Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence

Cached Iterative Weakening for Optimal Multi-Way Number Partitioning

Ethan L. Schreiber and **Richard E. Korf**
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095 USA
{ethan,korf}@cs.ucla.edu

Abstract

The NP-hard number-partitioning problem is to separate a multiset S of n positive integers into k subsets, such that the largest sum of the integers assigned to any subset is minimized. The classic application is scheduling a set of n jobs with different run times onto k identical machines such that the makespan, the time to complete the schedule, is minimized. We present a new algorithm, cached iterative weakening (CIW), for solving this problem optimally. It incorporates three ideas distinct from the previous state of the art: it explores the search space using iterative weakening instead of branch and bound; generates feasible subsets once and caches them instead of at each node of the search tree; and explores subsets in cardinality order instead of an arbitrary order. The previous state of the art is represented by three different algorithms depending on the values of n and k. We provide one algorithm which outperforms all previous algorithms for $k \geq 4$. Our run times are up to two orders of magnitude faster.

1 Introduction and Overview

The NP-hard number-partitioning problem is to separate a multiset S of n positive integers into k mutually exclusive and collectively exhaustive subsets $\langle S_1, ..., S_k \rangle$ such that the largest subset sum is minimized (Garey and Johnson 1979). For example, consider S=$\{8, 6, 5, 3, 2, 2, 1\}$ and k=3. The optimal partition is $\langle \{8, 1\}, \{5, 2, 2\}, \{6, 3\} \rangle$ with cost 9. This is a perfect partition since we can't do better than dividing the total sum of 27 into three subsets with sums 9 each. The classic application is scheduling a set of n jobs with different run times onto k identical machines such that the makespan, the time to complete the schedule, is minimized.

There is a large literature on solving this problem. The early work (Graham 1969; Coffman Jr, Garey, and Johnson 1978; Karmarkar and Karp 1982; França et al. 1994; Frangioni, Necciari, and Scutella 2004; Alvim and Ribeiro 2004) focused on approximation algorithms. (Korf 1998) presented an optimal algorithm for the 2-way partition problem. Since then, there has been work on number partitioning in both the artificial intelligence (Korf 2009; 2011; Moffitt 2013; Schreiber and Korf 2013) and operations research (Dell'Amico and Martello 1995; Mokotoff 2004; Dell'Amico et al. 2008) communities.

Copyright © 2014, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Number partitioning is closely related to the bin-packing problem (Garey and Johnson 1979). While number partitioning fixes the number of subsets k and minimizes the sum of the largest subset, bin packing fixes the maximum sum of the subsets (bins) and minimizes the number of subsets needed. Any number-partitioning algorithm can be used to solve a bin-packing problem and any bin-packing algorithm can be used to solve a number-partitioning problem.

In section 2, we describe the previous state of the art for optimally solving number partitioning. In section 3, we describe cached iterative weakening (CIW), our new state-of-the-art algorithm. In section 4, we experimentally compare CIW to the previous state of the art.

2 Background: SNP, MOF and BSBCP

The previous state of the art is represented by three different algorithms depending on n and k. There are two artificial intelligence algorithms, sequential number partitioning (SNP) (Korf, Schreiber, and Moffitt 2013) and the (Moffitt 2013) algorithm (MOF); and one operations research algorithm, binary search branch-and-cut-and-price (Belov and Scheithauer 2006; Schreiber and Korf 2013). Conceptually, CIW builds upon the AI algorithms, which we describe in detail in the remainder of this section. Abstractly, both SNP and MOF generate all feasible first subsets, then for each subset, recursively partition the remaining integers $k-1$ ways.

2.1 Bounds on the Subset Sums

The multiway version of the Karmarkar-Karp (KK) (Karmarkar and Karp 1982; Korf and Schreiber 2013) set differencing algorithm is an approximation algorithm for partitioning n integers into k sets. SNP and MOF calculate the initial upper bound (ub) on the sum of each subset using the cost of the KK partition of S into k subsets.

Given an upper bound (ub), the lower bound (lb) is $lb = \text{sum}(S) - (k-1)(ub-1)$ where $\text{sum}(S)$ is the sum of all integers of the set S. This is the smallest sum for the first subset that allows the remaining integers to be partitioned into $k-1$ subsets, all of whose sums are less than ub.

2.2 Recursive Partitioning

Both SNP and MOF generate all first subsets S_1 whose sums are between lb and $ub-1$. Then, they recursively partition the remaining integers $k-1$ ways into the partition

$\langle S_2, ..., S_k \rangle$. The algorithms look for lowest-cost complete partitions using depth-first branch and bound. The upper bound (ub) is the cost of the lowest-cost complete partition found so far. Initially, ub is set to the KK partition cost. As complete partitions with lower cost are found, ub is set to the new cost. For each partial partition $\langle S_1, ..., S_d \rangle$, SNP and MOF use ub, the remaining integers S^R and the depth d to compute a lower bound $lb = \text{sum}(S^R) - (k-d)(ub-1)$. If $lb \geq ub$, the search returns ub.

Otherwise, SNP and MOF generate all subsets S_d with sums in the range $[lb, ub-1]$ one at a time from S^R to create the partial partitions $P_d = \langle S_1, ..., S_{d-1}, S_d \rangle$ at depth d. For each partial partition P_d, the algorithms recursively partition S^R, the remaining integers, $k-d$ ways. If the cost of any of these recursive partitions is less than maxsum(P_d), the recursive search returns immediately. Since the cost of a partial partition is the max over its subset sums, maxsum(P_d) is the lowest possible cost for a complete partition that includes P_d. Otherwise, the algorithm returns the lesser of ub and the lowest-cost recursive partitioning.

Avoiding Duplicates The order of subsets in a partition is not important. For example, the partition $\langle \{8,1\}, \{5,2,2\}, \{6,3\} \rangle$ is equivalent to $\langle \{5,2,2\}, \{8,1\}, \{6,3\} \rangle$. In order to avoid such duplicates, the largest remaining integer is always included in the next subset of the partition.

2.3 Generating Subsets with sums in a Given Range

The first step of each partitioning is to generate all subsets from the remaining integers S^R whose sums fall in the range $[lb, ub-1]$. The core difference between SNP and MOF is the method for generating these subsets.

Inclusion-Exclusion (IE) MOF uses the inclusion-exclusion algorithm (IE) to generate subsets (Korf 2009). It searches a binary tree with each depth corresponding to an integer of S. Each node includes the integer in the subset on the left branch and excludes it on the right. The leaves correspond to complete subsets. IE sorts S then considers the integers in decreasing order, searching the tree from left to right always including integers before excluding them. It prunes the tree if the sum of the integers included at a node exceeds $ub-1$. Similarly, it prunes if the sum of the included integers at a node plus all non-assigned integers below the node is less than lb. In the worst case, this runs in time $O(2^n)$ and space $O(n)$. An IE tree is searched at each node of the partition search tree.

Extended Schroeppel and Shamir (ESS) The (Horowitz and Sahni 1974) algorithm (HS) for the subset sum problem (Is there a subset of S whose sum is equal to a $target$ value?) uses memory to improve upon the run time of IE. HS runs in time $O\left(\frac{n}{2}\, 2^{n/2}\right)$ and space $O\left(2^{\frac{n}{2}}\right)$. This is much faster than IE for large n but its memory requirements limit it to about $n = 50$ integers. It is not as fast for small n because of initial overhead. The (Schroeppel and Shamir 1981) algorithm (SS) is based on HS, but uses only $O(2^{n/4})$ space, limiting it to about $n = 100$ integers.

Both HS and SS solve the subset sum problem. With a little extra work, they can generate all sets in a range (Korf 2011). SNP generates subsets using the ranged version of SS, called extended Schroeppel and Shamir (ESS).

2.4 Binary-Search Branch-and-Cut-and-Price

Binary-search branch-and-cut-and-price (BSBCP), an operations research algorithm, solves number partitioning by solving a sequence of bin packing problems on S, varying the bin capacity C (maximum subset size). BSBCP performs a binary search over $[lb, ub]$ searching for the smallest value of C such that S can be packed into k bins. Our BSBCP implementation uses the BCP bin-packing solver from (Belov and Scheithauer 2006). Also see (Coffman Jr, Garey, and Johnson 1978; Schreiber and Korf 2013).

3 Cached Iterative Weakening (CIW)

Our new algorithm, cached iterative weakening (CIW), performs a recursive partitioning like sequential number partitioning (SNP) and the (Moffitt 2013) (MOF) algorithm. However, the methodology is different.

Call C^* the largest subset sum of an optimal partition for a particular number-partitioning problem. While searching for C^*, both SNP and MOF start with ub set to the KK partition which is typically much larger than C^*. They then search for better partitions until they find one with cost C^*. At this point, they verify optimality by proving there is no partition with all subsets having sums less than C^*. In contrast, CIW only considers partitions with cost less than or equal to C^*.

When constructing partitions, for each partial partition $\langle S_1, ..., S_{d-1} \rangle$, SNP and MOF generate the next subsets S_d using extended Schroeppel and Shamir (ESS) and inclusion-exclusion (IE) respectively. In contrast, CIW generates feasible sets once using ESS and caches them before performing the recursive partitioning.

3.1 Iterative Weakening

CIW begins by calculating $perfect = \lceil \text{sum}(S)/k \rceil$, a lower bound on partition cost, achieved if the sum of each of the k subsets of a partition differ by no more than one. In any partition, there must be at least one subset whose sum is at least as large as $perfect$.

Whereas SNP and MOF recursively partition S into k subsets decreasing ub until the optimal cost C^* is found and subsequently verified, CIW starts with ub set to $perfect$ and tries to recursively partition S into k sets no greater than ub. It iteratively increases ub until it finds C^*, the first value for which a partition is possible. This process is called iterative weakening (Provost 1993). In order to verify optimality, any optimal algorithm must consider all partial partitions with costs between $perfect$ and C^*. Even after a branch and bound algorithm finds an optimal partition of cost C^*, it still needs to verify its optimality. Iterative weakening **only** explores partial partitions with costs between $perfect$ and C^*.

Suppose we could efficiently generate subsets one by one in sum order starting with $perfect$. CIW iteratively chooses each of these subsets as the first subset S_1 of a partial partition. It sets ub to sum(S_1) and lb to $\text{sum}(S) - (k-1)(ub)$.

Then, given that it can efficiently generate all subsets in the range $[lb, ub]$, it determines whether there are $k-1$ of these subsets that are mutually exclusive and contain all the integers in $S^R = S - S_1$. If this is possible, ub is returned as the optimal partition cost. Otherwise, CIW moves onto the subset with the next larger sum. In this scheme, the cost of a partial partition is always the sum of its first subset S_1.

In the next section, we will discuss how to enable CIW to efficiently examine subsets one by one in sum order starting with $perfect$. In section 3.3, we will discuss how to efficiently determine if it is possible to partition the remaining integers S^R into $k-1$ subsets with sums in range $[lb, ub]$ at each iteration of iterative weakening.

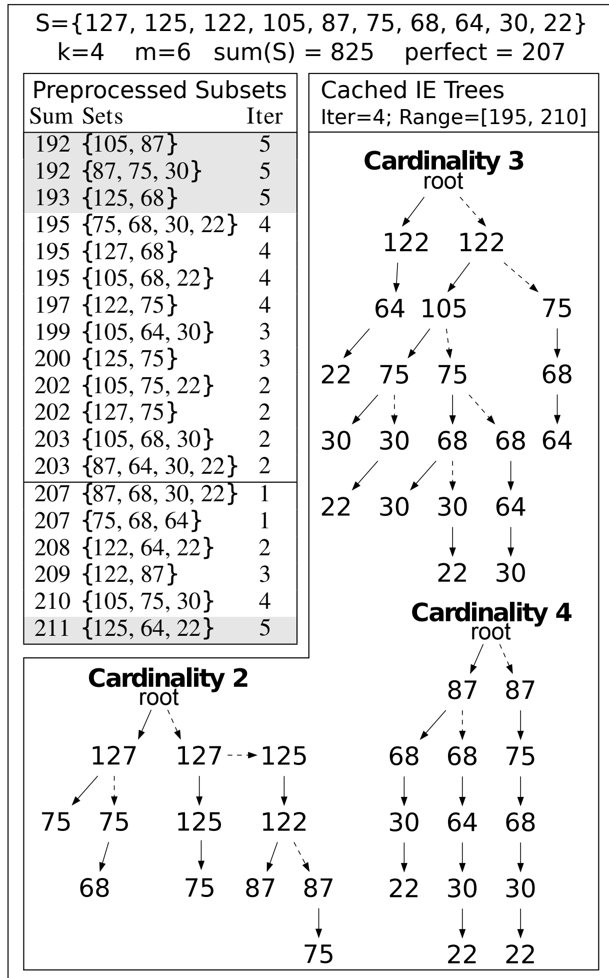

Figure 1: CIW example with the list of precomputed subsets and cached trees for cardinality 2, 3 and 4 during iteration 4.

3.2 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 $perfect$.

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

Extended Schroeppel and Shamir (ESS) efficiently generates all subsets in a given range. We wish to generate all subsets with sums in the range $[min, max]$. Unfortunately, we do not know the values of min and max before generating the m subsets with sums greater than or equal to perfect.

Therefore, in order to generate all subsets in the range $[min, max]$, CIW modifies ESS to use a min-heap and a max-heap. It initially sets max to the KK ub and min to the corresponding lb using ESS to generate sets with sums in this range. It puts each subset found with sum in the range $[min, perfect - 1]$ into the min-heap and those in the range $[perfect, max]$ into the max-heap. This continues until the max-heap contains m subsets. At this point, CIW resets max to the sum of the largest subset in the max-heap and recalculates min as $\text{sum}(S)-(k-1)(max)$. It then pops all subsets with sums less than min from the min-heap.

ESS continues searching for all sets in the new range $[min, max]$. Each time a subset with sum greater than or equal to $perfect$ but less than max is found, CIW pops the top subset from the max-heap and pushes this new subset onto the heap. max is set to the new max sum and min is updated accordingly, popping all subsets with sum less than min from the min-heap. When this modified ESS search is complete, the subsets from the min-heap and the max-heap are moved to one 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 $perfect$. If m iterations are performed without finding an optimal partition, the algorithm is run again to generate the next $2m$ subsets. If the $2m$ subsets are exhausted without finding an optimal solution, then $4m$ subsets are generated, then $8m$, etc. Thus, m is a parameter of CIW. We discuss setting m experimentally in section 4.

Example Consider the example number-partitioning problem with S = {127, 125, 122, 105, 87, 75, 68, 64, 30, 22} and $k = 4$. Figure 1 shows the array of 19 sets generated by modified ESS with $m = 6$ in the table at the top left. The final range $[min, max]$ is $[192, 211]$ and $perfect = 207$. There are 13 sets with sums in the range $[min, perfect - 1]$ shown above the horizontal line and 6 sets with sums in the range $[perfect, max]$ shown below the horizontal line. The 6 subset sums below 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 subset in that row first appears in the range $[lb, ub]$. For example, in the first iteration, $ub = 207$ and $lb = 825 - 3 \times 207 = 204$ with two subsets in range. In the second iteration, $ub = 208$ and $lb = 825 - 3 \times 208 = 201$ with seven sets in range, namely the rows with $Iter$ equal to 1 or 2. In the sixth iteration, all 19 sets in the table are in range.

3.3 Recursive Partitioning

At each iteration of iterative weakening, CIW chooses the first subset S_1 as the next subset with sum at least as large as $perfect$ from the stored array of subsets, sets $ub = \text{sum}(S_1)$ and $lb = sum(S) - (k-1)(ub)$. At this point, CIW attempts to recursively partition $S^R = S - S_1$ into $\langle S_2, ..., S_k \rangle$. Since all of the subsets in the range $[lb, ub]$ have already been generated, this is a matter of finding $k-1$ of these subsets which are mutually exclusive and contain all the integers of S^R.

A Simple Algorithm Given an array A of subsets, we present a recursive algorithm for determining if there are k mutually exclusive subsets containing all the integers of S.

For each first subset S_1 in A, copy all subsets of A that do not contain an integer in S_1 into a new array B. Then, recursively try to select $k-1$ disjoint subsets from B which contain the remaining integers of $S - S_1$. If $k = 0$, return true, else if the input array A is empty, return false.

While this algorithm is correct, it is inefficient, as the entire input array must be scanned for each recursive call. In the next two sections, we present an efficient algorithm which performs the same function.

Cached Inclusion-Exclusion (CIE) Trees After CIW chooses the first subset S_1 from the precomputed array, it uses cached inclusion-exclusion (CIE) to test if there are $k-1$ mutually exclusive subsets which contain all integers in S^R=$S - S_1$. CIE trees store all subsets whose sums are in the range $[lb, ub]$ of the current iterative weakening iteration. They are built incrementally by inserting all subsets in the new range $[lb, ub]$ at each iteration of iterative weakening. CIE trees are similar to IE trees (Section 2.3).

With IE trees, all feasible subsets, regardless of cardinality, can be found in one tree. In contrast, there is one CIE tree for each unique cardinality of feasible subset. The distribution of the cardinality of the subsets in range $[lb, ub]$ is not even. The average cardinality of a subset in an optimal solution is n/k. Typically, most subsets in an optimal solution have cardinality close to this average. Yet, there are often many more subsets with higher cardinality than n/k. In section 3.3, we will show how to leverage these cardinality trees so CIW never has to examine the higher cardinality subsets. In the example of figure 1, there are separate trees for subsets of cardinality 2, 3 and 4 storing all subsets with sums in the range $[195, 210]$ (every subset in iteration 1 through 4).

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

The nodes of each CIE tree correspond to one of the integers in S. The integer is included on the left branch and excluded on the right. In figure 1, solid arrows correspond to inclusion of the integer pointed to while dashed arrows correspond to exclusion. For example, in the cardinality two tree, from the root, following the left solid arrow to 127, then

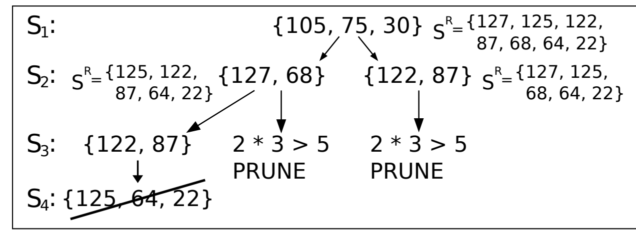

Figure 2: The recursive partitioning search tree for iteration 4.

the left solid arrow to 75 ($root \rightarrow 127 \rightarrow 75$) corresponds to the subset $\{127, 75\}$. Similarly, $root \rightarrow 127 \dashrightarrow 75 \rightarrow 68$ corresponds to the subset $\{127, 68\}$.

Recursive Partitioning with CIE Trees After selecting S_1 and calculating lb and ub, CIW adds all subsets with sums newly in the range $[lb, ub]$ from the stored array of subsets into the CIE tree of proper cardinality. If CIW finds $k-1$ mutually exclusive subsets containing all the integers of $S^R = S - S_1$, then the optimal cost is $ub = \text{sum}(S_1)$.

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 S_2, CIE searches its smallest cardinality tree first. Call $card$ the cardinality of the tree CIE is searching to generate S_d in partial partition $\langle S_1, ..., S_d \rangle$. If CIE finds a subset S_d of cardinality $card$, the recursive search begins searching for subset S_{d+1} in the $card$ CIE tree. If no more subsets are found in the $card$ CIE tree, the $card+1$ CIE tree is searched until no higher cardinality CIE tree exists. CIW prunes if $k - d \times card > |S^R|$ since there are not enough integers left in S^R to create $k - d$ subsets, each with cardinality $\geq card$.

Avoiding Duplicates Choosing subsets in cardinality order avoids many duplicates. However, if S_d and S_{d+1} in partial solution P_{d+1} have equal cardinality, to remove duplicates, the largest integer in S_{d+1} must be smaller than 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 $\{6,3\}$.

Example Figure 2 shows the recursive partitioning search tree for iteration 4 of our running example using the CIE trees from figure 1. The root of the tree is the subset $S_1 = \{105, 75, 30\}$ whose sum 210 is the ub for iteration 4 of iterative weakening. The search of candidate subsets for S_2 begins in the cardinality 2 CIE tree, including nodes before excluding them. Starting from the root of the CIE tree, CIW includes 127 but cannot include 75 since it is included in S_1. It excludes 75 and includes 68 giving us $S_2 = \{127, 68\}$. We continue to search the cardinality 2 CIE tree for S_3 but the largest integer must be less than 127 to avoid duplicates, so we exclude 127 and then include 125. We cannot include 75 since it is not in S^R, so we backtrack to exclude

$k \rightarrow$	3-Way			4-Way			5-Way			6-Way			7-Way		
$n \downarrow$	CIW	SNP	R	CIW	SNP	R	CIW	SNP	R	CIW	SNP	R	CIW	SNP	R
40	.161	.066	.408	.153	.149	.973	.158	.364	2.30	.228	.834	3.66	.219	1.96	8.97
41	.237	.114	.481	.225	.247	1.10	.217	.586	2.70	.201	1.29	6.41	.207	3.02	14.6
42	.313	.136	.433	.298	.328	1.10	.299	.856	2.87	.292	2.16	7.38	.292	5.03	17.2
43	.498	.221	.444	.465	.513	1.10	.460	1.27	2.77	.432	3.17	7.35	.407	7.47	18.4
44	.672	.278	.414	.602	.680	1.13	.528	1.77	3.36	.552	4.35	7.87	.522	11.1	21.3
45	.972	.486	.500	.929	1.13	1.22	.822	2.64	3.21	.781	7.07	9.05	.753	19.5	25.9
46	1.37	.592	.431	1.22	1.47	1.20	1.06	4.27	4.04	.952	10.9	11.4	.899	26.7	29.7
47	2.10	.971	.462	2.07	2.26	1.09	1.83	6.30	3.43	1.67	16.6	9.89	1.41	41.4	29.3
48	3.07	1.18	.385	2.58	3.18	1.23	2.16	9.30	4.30	1.93	24.6	12.7	1.61	62.9	39.1
49	4.68	2.06	.440	3.96	5.09	1.29	3.53	14.0	3.95	3.09	37.9	12.3	2.67	92.9	34.8
50	6.12	2.44	.399	5.25	6.89	1.31	4.35	18.6	4.28	3.44	50.7	14.7	2.93	135	46.3
51	9.40	4.10	.436	8.79	10.7	1.22	7.76	32.2	4.15	6.74	89.2	13.2	5.43	233	43.0
52	13.8	4.93	.357	11.1	14.4	1.30	8.93	44.4	4.97	7.55	125	16.5	6.36	364	57.1
53	20.4	8.81	.433	17.9	22.6	1.26	15.5	67.1	4.33	13.6	199	14.7	10.6	529	49.8
54	27.3	10.5	.384	22.8	31.7	1.39	18.8	104	5.52	15.4	309	20.1	12.2	864	70.8
55	44.6	17.2	.385	39.3	48.4	1.23	35.3	157	4.45	30.4	472	15.5	24.2	1328	55.0
56	66.9	20.7	.309	52.5	67.8	1.29	41.3	225	5.46	35.2	698	19.9	27.1	1945	71.8
57	98.0	36.7	.374	86.1	103	1.20	73.3	349	4.76	59.8	1119	18.7	46.6	3158	67.8
58	135	45.4	.336	108	143	1.33	83.1	536	6.45	68.0	1677	24.7	53.0	5015	94.7
59	221	73.6	.334	191	237	1.24	164	771	4.70	139	2448	17.6	106	7520	70.8
60	301	89.6	.298	260	307	1.18	206	1140	5.54	154	3728	24.1	119	10715	90.0

Table 1: Average time in seconds to optimally partition uniform random 48-bit integers 3, 4, 5, 6 and 7 ways.

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, so we prune. (Note that the set {125, 64, 22} is not in the cardinality 3 CIE tree.) We backtrack to generate the next S_3 subset. Continuing where we left off in the cardinality 2 tree when we generated $S_3 = \{122, 87\}$, we backtrack to exclude 87 but since 75 is not in S^R, we have exhaustively searched the cardinality 2 CIE tree. We move to the cardinality 3 tree. However, there are five integers left to partition into two subsets and the cardinality of the subsets left must be three or greater. Since $2 \times 3 > 5$, we prune.

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

In the next iteration, iterative weakening will set $S1 = \{125, 64, 22\}$ with $ub = \text{sum}(S_1) = 211$ and add the four gray rows with $Iter = 5$ from the table in figure 1 to the CIE trees. It is possible to partition S into $k = 4$ subsets all with sums less than 211, resulting in the optimal partition $\langle\{125, 64, 22\}, \{127, 75\}, \{122, 87\}, \{105, 68, 30\}\rangle$.

4 Experiments

We ran experiments for n from 40 to 60 and k from 3 to 12. [1] Generally, for these values of n, sequential number partitioning (SNP) is the previous state of the art for $k = 3$ to 7, the (Moffitt 2013) algorithm (MOF) for $k = 8$ to 10 and binary-search branch-and-cut-and-price (BSBCP) for $k = 11$ and 12. [2] We compare cached iterative weakening (CIW) to SNP for $k \leq 7$, to MOF for k from 8 to 10 and to BSBCP for k = 11 and 12. For each combination of n and k, we generated 100 problem instances. Each instance consists of n integers sampled uniformly at random in the range $[1, 2^{48} - 1]$ in order to generate hard instances without perfect partitions (Korf 2011). These are the first published results for high-precision number partitioning with $n > 50$. All experiments were run on an Intel Xeon X5680 CPU at 3.33GHz.

For a particular k, since the previous state of the art depends on n, one might infer that a hybrid recursive algorithm that uses one of SNP, MOF or BSBCP for recursive calls depending on n and k would outperform the individual algorithms. However, (Korf, Schreiber, and Moffitt 2013) shows that the hybrid algorithm in fact does not significantly outperform the best of the individual algorithms.

For CIW, we need to choose a value for m, the number of subsets to initially generate during the precomputing phase (section 3.2). Ideally, we want m to be exactly the number of subsets in the range $[perfect, C^*]$, but we do not know this number in advance. If m is set too small, CIW will have to run ESS multiple times. If m is set too large, CIW will waste time generating subsets in the precomputing step that

[1] Benchmarks at https://sites.google.com/site/elsbenchmarks/.

[2] There are exceptions for ($n \leq 45$; k=7), where MOF is faster than SNP and ($n \leq 46$; k=11), where MOF is faster than BSBCP.

$k \rightarrow$	8-Way			9-Way			10-Way			11-Way			12-Way		
$n \downarrow$	CIW	MOF	R	CIW	MOF	R	CIW	MOF	R	CIW	BCP	R	CIW	BCP	R
40	.189	1.24	6.58	.275	1.25	4.54	.404	1.39	3.45	.490	2.68	5.47	3.27	1.12	.342
41	.226	2.24	9.89	.247	2.08	8.44	.441	2.03	4.60	.427	3.29	7.70	1.98	1.52	.769
42	.330	3.77	11.4	.344	3.26	9.48	.630	3.27	5.20	.567	4.54	8.00	1.60	2.25	1.41
43	.433	5.88	13.6	.447	5.35	12.0	.644	5.27	8.18	.883	6.22	7.05	1.23	3.33	2.70
44	.510	8.45	16.6	.506	8.65	17.1	.840	9.25	11.0	1.18	8.60	7.28	1.12	4.28	3.82
45	.677	14.3	21.1	.713	13.7	19.1	.968	14.0	14.5	2.00	15.1	7.53	1.63	6.55	4.02
46	.820	23.7	28.9	.935	24.8	26.6	.913	21.9	24.0	2.43	21.2	8.74	1.91	8.85	4.63
47	1.25	38.2	30.6	1.39	41.4	29.7	1.35	42.6	31.6	2.79	30.5	10.9	3.38	14.7	4.36
48	1.48	60.0	40.4	1.55	59.4	38.2	1.42	56.5	40.0	3.10	48.9	15.8	4.72	24.4	5.16
49	2.20	100	45.6	2.04	98.0	48.0	1.93	92.8	48.2	3.32	65.6	19.7	7.56	32.4	4.28
50	2.29	166	72.6	2.23	154	68.8	2.32	141	60.9	3.14	92.8	29.6	13.9	55.3	3.97
51	4.25	281	66.1	3.42	274	80.0	3.82	263	69.0	3.91	146	37.5	13.5	64.3	4.75
52	5.08	418	82.3	3.97	382	96.1	5.28	362	68.5	4.23	187	44.3	13.8	85.6	6.19
53	8.45	707	83.7	6.77	724	107	7.56	735	97.2	5.44	273	50.3	16.2	109	6.71
54	9.33	1189	127	8.12	1116	137	8.02	1120	140	8.30	445	53.6	13.6	138	10.1
55	18.4	2082	113	14.5	1701	117	14.5	1920	133	14.9	721	48.4	17.4	250	14.4
56	20.6	3078	149	16.7	2878	172	14.4	2700	187	18.9	1325	70.2	14.6	315	21.5
57	34.7	5589	161	23.8	4983	209	18.0	4950	275	25.7	2354	91.6	16.2	464	28.7
58	41.1	9182	223	29.1	7636	262	23.7	8508	359	32.8	3962	121	23.8	807	33.9
59	71.2	13441	189	48.9	11874	243	35.6	12249	344	41.3	7292	177	36.1	1245	34.5
60	80.9	23085	285	56.6	21414	378	42.0	18036	429	47.8	11971	250	68.2	2225	32.6

Table 2: Average time in seconds to optimally partition uniform random 48-bit integers 8, 9, 10, 11 and 12 ways.

are never used in the iterative weakening step.

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

Table 1 compares CIW to SNP for k from 3 to 7. Each row corresponds to a value of n. There are three columns for each value of k. The first two columns report the average time to partition n integers into k subsets over 100 instances using CIW and SNP respectively. The third column is the ratio of the run time of SNP to CIW.

CIW is faster than SNP for $k \geq 4$ with the exception of $(n \leq 42; k = 4)$. SNP is faster than CIW for $k = 3$. For fixed n, SNP tends to get slower as k gets larger while CIW tends to get faster as k gets larger. For $5 \leq k \leq 7$, the ratios of the run times of SNP to CIW tend to grow as n gets larger, suggesting CIW is asymptotically faster than SNP. For $k = 3$ and 4, there is no clear trend. The biggest difference in the average run times is for $(n = 58; k = 7)$ where SNP takes 94.7 times longer than CIW.

Table 2 shows data in the same format as table 1 but for k from 8 to 12 and this time comparing CIW to MOF for k from 8 to 10 and CIW to BSBCP for 11 and 12. CIW outperforms both MOF and BSBCP for all n and k. The ratios of the run times of MOF to CIW and BSBCP to CIW grow as n gets larger for all k, again suggesting that CIW is asymptotically faster than MOF and BSBCP. The biggest difference in the average run times of MOF and CIW is for $(n = 60; k = 10)$ where MOF takes 429 times longer than CIW. The biggest difference for BSBCP is for $(n = 60; k = 11)$ where BSBCP takes 250 times longer than CIW.

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

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

5 Conclusions

The previous state of the art for optimally partitioning integers was represented by sequential number partitioning (Korf, Schreiber, and Moffitt 2013), the (Moffitt 2013) algorithm and binary-search branch and cut and price (Dell'Amico et al. 2008; Schreiber and Korf 2013) depending on n and k. We have presented cached iterative weakening (CIW), which outperforms all algorithms for $k \geq 4$.

CIW partitions S into k subsets like the previous AI algorithms but has three major improvements. It explores the search space using iterative weakening instead of branch and bound; generates feasible subsets once and caches them instead of at each node of the search tree; and explores subsets in cardinality order instead of an arbitrary order. These improvements make CIW up to two orders of magnitude faster than the previous state of the art.

Number partitioning is sometimes called the "easiest hard problem" (Mertens 2006). As compared to other NP-hard problems, it has very little structure. Nonetheless, there

have been continuous algorithmic improvements of orders of magnitude for solving this problem for over four decades. This leads us to believe that similar gains should be possible for more highly structured NP-hard problems.

References

Alvim, A. C., and Ribeiro, C. C. 2004. A hybrid bin-packing heuristic to multiprocessor scheduling. In *Experimental and Efficient Algorithms*. Springer. 1–13.

Belov, G., and Scheithauer, G. 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.

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

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

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

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

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

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

Graham, R. L. 1969. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics* 17(2):416–429.

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

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

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

Korf, R. E.; Schreiber, E. L.; and Moffitt, M. D. 2013. Optimal sequential multi-way number partitioning. In *International Symposium on Artificial Intelligence and Mathematics (ISAIM-2014)*.

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

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

Korf, R. E. 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.

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

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

Mokotoff, E. 2004. An exact algorithm for the identical parallel machine scheduling problem. *European Journal of Operational Research* 152(3):758–769.

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

Schreiber, E. L., and Korf, R. E. 2013. Improved bin completion for optimal bin packing and number partitioning. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, 651–658. AAAI Press.

Schroeppel, R., and Shamir, A. 1981. A t=o(2ˆn/2), s=o(2ˆn/4) algorithm for certain np-complete problems. *SIAM journal on Computing* 10(3):456–464.