Optimal Sequential Multi-Way Number Partitioning

Richard E. Korf, Ethan L. Schreiber, and Michael D. Moffitt
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
IBM Corp.
11400 Burnet Road
Austin, TX 78758
korf@cs.ucla.edu, ethan@cs.ucla.edu, moffitt@us.ibm.com

Abstract

Given a multiset of n positive integers, the NP-complete problem of number partitioning is to assign each integer to one of k subsets, such that the largest sum of the integers assigned to any subset is minimized. Last year, three different papers on optimally solving this problem appeared in the literature, two from the first two authors, and one from the third author. We resolve here competing claims of these papers, showing that different algorithms work best for different values of n and k, with orders of magnitude differences in their performance. We combine the best ideas from both approaches into a new algorithm, called sequential number partitioning, and also introduce a hybrid algorithm that achieves the best performance for each value of n and k. Number partitioning is closely related to bin-packing, and advances in either problem can be applied to the other.

Introduction and Overview

Given a multiset of n positive integers, the NP-complete problem of number partitioning is to assign each integer to one of k subsets, so that the largest sum of the integers assigned to any subset is minimized (Garey and Johnson 1979). For example, an optimal two-way partition of the integers $\{4, 5, 6, 7, 8\}$ is $\{4, 5, 6\}$ and $\{7, 8\}$, since both subsets sum to 15, which minimizes the largest subset sum. This is perhaps the simplest NP-complete problem to describe.

One application of number partitioning is to scheduling. Given a set of n jobs, each with an associated running time, and a set of k identical machines, such as computers or CPU cores, schedule each job on a machine in order to complete all the jobs as soon as possible. The completion time of each machine is the sum of the running times of the jobs assigned to it, while the total completion time is the longest completion time of any machine. Another application is to voting manipulation (Walsh 2009).

We begin with two-way partitioning, then consider multi-way partitioning. In each case, we describe the relevant algorithms, introduce two new algorithms for multi-way partitioning, and compare their performance experimentally.

Two-Way Number Partitioning

The subset-sum problem is to find a subset of a set of integers whose sum is closest to a given target value. Two-way partitioning is a special case of this problem, where the target value is half the sum of all the integers. We describe five different optimal algorithms for these problems. A sixth algorithm, dynamic programming, is not competitive in either time or space (Korf and Schreiber 2013).

Inclusion-Exclusion (IE)

Perhaps the simplest way to generate all subsets of a given set is to search a binary tree depth-first, where each level corresponds to a different element. Each node includes the element on the left branch, and excludes it on the right branch. The leaves correspond to complete subsets. We first sort the integers, then consider them in decreasing order, searching the tree from left to right. We prune the tree as follows. If the integers included at a given node exceed the sum of the best subset found so far, we prune that node. Similarly, if including all the remaining integers below a node does not generate a subset sum better than the best so far, we prune that node as well.

Complete Greedy Algorithm (CGA)

A similar algorithm that finds better solutions sooner is the complete greedy algorithm (CGA). It also sorts the integers in decreasing order, assigning a different integer at each level, and searches the same tree, but reorders the branches. The left branch of each node assigns the next integer to the subset with the smaller sum so far, and the right branch assigns it to the subset with the larger sum. Thus, the first solution found is that returned by the obvious greedy heuristic for this problem. CGA keeps track of the larger subset sum of the best solution found so far, and prunes a branch when the sum of either subset equals or exceeds this value.

Complete Karmarkar-Karp (CKK)

An even better algorithm is based on a heuristic approximation originally called set differencing (Karmarkar and Karp 1982), but usually referred to as KK. KK sorts the integers in decreasing order, and at each step replaces the two largest integers with their difference. This is equivalent to separating the two largest integers in different subsets, without committing to their final placement. For example, placing 8 and 7 in different subsets is equivalent to placing a 1 in the subset the 8 is assigned to. The difference is then treated as another integer to be assigned. The algorithm continues until only one integer is left, which is the difference between the subset sums of the final partition. Some additional bookkeeping is needed to construct the actual partition. The KK heuristic finds much better solutions than the greedy heuristic.

The Complete Karmarkar-Karp algorithm (CKK) is a complete optimal algorithm (Korf 1998). While KK always places the two largest integers in different subsets, the only other option is to place them in the same subset, by replacing them with their sum. Thus, CKK searches a binary tree where at each node the left branch replaces the two largest integers by their difference, and the right branch replaces them by their sum. The first solution found is the KK solution. If the largest integer equals or exceeds the sum of the remaining integers, they are placed in opposite subsets.

The time complexity of IE, CGA, and CKK are all about $O(2^n)$, where n is the number of integers. Pruning reduces this complexity slightly. Their space complexity is $O(n)$.

Horowitz and Sahni (HS)

Horowitz and Sahni (HS) presented a faster algorithm for the subset sum problem. It divides the n integers into two "half" sets a and c, each of size $n/2$. Then it generates all $2^{n/2}$ subsets of each half set, including the empty set. The two lists of subsets are sorted by their subset sums. Any subset of the original integers consists of a subset of the a integers concatenated with a subset of the c integers. Next, it initializes a pointer to the empty subset from the a list, and the complete subset from the c list. If the subset sum pointed to by the a pointer, plus the subset sum pointed to by the c pointer, is more than half the sum of all the integers, the c pointer is decremented to the subset with the next smaller sum. Alternatively, if the sum of the subset sums pointed to by the two pointers is less than half the total sum, the a pointer is incremented to the subset with the next larger sum. If the sum of the two subset sums equals half the total sum, the algorithm terminates. Else, HS continues until either list of subsets is exhausted, returning the best solution found.

HS runs in $O((n/2)2^{n/2})$ time and $O(2^{n/2})$ space. This is much faster than IE, CGA and CKK, but its memory requirement limits it to about 50 integers.

Schroeppel and Shamir (SS)

The (Schroeppel and Shamir 1981) algorithm (SS) is based on HS, but uses much less space. HS uses the subsets from the a and c lists in order of their subset sums. Rather than generating, storing, and sorting all these subsets, SS generates them as needed in order of their subset sums.

SS divides the n integers into four sets a, b, c and d, each of size $n/4$, generates all $2^{n/4}$ subsets of each set, and sorts them in order of their sums. The subsets from the a and b lists are combined in a min heap that generates all subsets of elements from a and b in increasing order of their sums. Each element of the heap consists of a subset from the a list, and a subset from the b list. Initially, it contains all pairs combining the empty set from the a list with each subset from the b list. The top of the heap contains the pair whose subset sum is the current smallest. Whenever a pair (a_i, b_j) is popped off the top of the heap, it is replaced in the heap by a new pair (a_{i+1}, b_j). Similarly, the subsets from the c and d lists are combined in a max heap, which returns all subsets from the c and d lists in decreasing order of their sums. SS uses these heaps to generate the subset sums in sorted order, and combines them in the same way as the HS algorithm.

SS runs in time $O((n/4)2^{n/2})$, but only requires $O(2^{n/4})$ space, making it practical for up to about 100 integers.

A recent algorithm reduces this runtime to approximately $O(2^{n/3})$ (Howgrave-Graham and Joux 2010), but is probabilistic, solving only the decision problem for a given subset sum. It cannot prove there is no solution for a given sum, and doesn't return the subset sum closest to a target value.

Efficiently Generating Complement Sets

For every subset generated, there is a complement subset. For efficiency, we do not want to generate both sets from scratch, but generate the complement sum by subtracting the original sum from the total sum. This optimization is obvious for the $O(2^n)$ algorithms described above. For CGA, for example, we only put the largest number in one of the subsets. To implement this for HS or SS, we simply exclude the largest integer when generating the original sets.

Performance of Two-Way Partitioning

Asymptotic Complexity

Our two-way partitioning algorithms fall into two classes: linear space and exponential space. Inclusion-exclusion (IE), the complete greedy algorithm (CGA) and the complete Karmarkar-Karp algorithm (CKK) each run in $O(2^n)$ time, and use only $O(n)$ space. Horowitz and Sahni (HS) runs in $O((n/2)2^{n/2})$ time and uses $O(2^{n/2})$ space, while SS runs in $O((n/4)2^{n/2})$ time and uses $O(2^{n/4})$ space.

Choice of Benchmarks

For the experiments in this paper, we use use integers chosen randomly and uniformly from zero to $2^{48}-1$. The reason for such high-precision integers is to avoid *perfect partitions*. If the sum of all the integers is divisible by the number of subsets k, then all subset sums are equal in a perfect partition. Otherwise, the subset sums in a perfect partition differ by at most one. The example at the beginning of this paper is an example of a perfect partition. Once a perfect partition is found, search terminates immediately, since any perfect partition is optimal. This makes problem instances with perfect partitions easier to solve, and those without perfect partitions more difficult. Thus, we use high-precision integers to create hard problems without perfect partitions.

Empirical Results

Among the $O(2^n)$ algorithms, CKK is the fastest, followed by CGA, and then IE. Their relative performance diverges with increasing n. At $n = 40$, CKK is about about twice as fast as CGA, and about about three times as fast as IE. Among the $O(n2^{n/2})$ algorithms, SS is about twice as fast as HS, and since it also uses much less memory, SS dominates HS. Both HS and SS have higher constant factor overheads than the $O(2^n)$ algorithms, however. For $n \leq 11$, CKK is the fastest, while for $n \geq 12$, SS is the fastest.

Columns 2, 3, and 4 of Table 1 show the performance of IE and SS for two-way partitioning of 30 through 50 integers. We chose to show IE to allow a comparison with the algorithm described in (Moffitt 2013), which is essentially IE for two-way partitioning. Columns 2 and 3 show running times in seconds, averaged over 100 different instances, for IE and SS, respectively. The empty positions in column 2 represent problems that took IE too long to run. Column 4 is the ratio of the running times of the two algorithms. As expected, this ratio increases monotonically with increasing n. For $n = 44$, SS is over 4500 times faster than IE.

Multi-Way Partitioning

For partitioning more than two ways, the previous state of the art is represented by two different algorithms: recursive number partitioning (RNP) (Korf and Schreiber 2013), and the (Moffitt 2013) algorithm.

Three-Way Partitioning

We begin with three-way partitioning. Both algorithms first run a polynomial-time heuristic to get an approximate solution, providing an upper bound b on the largest subset sum in an optimal solution. RNP uses the generalization of the two-way Karmarkar-Karp approximation to multiway partitioning, while (Moffitt 2013) uses the greedy approximation. Then, both algorithms construct first subsets which could be part of a better three-way partition. The upper bound on the sum of these subsets is $b - 1$, and the lower bound on their sum is $s - 2 * (b - 1)$, where s is the sum of all the integers. The reason for the lower bound is that it must be possible to partition the remaining integers into two sets, both of whose sums are less than b. To eliminate duplicate partitions that differ only by a permutation of the subsets, both algorithms include the largest integer in these first subsets.

To generate these subsets, (Moffitt 2013) uses the inclusion-exclusion (IE) algorithm, while RNP uses an extension of the Schroeppel and Shamir algorithm (ESS) (Korf 2011). Rather than generating a single subset whose sum is closest to a target value, ESS generates all subsets whose sums fall between the above lower and upper bounds.

For each first subset in the computed range, both algorithms partition the remaining integers two ways. While optimally partitioning the remaining elements two ways will produce the best three-way partition that includes the first subset, it's not strictly necessary. The remaining integers only have to be partitioned into two sets, both of whose sums are less than or equal to the sum of the first subset, since the overall objective function is the largest subset sum.

Both algorithms use branch-and-bound. If the largest subset sum in a new solution is less than b, the best so far, b is reduced to the new cost, and the lower bound is increased accordingly, until an optimal solution is found and verified.

Four or More-Way Partitioning

For four or more way partitioning, the two algorithms differ in another important respect. (Moffitt 2013) generates all possible first subsets with sums in the computed range, and for each of these, recursively partitions the remaining integers $k-1$ ways, generating the subsets sequentially. In addition to keeping track of the best solution so far, it also keeps track of m, the maximum subset sum among the completed subsets in the current solution. For each recursive call, it returns when it has partitioned the remaining integers into subsets whose sums are all less than or equal to m, rather than optimally partitioning the remaining integers. Also, to eliminate duplicate partitions, each subset always includes the largest remaining integer.

In contrast, for four-way partitioning, RNP generates all two-way partitions that could possibly be further divided into a four-way partition better than the current best so far. In other words, each top-level subset sum must be less than or equal to $2(b - 1)$, so that it could be partitioned into two subsets, each with sums less than b. For each such top-level partition, the subset with fewer integers is optimally partitioned two ways, and if both resulting subset sums are less that b, then the other subset is partitioned two ways. If m is the larger subset sum of the first two final subsets, the other top-level subset is only partitioned into two subsets both of whose sums are less than or equal to m if possible, rather than optimally partitioning it two ways.

For five-way partitioning, RNP partitions all the integers at the top level into two subsets, with the first subpartitioned two ways and the second subpartitioned three ways. Six-way partitioning divides the integers into two subsets at the top level, each of which are then subpartitioned three ways, etc. In other words, while (Moffitt 2013) generates the final subsets sequentially, RNP recursively decomposes the original integers into the final subsets in a balanced fashion. Sequential partitioning was implemented in (Korf 2009), but then replaced with balanced recursive partitioning (Korf 2011).

Sequentially generating the final subsets is more efficient than the balanced recursive decomposition of RNP, and the difference increases with increasing k. This can be seen by comparing our results in Tables 1 and 2 with those in Table 4 of (Korf and Schreiber 2013). If we plot the distribution of the sums of all subsets of a set of integers, we get a normal or bell-shaped curve. The most common subset sums are those closest to half the sum of all the integers. For larger values of k, the subsets whose sums are closest to s/k, where s is the sum of all the integers, are much less common. Since there are fewer of them, it is more efficient to generate these individual subsets, than to recursively partition the numbers in a balanced way. This is the primary reason for the dramatic performance improvements shown in (Moffitt 2013), compared to RNP, for seven or more subsets.

Sequential Number Partitioning (SNP)

Based on this observation, we propose a new algorithm, called *sequential number partitioning* (SNP). For k-way partitioning, SNP generates all first subsets with sums within the lower and upper bounds described above, and then for each, recursively partitions the remaining numbers $k - 1$ ways, as in (Moffitt 2013) and (Korf 2009). As soon as SNP finds a recursive partition whose largest sum is no greater than m, the largest sum of the completed subsets in the current solution, it returns, and otherwise it optimally partitions

k	2-Way			3-Way			4-Way			5-Way			6-Way		
n	IE	SS	Ratio	Mof	SNP	Ratio	Mof	SNP	Ratio	Mof	SNP	Ratio	Mof	SNP	Ratio
30	.162	.002	106	.036	.002	19.5	.019	.004	5.40	.014	.008	1.82	.013	.017	.776
31	.304	.002	133	.061	.003	22.0	.031	.006	5.59	.023	.012	2.02	.019	.023	.822
32	.590	.003	175	.114	.004	30.9	.055	.007	7.66	.039	.016	2.42	.034	.036	.962
33	1.13	.005	215	.202	.006	33.3	.092	.012	7.93	.064	.025	2.51	.053	.053	1.00
34	2.15	.007	298	.364	.008	48.3	.154	.015	10.1	.107	.034	3.11	.088	.075	1.17
35	4.13	.011	389	.649	.012	54.1	.264	.024	11.3	.171	.051	3.36	.143	.113	1.27
36	7.88	.016	494	1.14	.015	76.9	.453	.032	14.3	.279	.071	3.96	.224	.159	1.41
37	15.6	.024	649	2.08	.025	83.1	.784	.051	15.3	.475	.114	4.18	.364	.257	1.42
38	29.9	.034	885	3.80	.031	121	1.32	.067	19.6	.740	.150	4.93	.583	.362	1.61
39	56.1	.049	115	6.76	.051	134	2.30	.110	20.9	1.27	.240	5.28	.960	.574	1.67
40	112	.074	1524	12.6	.063	200	4.07	.143	28.5	2.04	.334	6.10	1.54	.829	1.86
41	213	.110	1937	22.7	.110	206	6.63	.227	29.2	3.56	.536	6.63	2.52	1.27	1.98
42	415	.156	2658	41.6	.134	310	11.5	.295	38.9	6.33	.755	8.38	4.44	1.86	2.38
43	779	.221	3533	74.8	.222	336	19.0	.485	39.2	9.63	1.13	8.52	7.12	2.89	2.47
44	1542	.339	4556	139	.274	506	34.2	.644	53.2	16.8	1.67	10.0	11.6	4.24	2.74
45		.490		254	.472	538	57.1	1.02	55.9	28.9	2.63	11.0	19.5	6.35	3.07
46		.699		460	.557	825	105	1.41	73.9	49.9	3.76	13.3	32.3	9.38	3.45
47		.962		851	.938	908	182	2.25	81.0	84.0	5.99	14.0	54.5	15.4	3.54
48		1.45			1.14		303	2.93	104	139	8.13	17.1	88.0	22.0	3.99
49		1.93			1.99		539	4.66	116	233	12.8	18.2	145	33.5	4.32
50		2.29			2.35		956	6.61	145	397	18.2	21.8	244	49.5	4.93

Table 1: Average Time in Seconds to Optimally Partition Uniform Random 48-bit Integers 2, 3, 4, 5, and 6 Ways

the remaining numbers, if there is a partition whose largest subset is less than b, the largest subset of the best complete solution found so far. The main difference between SNP and (Moffitt 2013) and (Korf 2009) is that SNP uses the extended Schroeppel and Shamir algorithm (ESS) to generate the subsets, rather than inclusion-exclusion (IE). Since ESS is less efficient for small n, SNP uses the complete greedy algorithm (CGA) for recursive calls with small n and $k > 2$, and the complete Karmarkar Karp algorithm (CKK) for two-way partitioning of 11 or fewer integers.

Dominance Pruning

(Moffitt 2013) also adds *dominance pruning*. Dominance pruning was first introduced by (Martello and Toth 1990a; 1990b) for bin packing, and used in (Korf 2002; 2003), but was not previously used for number partitioning. A given bin packing need not be considered as part of a solution if it is dominated by another packing of the same bin that always leads to a solution at least as good. The simplest example involves two integers x and y that sum to exactly the bin capacity. In that case, there always is an optimal solution with x and y in the same bin, and we can reject any solutions that put them in different bins. The reason is that given a complete bin containing x but not y, all other integers in the bin with x can be swapped with y, bringing x and y into the same bin, without increasing the total number of bins.

The general case of dominance for number partitioning is as follows: In any completed subset, if any single excluded integer can be swapped for any set of included integers whose sum is less than or equal to the excluded integer, without exceeding the available bin capacity, then the subset is dominated, and need not be considered. We explain below what we mean by "available bin capacity".

(Moffitt 2013) implements two special cases of this general rule. First, if an integer x is excluded, and including it along with the larger integers included in the subset would not raise the subset sum to b or more, the largest subset sum in the best solution found so far, the subset must include a set of smaller integers whose sum exceeds x. This is done by raising the lower bound on the subset sum, if necessary. Second, any excluded number whose inclusion would not raise the subset sum beyond m, the largest subset sum of the completed subsets in the current solution, must be included.

SNP also uses dominance pruning, but in a slightly different way. The first rule described above from (Moffitt 2013) is not valid in SNP, because the subsets are generated in a different order. Inclusion-exclusion always includes an integer before excluding it, and assigns integers in decreasing order. ESS generates subsets in a different and more complex order. ESS implements the first rule above, but using m instead of b as the bin capacity. It implements the second rule the same way. In addition, if any integer excluded from a subset can be swapped with the next smaller included integer without exceeding m, the subset is pruned as well. Full dominance pruning, which requires generating all subsets of included numbers, was too expensive to be worthwhile.

Empirical Performance Comparison

We empirically compared the performance of the (Moffitt 2013) algorithm with sequential number partitioning (SNP), for two through ten-way partitioning. We used 48-bit integers uniformly distributed between 0 and $2^{48} - 1$ to gen-

k	7-Way			8-Way			9-Way			10-Way		
n	Mof	SNP	Ratio	Mof	SNP	Ratio	Mof	SNP	Ratio	Mof	SNP	Ratio
30	.012	.031	2.53	.013	.054	4.05	.008	.070	9.09	.016	.217	13.6
31	.017	.044	2.56	.018	.074	4.09	.016	.101	6.20	.013	.275	21.9
32	.032	.069	2.18	.029	.116	4.02	.022	.145	6.69	.018	.210	12.0
33	.050	.105	2.10	.051	.202	3.96	.045	.294	6.48	.040	.377	9.43
34	.076	.155	2.04	.071	.293	4.11	.076	.462	6.07	.064	.570	8.94
35	.124	.233	1.88	.129	.476	3.70	.115	.737	6.43	.125	.979	7.83
36	.218	.375	1.72	.203	.729	3.60	.181	1.15	6.35	.207	1.80	8.60
37	.368	.625	1.70	.320	1.14	3.58	.293	1.89	6.47	.344	2.90	8.41
38	.521	.787	1.51	.510	1.82	3.56	.522	3.06	5.85	.467	4.24	9.08
39	.835	1.16	1.38	.779	2.55	3.27	.751	4.46	5.93	.708	6.88	9.72
40	1.36	1.85	1.63	1.41	4.27	3.04	1.24	7.74	6.22	1.13	12.1	10.7
41	2.18	2.82	1.29	2.08	6.44	3.09	1.90	11.8	6.21	1.78	18.9	10.6
42	3.49	3.97	1.14	3.44	9.49	2.76	2.97	17.9	6.04	2.94	32.4	11.0
43	5.67	6.97	1.23	5.66	15.5	2.73	5.31	30.5	5.75	5.10	54.7	10.7
44	9.70	10.5	1.08	8.44	22.4	2.65	9.05	51.0	5.63	7.63	78.5	10.3
45	15.6	15.2	.975	15.2	40.1	2.64	13.2	73.5	5.56	13.1	147	11.2
46	24.6	22.9	.932	23.3	53.7	2.31	22.0	115	5.23	23.6	252	10.7
47	41.3	36.2	.877	36.1	84.3	2.34	33.8	181	5.34	36.1	391	10.8
48	66.8	53.8	.805	57.9	132	2.28	62.5	332	5.32	58.3	556	9.54
49	110	81.2	.740	98.7	208	2.11	85.0	440	5.18	88.5	909	10.3
50	182	128	.704	147	289	1.96	155	765	4.94	143	1551	10.8

Table 2: Average Time in Seconds to Optimally Partition Uniform Random 48-bit Integers 7, 8, 9, and 10 Ways

erate hard instances without perfect partitions. Table one shows the results for two through six-way partitioning, with n ranging from 30 to 50 from the top to bottom row. The two-way data was already discussed. In each group of three columns, the first two show the running times in seconds to optimally partition n integers, averaged over 100 different instances, for the (Moffitt 2013) algorithm and SNP, respectively. The third column in each group shows the running time of the (Moffitt 2013) algorithm divided by the running time of SNP. This is how many times faster SNP is than (Moffitt 2013). The ratios are based on higher-precision data than that displayed in the table.

In each case except for 6-way partitioning with $n \leq 32$, SNP is faster than (Moffitt 2013). Furthermore, for a given number of subsets, the ratios of the running times increase monotonically with increasing n, strongly suggesting that SNP is asymptotically faster than (Moffitt 2013). For three-way partitioning, we see speedups of over 900 times for $n = 47$. For 4-way partitioning, SNP is over 140 times faster than (Moffitt 2013) for $n = 50$. For 5-way partitioning, it is over 20 times faster for $n = 50$. For 6-way partitioning, SNP is slower than (Moffitt 2013) for n up through 32, but SNP is faster for larger values of n, and almost five times faster by $n = 50$. This transition occurs at $n = 15$ for three-way partitioning, $n = 19$ for four-way partitioning, and $n = 25$ for five-way partitioning. The speedup of SNP over (Moffitt 2013) decreases with larger numbers of subsets, however.

Table 2 shows data in the same format as Table 1, but for 7, 8, 9, and 10-way partitioning. Unlike Table 1, the ratios presented here are the running times of SNP divided by the running times of (Moffitt 2013), indicating how many times faster (Moffitt 2013) is than SNP. For 7-way partitioning, (Moffitt 2013) is faster for n up through 44, but SNP is faster for larger n. For eight-way partitioning, (Moffitt 2013) is about four times faster then SNP for $n = 30$, but decreases to about two times faster by $n = 50$. We conjecture that for some value of $n > 50$, SNP will be faster than (Moffitt 2013). For nine-way partitioning, we also see a small decrease in the ratio of the running times with increasing n, but (Moffitt 2013) is almost five times faster than SNP at $n = 50$. We see a similar pattern for ten-way partitioning, with (Moffitt 2013) about ten times faster than SNP. We are still running SNP to partition 49 and 50 integers ten ways.

A Hybrid Algorithm

As shown in these tables, the fastest partitioning algorithm depends on the number of subsets k and the number of integers n, and the performance differences between them can be orders of magnitude. In particular, (Moffitt 2013) is faster for smaller values of n, and SNP is faster for larger values of n. The crossover point depends on k, and increases with increasing k. This suggests a hybrid algorithm in which each top-level or recursive call runs the algorithm that is fastest for the particular values of k and n. For two-way partitioning, the complete Karmarkar-Karp (CKK) is used for $n \leq 11$, and SS is used for larger values of n. For three-way partitioning, the complete greedy algorithm (CGA) is used for $n \leq 10$, and SNP is used for larger values of n. For all larger values of k, (Moffitt 2013) is used for smaller values of n, and SNP is used for larger values. The crossover values were all determined experimentally.

We implemented this algorithm, and its running times are within one percent of the faster running times shown in Tables 1 and 2, as a function of n and k. Since this is the performance one would expect simply by choosing the better algorithm at the top level, why doesn't the full hybrid algorithm perform better? The reason is that because of the crossover points, most recursive calls are executed by the same algorithm chosen at the top level. For example, if the top-level call is made to SNP, most of the recursive calls are made to SNP, and similarly for the (Moffitt 2013) algorithm.

Relation to Bin Packing

Number partitioning in closely related to another simple NP-complete problem called bin packing. In bin-packing, we are given a set of n integers, and a fixed bin capacity c. We want to pack each integer into a bin so that the sum of the integers packed into each bin does not exceed the capacity c, while minimizing the number of bins used. The only difference between number partitioning and bin packing is that number partitioning fixes the number of subsets or bins, and minimizes the needed capacity, while bin packing fixes the capacity and minimizes the number of bins.

While all NP-complete problems are polynomially reducible to each other, the close relationship between number partitioning and bin-packing means that algorithms for one are often directly applicable to the other. For example, the usual way to solve a bin-packing problem is to first run an polynomial-time approximation algorithm, such as best-fit decreasing, to get an upper bound on the number of bins needed. Then reduce the number of bins by one with each iteration, until all the items can no longer be packed into the given number of bins, or a lower bound on the number of bins is reached, such as the sum of all the integers, divided by the number of bins, rounded up. Almost all bin-packing instances can be solved this way in just a few iterations. Since each of these iterations has a fixed number of bins, they can be solved by a number partitioning algorithm, where the bin capacity is fixed, and which terminates when it either succeeds or fails to achieve a partition whose largest subset sum doesn't exceed the bin capacity.

Similarly, any bin-packing algorithm can be used to optimally solve number partitioning as follows. First, run a polynomial-time heuristic, such as the Karmarkar-Karp heuristic, to get an upper bound on the maximum subset sum, and also compute a lower bound on the maximum subset sum, such as the sum of all the integers divided by the number of subsets and rounded up. Then perform a binary search over this range of subset sums, where each probe runs a bin-packing algorithm with a fixed capacity. If the number of bins needed exceeds k, the number of subsets, then increase the bin capacity on the next probe. Otherwise, decrease the bin capacity on the next probe. We can terminate each bin-packing probe when we either achieve a feasible packing with k bins, or determine that it isn't feasible.

This is the strategy used in the operations research community to solve number partitioning (Coffman, Garey, and Johnson 1978; Dell'Amico et al. 2008), which they call "identical parallel machine scheduling", following the application presented at the beginning of this paper. In our experiments with this approach, it is much slower than our hybrid algorithm for two through nine-way partitioning, but performs about the same for ten-way partitioning. The difference in performance between the two methods decreases with increasing numbers of subsets, suggesting that the bin-packing approach may be faster with more than ten subsets.

Conclusions

The previous state of the art for optimally partitioning integers up to ten ways is represented by two different lines of research, published independently. We compare the two approaches, and introduce a new algorithm, sequential number partitioning (SNP), that generates subsets sequentially, but uses an extension of the Schroeppel and Shamir algorithm (ESS) to generate the individual subsets. For large n, SNP is much faster than (Moffitt 2013), but for small n, (Moffitt 2013) is faster. We also propose a hybrid combination of these two algorithms that achieves the performance of the faster algorithm, as a function of n and k.

Number partitioning and bin packing are closely related, and we consider them different variations of the same problem. In the operations research community, number partitioning is solved by repeated applications of bin packing, performing a binary search over a range of bin capacities. In our experiments, this approach is much slower than our hybrid algorithm for up to nine subsets, but perfoms comparably for ten subsets, and may be faster with more subsets.

All the algorithms discussed here are anytime algorithms, meaning that they return an approximate solution almost immediately, and then continue to find better solutions, eventually finding, and still later verifying, an optimal solution. Thus, on problems that are too large to be solved optimally, they can run for as long as time is available, returning the best solution found in that time.

References

Coffman, E. J.; 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.; Iori, M.; Martello, S.; and Monaci, M. 2008. Heuristic and exact algorithms for the identical parallel machine scheduling problem. *INFORMS Journal on Computing* 20(23):333–344.

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

Howgrave-Graham, N., and Joux, A. 2010. New generic algorithms for hard knapsacks. In *Proceedings of EUROCRYPT 2010*, 235–256. LNCS 6110.

Karmarkar, N., and Karp, R. M. 1982. The differencing method of set partitioning. Technical Report UCB/CSD 82/113, C.S. Division, University of California, Berkeley.

Korf, R. E., and Schreiber, E. L. 2013. Optimally scheduling small numbers of identical parallel machines. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-13)*, 144–152.

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

Korf, R. E. 2002. A new algorithm for optimal bin packing. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-02)*, 731–736.

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

Korf, R. E. 2009. Multi-way number partitioning. In *Proceedings of the 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 International Joint Conference on Artificial Intelligence (IJCAI-11)*, 591–596.

Martello, S., and Toth, P. 1990a. Bin-packing problem. In *Knapsack Problems: Algorithms and Computer Implementations*. Wiley. chapter 8, 221–245.

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

Moffitt, M. D. 2013. Search strategies for optimal multi-way number partitioning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-13)*, 623–629.

Schroeppel, R., and Shamir, A. 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.

Walsh, T. 2009. Where are the really hard manipulation problems? The phase transition in manipulating the veto rule. In *Proceedings of the International Joint Confercne on Artificial Intelligence (IJCAI-09)*, 324–329.