A Hybrid Recursive Multi-Way Number Partitioning Algorithm

Richard E. Korf
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
korf@cs.ucla.edu

Abstract

The number partitioning problem is to divide a given set of n positive integers into k subsets, so that the sum of the numbers in each subset are as nearly equal as possible. While effective algorithms for two-way partitioning exist, multi-way partitioning is much more challenging. We introduce an improved algorithm for optimal multi-way partitioning, by combining several existing algorithms with some new extensions. We test our algorithm for partitioning 31-bit integers from three to ten ways, and demonstrate orders of magnitude speedup over the previous state of the art.

1 Introduction and Applications

Given a multiset of positive integers, the number-partitioning problem is to divide them into a set of mutually exclusive and collectively exhaustive subsets so that the sum of the numbers in each subset are as nearly equal as possible. For example, given the integers {8,7,6,5,4}, if we divide them into the subsets {8,7} and {6,5,4}, the sum of the numbers in each subset is 15. This is optimal, and in fact a *perfect partition*. If the sum of the numbers is not divisible by the number of subsets, the subset sums in a perfect partition will differ by one. We focus here on optimal solutions, but the algorithms we discuss here are anytime algorithms, returning solutions immediately, and finding better solutions as they continue to run.

Number partitioning is perhaps the simplest NP-complete problem to describe. One application is multi-processor scheduling [Garey and Johnson, 1979]. Given a set of jobs, each with an associated completion time, and two or more identical processors, assign each job to a processor to complete all the jobs as soon as possible. Another application of number partitioning is in voting manipulation [Walsh, 2009].

There are three natural objective functions for number partitioning: 1) minimizing the largest subset sum, 2) maximizing the smallest subset sum, and 3) minimizing the difference between the largest and smallest subset sums. For two-way partitioning, all these objective functions are equivalent, but for multi-way partitioning, no two of them are equivalent [Korf, 2010]. We choose to minimize the largest subset sum, which corresponds to minimizing the total time in a scheduling application. An analogous set of algorithms can be used to maximize the smallest subset sum, which is the objective function for the voting manipulation application.

Minimizing the largest subset sum also allows our number-partitioning algorithms to be directly applied to bin packing. In bin packing, each of a set of numbers is assigned to a bin of fixed capacity, so that the sum of the numbers in each bin do not exceed the bin capacity, while minimizing the number of bins used. In practice, heuristic approximations for bin packing, such as best-fit decreasing, use only a few more bins than a simple lower bound, such as the sum of all numbers divided by the bin capacity. Thus, an effective bin-packing strategy is to allocate a fixed number of bins, and then to iteratively reduce the number of bins until a solution is no longer possible, or a lower bound is reached. Our number-partitioning algorithms are branch-and-bound algorithms that keep track of the largest subset sum in the best solution found so far. Thus, we could solve a bin-packing problem with a fixed number of k bins by partitioning the numbers into k subsets with a maximum subset sum equal to the bin capacity.

2 Prior Work

First we describe existing algorithms for number partitioning, both to establish the state of the art, and because our new algorithm incorporates most of them as components.

2.1 Complete Greedy Algorithm (CGA)

The greedy heuristic for this problem sorts the numbers in decreasing order, and then assigns each number in turn to a subset with the smallest sum so far. This heuristic can be extended to a Complete Greedy Algorithm (CGA) [Korf, 1998] by sorting the numbers in decreasing order, and searching a tree, where each level corresponds to a different number, and each branch assigns that number to a different subset. To avoid duplicate solutions that differ only by a permutation of the subsets, a number is never assigned to more than one empty subset. We keep track of the largest subset sum in the current best solution, and if the assignment of a number to a subset causes its sum to equal or exceed the current bound, that assignment is pruned. If a perfect partition is found, or one in which the largest subset sum equals the largest number, the search returns it immediately. Without a perfect partition, the asymptotic time complexity of CGA is only slightly better than $O(k^n)$, but it has very low overhead per assignment.

2.2 Karmarkar-Karp Heuristic (KK)

The Karmarkar-Karp [Karmarkar and Karp, 1982] (KK) heuristic is a polynomial-time approximation algorithm for this problem. It applies to any number of subsets, but is simplest for two-way partitioning. It sorts the numbers in decreasing order, then replaces the two largest numbers in the sorted order by their difference. This is equivalent to separating them in different subsets. This is repeated until only one number is left, which is the difference between the final two subset sums. Adding the difference to the sum of all the numbers, then dividing by two, yields the larger subset sum.

2.3 Complete-Karmarkar-Karp (CKK) Algorithm

The KK heuristic can be extended to a "Complete Karmarkar-Karp" (CKK) algorithm [Korf, 1998]. We explain the two-way version here, but it also applies to multi-way partitioning. While the KK heuristic separates the two largest numbers in different subsets, the only other option is to assign them to the same subset, by replacing them by their sum. CKK searches a binary tree where the left branch of a node replaces the two largest numbers by their difference, and the right branch replaces them by their sum. If the largest number equals or exceeds the sum of the remaining numbers, they are all placed in a separate subset. The first solution found is the KK solution, but CKK eventually finds and verifies an optimal solution.

2.4 Subset Sum Problem

The subset sum problem is to find a subset of a given set of numbers whose sum is closest to a given target value. Two-way partitioning is equivalent to a subset sum problem where the target value is half the sum of all the numbers. We next present two algorithms for the subset sum problem.

Horowitz and Sahni (HS)

The Horowitz and Sahni (HS) algorithm [Horowitz and Sahni, 1974] divides the n numbers into two half sets of size $n/2$, generates all possible subsets from each half, including the empty set, and sorts them in increasing order of their sums. Any subset of the original numbers must consist of a subset from the first half plus a subset from the second half.

The algorithm maintains two pointers, one into each sorted list of subset sums. The *first* pointer starts at the smallest sum in the first list, and the *second* pointer starts at the largest sum in the second list. It adds the two sums pointed to. If the sum equals the target value, the search terminates. If the sum is less than the target, the *first* pointer is incremented by one position. If the sum is greater than the target, the *second* pointer is decremented by one position. The algorithm increments the *first* pointer and decrements the *second* pointer, until the target value is found, or either pointer exhausts its list, and returns the subset sum closest to the target value.

The running time of HS is $O(n2^{n/2})$, due to the time to sort the lists of subsets. Its main drawback is the space to store these lists, each of length $2^{n/2}$.

Schroeppel and Shamir (SS)

The Schroeppel and Shamir (SS) algorithm [Schroeppel and Shamir, 1981] uses less memory than HS. HS accesses the two lists of subset sums in sorted order. Rather than precomputing and sorting these lists, SS generates them in order.

To generate all subset sums from the first half list in order, SS divides the list into two lists a and b, each of size $n/4$, generates all subsets from each list, and sorts them in increasing order of their sums. Then, a min heap is constructed of pairs of subsets from a and b. Initially, the heap includes all pairs of the form (a_1, b_j), with (a_1, b_1) on top. As each subset from the combined list of a and b is needed, it is removed from the top of the heap. This element (a_i, b_j) is replaced in the heap with the pair (a_{i+1}, b_j). This generates all pairs of subsets from a and b in increasing order of their total sum.

Similarly, all subsets from the second half list of numbers are generated in decreasing order of their sum using a max heap. We then use the HS scheme to generate the subset whose sum is closest to the target value. The running time of SS is $O(n2^{n/2})$, and the memory required is $O(2^{n/4})$, making time the limiting resource. With three gigabytes of memory, SS can solve problems of up to about 100 numbers, including space to store the subsets and their sums. In addition to using less memory, SS often runs faster than HS.

2.5 Recursive Number Partitioning (RNP)

The previous best algorithm for multi-way partitioning is Recursive Number Partitioning (RNP) [Korf, 2009]. RNP first runs the KK heuristic to generate an initial solution. If k is odd, it generates all first subsets that could be part of a k-way partition better than the current best. For each, it optimally partitions the remaining numbers $k-1$ ways. If k is even, it generates all possible two-way partitions of the numbers, such that subpartitioning each subset $k/2$ ways could result in a k-way partition better than the current best. It then optimally partitions the first subset $k/2$ ways, and if the resulting maximum subset sum is less than that of the current best solution, it optimally partitions the second subset $k/2$ ways.

To generate the first subsets for odd k, or to partition the numbers two ways for even k, RNP sorts the numbers in decreasing order, and then searches an inclusion-exclusion binary tree. Each level in the tree corresponds to a different number, and at each node one branch includes the number in the subset, and the other branch excludes it. Pruning is performed to eliminate subsets whose sum would fall outside of specified bounds, to be discussed below. For optimal two-way partitioning, RNP uses the CKK algorithm.

3 Improved Recursive Number Partitioning

We now present our new algorithm, which we call Improved Recursive Number Partitioning (IRNP). It takes a list of n integers, and returns an optimal partition into k subsets. Like RNP, IRNP is an anytime branch-and-bound algorithm that first computes the KK approximation, then continues to find better solutions until it finds and verifies an optimal solution. If it finds a perfect partition, or one whose maximum subset sum equals the largest number, it returns it immediately.

3.1 Greedy is Optimal for $n \leq k+2$

It is easy to show that if $n \leq k+2$, the greedy heuristic returns optimal solutions, and is used in these cases. For larger n, there are instances where neither greedy nor KK are optimal. A proof of this is available from the author upon request.

3.2 Partitioning Small Sets of Numbers

For optimal two-way partitioning for $n \geq 5$, we use CKK for $n \leq 16$, and SS for $n > 16$. This is because CKK runs in $O(2^n)$ time, and SS runs in $O(n2^{n/2})$ time, but the constant overhead of SS is much greater than for CKK. For partitioning more than two ways, CKK is much less efficient than CGA. Thus, for $k > 2$ and small values of $n > k + 2$, we use CGA to compute optimal partitions. Table 1 shows the largest values of n for which CKK ($n = 2$) or CGA ($n > 2$) are faster than IRNP. They were determined experimentally.

k	2	3	4	5	6	7	8	9	10
n	16	12	14	16	19	21	25	27	31

Table 1: Largest n for Which CKK or CGA is Fastest

3.3 Recursive Principle of Optimality

For larger values of n, our algorithm relies on a principle of optimality, which we define and prove here. First, we define a *decomposition* of a partition as a two-way partition of the subsets of the partition. For example, there are three different decompositions of a three-way partition, one for each division into one set vs. the other two sets. Similarly, there are seven different decompositions of a four-way partition, four of which decompose it into one set vs. the other three, and three of which decompose it into two sets vs. two sets.

Given this definition, and an objective function that minimizes the largest subset sum in a partition, we can state and prove the following theorem: Given any optimal k-way partition, and any decomposition of that partition into k_1 sets and k_2 sets such that $k_1 + k_2 = k$, then any optimal partition of the numbers in the k_1 sets into k_1 sets, combined with any optimal partition of the numbers in the k_2 sets into k_2 sets, results in an optimal k-way partition.

The proof of this theorem is very simple. Given a collection of k_1 subsets, optimally partitioning the numbers in those subsets k_1 ways can only leave their maximum subset sum the same, or reduce it, but it cannot increase it. Thus, given an optimal k-way partition, optimally partitioning the numbers in the k_1 subsets and the k_2 subsets in k_1 ways and k_2 ways, respectively, cannot increase the maximum subset sum. Thus, the new k-way partition must also be optimal.

The value of this principle, for example, is that given any two-way partition of all the numbers, we can compute the best four-way partition that respects that two-way partition by optimally subpartitioning each of the two subsets two ways.

This principle of optimality is also valid for the objective function that maximizes the smallest subset sum, by a completely symmetric argument. It is not valid for the objective function that minimizes the difference between the largest and the smallest subset sum, however [Korf, 2010].

3.4 Generating all Subset Sums in a Given Range

For three-way partitioning of more than twelve numbers, like RNP, IRNP generates all subsets that could be part of a partition better than the current best, and for each it optimally partitions the remaining numbers two ways. This requires generating all subsets with sums in a given range, the bounds of which will be described below. While RNP generates these subsets by searching an inclusion-exclusion binary tree, IRNP uses a new algorithm which is an extension of the SS algorithm. This extension is necessary because SS only returns a single subset whose sum is closest to a specified target value. For simplicity, we first describe our extension to HS.

Extending Horowitz and Sahni (EHS)

HS maintains two pointers, a *first* pointer into the first list of subsets and a *second* pointer into the second list. In our extended HS algorithm (EHS), we replace the *second* pointer with two pointers into the second list, which we call *low* and *high*. At any given time, *low* is the largest index such that the sum of the subsets pointed to by *first* and *low* is less than or equal to the lower bound. Similarly, *high* is the smallest index such that the sum of the subsets pointed to by *first* and *high* is greater than or equal to the upper bound. *First* is incremented, and *low* and *high* are decremented to maintain this invariant, thereby generating all subsets whose sums lie within the lower and upper bounds.

Extending Schroeppel and Shamir (ESS)

Extending SS to generate all subsets whose sums are within a given range is a little more complex. The top of the min heap corresponds to the subset pointed to by the *first* pointer in HS, and the top of the max heap corresponds to the subset pointed to by the *second* pointer. To extend this to a range of subsets, we explicitly store on a separate list those subsets whose sums fall between those pointed to by the *low* and *high* pointers in EHS. The first subset on this list corresponds to the subset pointed to by the *low* pointer in EHS, and the last subset corresponds to the subset pointed to by the *high* pointer in EHS. At any given time, the subsets whose sums are within the bounds are the subsets on this list, combined with the subset at the top of the min heap. As we pop subsets off the min heap, corresponding to incrementing the *first* pointer in EHS, we add subsets with smaller sums to the list by popping them off the max heap, corresponding to decrementing the *low* pointer, and discard subsets from the list whose sums exceed the upper bound when combined with that on top of the min heap, corresponding to decrementing the *high* pointer. This list of subsets is implemented as a circular buffer with pointers to the head and tail, corresponding to the *low* and *high* pointers of EHS. This modification increases the space needed by SS, but in all our experiments, a list of a hundred thousand subsets was more than sufficient.

3.5 Tighter Bounds on Subset Sums

ESS generates all subsets whose sums are within specified bounds. For simplicity we describe those bounds for three-way partitioning, but they generalize in a straightforward way to k-way partitioning. Let m be the maximum subset sum in the current best solution, and let s be the sum of all the numbers. Each subset sum must be less than m, and thus $m - 1$ is an upper bound on all subsets. The first subset sum must also be large enough that the remaining numbers can be partitioned into two subsets with both sums less than m. Thus, $s - 2(m - 1)$ is a lower bound on the first subset.

We can tighten these bounds by eliminating duplicate partitions that differ only by a permutation of the subsets. In any three-way partition, there must be at least one subset whose sum is less than or equal to $\lfloor n/3 \rfloor$. Thus, RNP uses a lower bound of $s - 2(m-1)$, and an upper bound of $\lfloor n/3 \rfloor$ for the first subset in a three-way partition.

IRNP tightens these bounds even further. In any three-way partition there must be at least one subset sum greater than or equal to $\lceil n/3 \rceil$. IRNP uses a lower bound of $\lceil n/3 \rceil$ and an upper bound of $m-1$ on the first subset. For three-way partitioning, the difference between the lower and upper bounds of IRNP is half that of RNP, leading to the generation of about half as many first subsets as RNP.

3.6 Top-Level Decomposition

For even values of k and values of n greater than those in Table 1, IRNP generates all two-way decompositions that could lead to a k-way partition better than the current best. This is done by using ESS to generate a subset with a lower bound of $\lceil s/(k/2) \rceil$ on its sum and an upper bound of $(k/2)(m-1)$, and then optimally partitioning this subset and its complement $k/2$ ways. As a heuristic, the subset with fewer numbers is partitioned first, and only if all its subset sums are less than m is the subset with more numbers partitioned. RNP uses the same top-level decomposition for even k, but a different set of bounds, with the same difference between them.

For odd values of k, RNP generates all first subsets that could lead to a better solution, and for each optimally partitions the remaining numbers $k-1$ ways. For example, for five-way partitioning, RNP generates single subsets and then optimally partitions the remaining numbers four ways. By contrast, for odd k, IRNP decomposes the numbers into two subsets, one of which is subpartitioned into $\lfloor k/2 \rfloor$ subsets, and the other is subpartitioned into $\lceil k/2 \rceil$ subsets. For five-way partitioning, for example, IRNP decomposes the numbers into a subset to be subpartitioned two ways, with its complement to be subpartitioned three ways. The lower bound on the sum of this first set is $\lfloor k/2 \rfloor (s/k)$, and the upper bound is $\lfloor k/2 \rfloor (m-1)$. While either strategy ultimately results in $k-1$ partitionings, the IRNP strategy is much more efficient for odd values of k greater than three. Given a top-level decomposition, it is usually more efficient to first subpartition the subset with the fewest numbers, even if it is partitioned more ways. Again, the larger subset is only partitioned if every subset of the smaller subset has a sum less than m.

4 How IRNP Differs from RNP

While both IRNP and RNP use recursive decomposition, there are significant differences between them. The first is that RNP is purely recursive for $k > 2$, and uses CKK for all optimal two-way partitions, while IRNP is a hybrid. For optimal two-way partitioning of small sets IRNP uses CKK, but for larger sets it uses SS. For partitioning more than two ways, IRNP uses CGA for small sets, and recursive decomposition for large sets. The second is that RNP searches an inclusion-exclusion tree to generate all subsets in a given range, while IRNP uses our extension of SS (ESS) for this. Both algorithms use the same top-level decomposition for even values of k, but they differ significantly for odd k. In particular, RNP generates first subsets and then recursively partitions the remaining numbers $k-1$ ways, while IRNP divides the numbers into subsets which are subpartitioned $\lfloor k/2 \rfloor$ and $\lceil k/2 \rceil$ ways. Finally, IRNP uses lower and upper bounds on the subset sums with a smaller range than those of RNP for odd k.

Another difference is that RNP was originally implemented to minimize the difference between the largest and smallest subset sums, while IRNP minimizes the largest subset sum. While our principle of optimality does not apply to the former objective function [Korf, 2010], it was easy to modify our RNP code to minimize the largest subset sum, allowing the direct experimental comparisons described below.

5 Experimental Results

Our experimental results are shown in Tables 2 and 3, which show three through six-way partitioning, and seven through ten-way partitioning, respectively. These are the first published results for optimal partitioning into more than five sets.

In each case, we compare IRNP to RNP. We originally wrote separate RNP programs for three, four, and five-way partitioning. In each case, we modified those programs to minimize the largest subset sum. For partitioning more than five ways, we implemented a general-purpose version of RNP, minimizing the largest subset sum.

The first column (n) shows the number of numbers partitioned. For each value of k, we show in three successive columns the running times for RNP, IRNP, and the ratio of the two. Each running time is the average of 100 random instances, and is displayed either as a decimal fraction of a second, or as hours:minutes:seconds. The ratios are based on the total running times for all 100 instances in microseconds.

In each case, we chose random numbers uniformly distributed from 0 to $2^{31}-1$. This is the precision of the standard random number generator in the Linux C library, and is equivalent to one second in over 68 years, in problems such as multiprocessor scheduling. The precision of the numbers only affects the performance of the algorithms when perfect partitions exist, since once one is found, both algorithms return it immediately. For the values of n that we ran, this only occurred with three and four-way partitioning.

For three-way partitioning, the running times of both algorithms increase with n, and then decrease. This is due to the presence of perfect partitions, near what is called the "phase transition". For $n = 44$, the hardest problems for RNP, IRNP runs almost 900 times faster. This is due to the tighter bounds discussed above, our ESS algorithm for generating first subsets, and SS for two-way partitioning of more than 16 numbers. IRNP solved these problems in less than a second on average. The increase in the running time of IRNP for problems larger than 48 is due to the fact that our implementation was not optimized for problems with perfect partitions.

For four-way partitioning, the running time of IRNP increases with n, reaching a peak at $n = 51$, again due to the appearance of perfect partitions at the phase transition. The ratio of the speed of IRNP to RNP increases monotonically, and should continue to increase until $n = 51$. For $n = 48$, IRNP is almost 300 times faster than RNP. This is due to ESS,

k	Three Way			Four Way			Five Way			Six Way		
n	RNP	IRNP	Ratio	RNP	IRNP	Ratio	RNP	IRNP	Ratio	RNP	IRNP	Ratio
25	.015	.001	13	.017	.004	5	.117	.013	9	.227	.160	1
26	.027	.002	16	.029	.006	5	.210	.019	9	.381	.237	2
27	.049	.002	21	.050	.009	6	.418	.030	13	.616	.367	2
28	.087	.004	24	.086	.014	6	:01	.044	16	:01	.567	2
29	.158	.005	34	.148	.021	7	:01	.064	23	:02	:01	2
30	.279	.007	39	.278	.036	8	:03	.099	30	:04	:01	3
31	.500	.009	54	.474	.052	9	:05	.150	35	:06	:02	3
32	.939	.015	62	.831	.081	10	:11	.244	45	:12	:04	3
33	:02	.020	83	:02	.132	12	:22	.374	58	:20	:06	3
34	:03	.030	99	:03	.209	14	:43	.645	66	:38	:10	4
35	:06	.039	140	:05	.289	18	1:44	:01	97	1:07	:15	4
36	:10	.064	159	:09	.443	21	3:28	:02	124	2:23	:26	5
37	:19	.084	222	:17	.668	26	8:25	:03	178	3:55	:42	6
38	:35	.128	272	:32	.979	33	14:31	:04	202	7:34	1:10	7
39	1:03	.167	376	1:00	:01	42	40:06	:08	318	13:43	1:48	8
40	1:55	.259	445	1:52	:02	49	1:31:09	:12	465	26:23	3:10	8
41	3:00	.309	582	3:31	:04	59	2:45:37	:17	585	47:49	5:10	9
42	4:40	.428	655	6:29	:06	69	5:50:09	:30	695	1:30:26	8:27	11
43	6:23	.457	837	11:44	:08	91		:44			13:59	
44	6:32	.440	888	22:20	:12	116		1:07			22:47	
45	5:07	.410	748	42:18	:18	143		1:49			37:53	
46	1:15	.246	304	1:20:10	:27	175		2:48				
47	1:01	.304	199	2:32:19	:39	234		4:12				
48	:48	.304	158	4:43:31	:57	299		7:07				
49	:34	.395	85		1:10			10:56				
50	:25	.372	68		1:27			18:20				
51	:21	.630	33		1:42			27:46				
52	:16	.642	25		1:27			42:06				

Table 2: Average Time to Optimally Partition 31-bit Integers Three, Four, Five, and Six Ways

and SS for optimal two-way partitioning.

For five-way partitioning, we see a monotonic increase in running time, since perfect partitions don't appear in problems of this size. Here we see the effect of IRNP's top-level decomposition for odd $k > 3$, as well as its tighter bounds, and the use of ESS. The effect of SS for two-way partitioning diminishes with increasing k, since a smaller percentage of two-way partitions involve more than 16 numbers. For 42 numbers, IRNP is almost 700 times faster than RNP.

The smallest speedups we see are for six and eight-way partitioning. We eventually see a monotonic increase in the ratios, but the speedups are only one order of magnitude for six-way partitioning, and a factor of seven for eight-way partitioning. The reason is that in neither case does our new top-level decomposition come into play, nor is there much effect from SS, since most of the two-way subproblems have less than 17 numbers. Thus, most of the improvement is due to ESS, and the tighter bounds on the three-way subproblems generated by six-way partitioning. The observed speedups are limited by the time to run RNP on larger problems.

Table 3 shows an effect we don't see in Table 2. The speedup ratios start high, and then decrease, before increasing again. This is due to the use of CGA for partitioning small sets. For values of n above the horizontal lines, IRNP just calls CGA. The enormous speedup ratios shown, over eight million for ten-way partitioning of 21 values for example, indicate that CGA is dramatically more efficient than RNP for these problems. As shown in [Korf, 2009], however, CGA is many orders of magnitude slower than RNP for large values of n. CGA has a higher asymptotic complexity than RNP, but much lower constant factor overhead. This is the reason that the ratios initially decrease with increasing n. For seven and eight-way partitioning, we see these ratios eventually bottom out, and then increase almost monotonically with increasing n. For seven-way partitioning of 32 numbers, for example, IRNP is over 900 times faster than RNP. We see a similar pattern for nine-way partitioning and would expect it for ten-way partitioning as well.

6 Conclusions and Further Work

We presented an improved algorithm (IRNP) for multi-way number partitioning. It's an anytime algorithm that returns an approximate solution immediately, and continues to find better solutions until it finds and verifies an optimal solution. It uses several existing algorithms, including KK, CKK, CGA, and SS. It also uses a new algorithm (ESS) we developed which extends the SS algorithm to return all subsets within a specified set of bounds. We stated and proved a general principle of optimality upon which IRNP is based. IRNP outperforms RNP, the previous best algorithm for this problem,

n	Seven Way			Eight Way			Nine Way			Ten Way		
	RNP	IRNP	Ratio	RNP	IRNP	Ratio	RNP	IRNP	Ratio	RNP	IRNP	Ratio
20	:05	.005	969	:02	.001	2232	:59	.000	2957701	:57	.000	5675522
21	:06	.012	472	:03	.007	401	2:30	.001	283296	1:27	.000	8737230
22	:10	.038	253	:07	.020	356	6:20	.005	76928	6:25	.001	653132
23	:05	.062	76	:06	.068	88	12:11	.076	9562	9:50	.011	53964
24	:08	.131	61	:10	.339	39	25:27	.206	7425	23:18	.041	34199
25	:23	.235	97	:10	.937	10	50:32	.438	6928	32:51	.061	32478
26	:54	.435	123	:14	:02	7	55:01	:01	2387		:01	
27	1:45	.791	132	:12	:04	3	41:31	:05	507		:02	
28	4:12	:01	186	:23	:09	2	1:00:32	:11	332		:08	
29	10:07	:03	232	:34	:21	2	3:27:13	:46	272		:28	
30	22:25	:04	356	1:08	:37	2	5:50:21	:59	357		1:10	
31	34:06	:06	346	1:42	:48	2		1:43			11:33	
32	2:26:43	:09	941	3:38	1:16	3		7:01			15:17	
33		:16		6:05	2:07	3		7:13			40:07	
34		:36		11:59	3:13	4		10:53			1:07:59	
35		:42		21:54	5:15	4		15:18			1:53:57	
36		1:14		44:48	8:24	5		25:29				
37		1:56		1:26:27	13:40	6		38:18				
38		3:24		2:59:12	24:49	7		1:16:30				
39		5:11			42:36			2:03:13				
40		8:59			1:20:43							

Table 3: Average Time to Optimally Partition 31-bit Integers Seven, Eight, Nine, and Ten Ways

by orders of magnitude in practice. For three-way partitioning of 31-bit numbers, IRNP solves the hardest problem instances in less than a second on average. For four-way partitioning of 31-bit integers, it solves the hardest instances in less than two minutes on average. We have proven that for $n \leq k+2$, the greedy heuristic returns optimal solutions. We also presented the first data on optimally partitioning numbers more than five ways. One of the surprising results of this work is that for partitioning 31-bit integers into more than ten subsets, for problem sizes that can be solved optimally in reasonable time on current machines, the simplest optimal algorithm, CGA, may also be the fastest.

The precision of the numbers affects the existence of perfect partitions. For a given number of subsets, decreasing the precision increases the density of perfect partitions, making such problems easier to solve. In the absence of perfect partitions, however, the performance of all these algorithms is unaffected by the precision. Without perfect partitions, the speedup of IRNP compared to RNP appears to increase without bound, with the observed speedups limited only by the time available to run RNP on larger problems.

We also pointed out a connection between number partitioning and bin packing. In particular, any branch-and-bound algorithm for number partitioning that minimizes the largest subset sum can be directly applied to bin packing. In future work, we plan to study whether our number-partitioning algorithms can also improve the state of the art for bin packing.

Acknowledgments

This work was supported by NSF grant IIS-0713178. Thanks to Satish Gupta and IBM for the computer the experiments were run on, and thanks to Eli Gafni for suggesting a shorter proof that greedy is optimal for $n \leq k+2$.

References

[Garey and Johnson, 1979] 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.

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

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

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

[Korf, 2009] Korf, R. E. 2009. Multi-way number partitioning. In *Proceedings of IJCAI-09*, 538–543.

[Korf, 2010] Korf, R. E. 2010. Objective functions for multi-way number partitioning. In *Proceedings of the Symposium on Combinatorial Search (SOCS-10).*

[Schroeppel and Shamir, 1981] 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, 2009] Walsh, T. 2009. Where are the really hard manipulation problems? The phase transition in manipulating the veto rule. In *Proceedings of IJCAI-09*, 324–329.