm for number partitioning. *Artificial Intelligence* 106, 2, 181–203.

R. E. Korf. 2002. A new algorithm for optimal bin packing. In *Proceedings of the National Conference on Artificial Intelligence*. 731–736.

R. E. Korf. 2003. An improved algorithm for optimal bin packing. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03), Acapulco, Mexico*. 1252–1258.

R. E. Korf. 2009. Multi-way number partitioning. *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'09)*, 538–543.

R. E. Korf. 2011. A hybrid recursive multi-way number partitioning algorithm. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11), Barcelona, Catalonia, Spain*. 591–596.

R. E. Korf and E. L. Schreiber. 2013. Optimally scheduling small numbers of identical parallel machines. In *23rd International Conference on Automated Planning and Scheduling*.

R. E. Korf, E. L. Schreiber, and M. D. Moffitt. 2014. Optimal sequential multi-way number partitioning. In *International Symposium on Artificial Intelligence and Mathematics (ISAIM'14)*.

S. Martello and P. Toth. 1990a. *Knapsack Problems: Algorithms and Computer Implementations*. Chichester: John Wiley & Sons, New York, NY.

S. Martello and P. Toth. 1990b. Lower bounds and reduction procedures for the bin packing problem. *Discrete Applied Mathematics* 28, 1, 59–70.

S. Mertens. 2006. The easiest hard problem: Number partitioning. *Computational Complexity and Statistical Physics* 125, 2, 125–140.

M. D. Moffitt. 2013. Search strategies for optimal multi-way number partitioning. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*. AAAI Press, 623–629.

J. L. Mokotoff, E. Jimeno and A. I. Gutiérrez. 2001. List scheduling algorithms to minimize the makespan on identical parallel machines. *Top* 9, 2, 243–269.

M. Padberg and G. Rinaldi. 1991. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review* 33, 1. 60–100.

M. L. Pinedo. 2012. *Scheduling: Theory, Algorithms, and Systems*. Springer.

F. J. Provost. 1993. Iterative weakening: Optimal and near-optimal policies for the selection of search bias. In *AAAI*. 749–755.

V. Sarkar. 1989. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA.

J. E. Schoenfield. 2002. Fast, exact solution of open bin packing problems without linear programming. *Draft, US Army Space & Missile Defense Command* 45. Technical Report.

E. L. Schreiber and R. E. Korf. 2013. Improved bin completion for optimal bin packing and number partitioning. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI'13), Beijing, China*. AAAI Press, 651–658.

E. L. Schreiber and R. E. Korf. 2014. Cached iterative weakening for optimal multi-way number partitioning. In *Proceedings of the 28th Annual Conference on Artificial Intelligence (AAAI'14), Quebec City, Canada*. AAAI Press.

R. Schroeppel and A. Shamir. 1981. A T=O($2^{n/2}$), S=O($2^{n/4}$) algorithm for certain NP-complete problems. *SIAM Journal of Computing* 10, 3, 456–464.

T. Walsh. 2009. Where are the really hard manipulation problems? The phase transition in manipulating the veto rule. In *IJCAI*. 324–329.

B. Yakir. 1996. The differencing algorithm LDM for partitioning: A proof of a conjecture of Karmarkar and Karp. *Mathematics of Operations Research* 21, 1, 85–99.

M. Yue. 1991. A simple proof of the inequality FFD (L) 11/9 OPT (L)+ 1, L for the FFD bin-packing algorithm. *Acta Mathematicae Applicatae Sinica* 7, 4, 321–331.

Received August 2016; revised May 2017; accepted January 2018