From Approximate to Optimal Solutions: A Case Study of Number Partitioning

Richard E. Korf
Computer Science Department
University of California, Los Angeles
Los Angeles, Ca. 90024
korf@cs.ucla.edu

Abstract

Given a set of numbers, the two-way partitioning problem is to divide them into two subsets, so that the sum of the numbers in each subset are as nearly equal as possible. The problem is NP-complete, and is contained in many scheduling applications. Based on a polynomial-time heuristic due to Karmarkar and Karp, we present a new algorithm, called Complete Karmarkar Karp (CKK), that optimally solves the general number-partitioning problem. CKK significantly outperforms the best previously-known algorithms for this problem. By restricting the numbers to twelve significant digits, we can optimally solve two-way partitioning problems of arbitrary size in practice. CKK first returns the Karmarkar-Karp solution, then continues to find better solutions as time allows. Almost five orders of magnitude improvement in solution quality is obtained within a minute of running time. Rather than building a single solution one element at a time, CKK constructs subsolutions, and combines them in all possible ways. CKK is directly applicable to the 0/1 knapsack problem, since it can be reduced to number partitioning. This general approach may also be applicable to other NP-hard problems as well.

1 Introduction and Overview

Consider the following very simple scheduling problem. We are given two identical machines, a set of jobs, and the time required to process each job on either machine. Assign each job to one of the machines, in order to complete all the jobs in the shortest elapsed time. In other words, divide the job processing times into two subsets, so that the sum of the times in each subset are as nearly equal as possible. This is the two-way number partitioning problem, and is NP-complete[1]. The generalization to K-way partitioning with K machines is straightforward, with the cost function being the difference between the largest and smallest subset sums. This basic problem is likely to occur as a subproblem in many practical scheduling applications.

For example, consider the set of numbers $(4, 5, 6, 7, 8)$. If we divide it into the two subsets $(7, 8)$ and $(4, 5, 6)$, the sum of each subset is 15, and the difference of the subset sums is zero. In addition to being optimal, this is also a perfect partition. Note that if the sum of all the numbers is odd, a perfect partition will have a subset difference of one.

We first present previous work on this problem, including a polynomial-time greedy heuristic, and an exponential-time algorithm to find optimal solutions. We also present an elegant polynomial-time approximation algorithm due to Karmarkar and Karp, called set differencing[5], or the KK heuristic, which dramatically outperforms the greedy heuristic. The main contribution of this paper is to extend the KK heuristic to a complete algorithm, which we call Complete Karmarkar Karp (CKK). The first solution returned by CKK is the KK solution, and as the algorithm continues to run it finds better solutions, until it eventually finds and verifies an optimal solution.

We present experimental results comparing CKK to the standard algorithm for finding optimal solutions. CKK appears to be asymptotically faster than the standard algorithm, and provides orders of magnitude improvement when perfect partitions exist. Due to the existence of perfect partitions, it is possible in practice to optimally partition arbitrarily large sets of numbers, if the number of significant digits in each number is limited. This limit is about twelve decimal digits for two-way partitioning. This is not a limitation in practice, since no physical quantities are known with more than twelve digits of precision. For example, this would represent an accuracy of one second in over 30,000 years. We have performed the same experiments for three-way partitioning, with very similar results, except that the precision limit for optimally partitioning large sets is only six decimal digits.

CKK is the best existing algorithm for number partitioning, outperforming even stochastic approaches. It also can be directly applied to the 0/1 knapsack problem, which can be reduced to number partitioning. Instead of incrementally building a single partition, CKK constructs a large number of subpartitions, and combines them together in all possible ways. We believe that this general strategy may be applicable to other NP-complete problems as well.

2 Previous Work

2.1 Greedy Heuristic

The obvious greedy heuristic for this problem is to first sort the numbers in decreasing order, and arbitrarily place the largest number in one of two subsets. Each remaining number is then placed in the subset with the smaller total sum thus far, until all numbers are assigned.

For example, given the sorted numbers $(8,7,6,5,4)$, the greedy algorithm would proceed through the following states, where the numbers outside the list are the current subset sums: $8, 0(7,6,5,4)$, $8, 7(6,5,4)$, $8, 13(5,4)$, $13, 13(4)$, $13, 17()$, for a final subset difference of 4. Note that the greedy algorithm does not find the optimal solution in this case. The above notation maintains both subset sums, but to find the value of the final difference, we only need the difference of the two subset sums. Thus we can rewrite the above trace as: $8(7,6,5,4)$, $1(6,5,4)$, $5(5,4)$, $0(4)$, $4()$. In practice, however, we would keep track of the actual subsets as well.

This algorithm requires $O(N \log N)$ time to sort the N numbers, and then $O(N)$ time to assign them, for an overall time complexity of $O(N \log N)$.

2.2 Set Differencing (Karmarkar-Karp)

The set differencing method of Karmarkar and Karp[5], also known as the KK heuristic, is another polynomial-time approximation algorithm. It also begins by sorting the numbers in decreasing order. In our example, the two largest numbers are 8 and 7. The algorithm commits to placing these two numbers in different subsets, while deferring the decision about which subset each will go in. For example, if we place the 8 in the left subset, and the 7 in the right subset, this is equivalent to placing their difference of 1 in the left subset, since we can subtract 7 from both subsets without affecting the final difference. Similarly, placing the 8 in the right subset and the 7 in the left subset is equivalent to placing 1 in the right subset. The algorithm removes the two largest numbers, computes their difference of 1, and then treats the 1 just like any other number, inserting it in sorted order in the remaining list of numbers. The algorithm continues removing the two largest numbers, replacing them by their difference in the sorted list, until there is only one number left. This number represents the value of the final subset difference.

For example, given the sorted numbers $(8,7,6,5,4)$, the 8 and 7 are replaced by their difference of 1, which is inserted in the remaining list, resulting in $(6,5,4,1)$. Next, the 6 and 5 are replaced by their difference of 1, yielding $(4,1,1)$. The 4 and 1 are replaced by their difference of 3, giving $(3,1)$, and finally the difference of these last two numbers is the final subset difference of 2. The KK heuristic also fails to find the optimal partition in this case, but does better than the greedy heuristic.

While the above algorithm computes the final difference of the subset sums, computing the actual partition is slightly more involved. The algorithm builds a tree, initially with one node for each original number, and no edges. Each differencing operation adds an edge between two numbers, to signify that they must go in different subsets. The resulting graph forms a spanning tree of the original nodes, which is then two-colored to determine the actual subsets, with all the numbers of one color going in one subset.

For example, Figure 1 shows the final tree for the example above. First, replacing 8 and 7 by their difference creates an edge between them. The larger of the two, node 8, represents their difference of 1. Next, replacing 6 and 5 by their difference adds an edge between them, with node 6 representing their difference of 1. If we then take the difference of 4, and the 1 from the difference between 7 and 8, we add an edge between 4 and 8, since node 8 represents the difference of 1. Since 4 is larger than 1, node 4 represents their difference of 3. Finally, an edge is added between node 4 and node 6, which represents the remaining value of 1.

Figure 1: Tree from KK partitioning of (4,5,6,7,8)

In general, the resulting graph forms a spanning tree of the original nodes, since all the numbers must eventually be combined, and $N-1$ edges are created, one for each differencing operation. We then color the nodes of the graph with two colors, so that no adjacent nodes receive the same color, to get the final partition itself. To two-color a tree, color one node arbitrarily, treating this node as the root, and then color all the nodes at a given depth from the root the same color, alternating colors with each level. Two-coloring the above graph results in the subsets $(7,4,5)$, and $(8,6)$, whose subset sums are 16 and 14, respectively, for a final partition difference of 2.

The running time of this algorithm is $O(N \log N)$ to sort the N numbers, $O(N \log N)$ for the differencing, since each difference must be inserted into the sorted order, and finally $O(N)$ to two-color the graph, for an overall time complexity of $O(N \log N)$.

The KK heuristic finds much better solutions on average than the greedy heuristic. Figure 2 shows comparative data for the two algorithms, partitioning random integers uniformly distributed from 0 to 10 billion. The horizontal axis is the number of values in the original set, and the vertical axis is the difference of the final subset sums, on a logarithmic scale. Each data point is an average of 1000 random problem instances. As the number of values increases, the final difference found by the KK heuristic is orders of magnitude smaller than for the greedy heuristic. We also show the optimal solution quality, which is orders of magnitude better than even the KK solution in this range. The optimal solution data points are averages of only 100 problem instances each. With 40 or more numbers of this size, a perfect partition difference of zero or one was found in every case. If we extend the graph further to the right, at about 300 numbers the KK line joins the optimal line, finding a perfect partition almost every time. The greedy line, however remains almost flat, requiring a thousand numbers to drop another order of magnitude in solution quality.

The explanation for the difference between the qual-

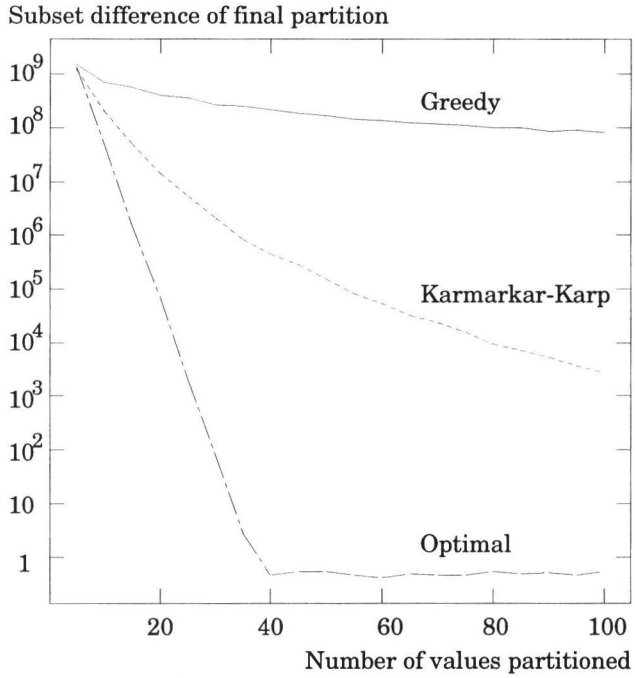

Figure 2: Greedy vs. KK vs. optimal solution quality

ity of the greedy and KK solutions is quite simple. The difference of the final partition is on the order of the size of the last number to be assigned. For the greedy heuristic, this is the size of the smallest original number. This explains the small improvement with increasing numbers of values, since the more values we start with, the smaller the smallest of them is likely to be. For N numbers uniformly distributed between 0 and 1, the greedy method produces a final difference of $O(1/N)$. For the KK method, however, repeated differencing operations dramatically reduce the size of the remaining numbers. The more numbers we start with, the more differencing operations, and hence the smaller the size of the last numbers. Karmarkar and Karp have shown that the value of the final difference is $O(1/N^{\alpha \log N})$, for some constant α[5].

2.3 Finding Optimal Solutions

Both algorithms above run in $O(N \log N)$ time, but only find approximate solutions. If we want an optimal solution, the obvious algorithm is to search a binary tree, where at each node the left branch assigns the next unassigned number to one subset, and the right branch assigns it to the other subset. We keep track of the best final difference found during the search, and return it as the result, along with the actual subsets if desired.

The time complexity of this algorithm is $O(2^N)$, since we have to search a binary tree of depth N, and its space complexity is $O(N)$, since we can search the tree depth-first. There are two ways of pruning this tree, however. If we reach a node where the difference between the current subset sums is greater than or equal to the sum of all the remaining unassigned numbers, the best we can do is to assign all the remaining numbers to the smaller subset. For example, consider the state $15, 0(6, 5, 4)$, which results from assigning the 8 and 7 to the same subset. Since the sum of 6, 5, and 4 is no greater than the current subset difference of 15, the best we can do is to put all the remaining numbers in the other subset. Since this pruning doesn't depend on the best solution found so far, the size of the tree is independent of the order in which it is searched.

If we reach a terminal node whose subset difference is zero or 1, representing a perfect partition, then we can terminate the search. The above example illustrates this, since once we assign the remaining numbers to the other subset, the resulting complete partition has a difference of zero. If a perfect partition exists, then the search order matters, since the sooner we find a perfect partition, the sooner we can quit. The obvious way to order the search is to sort the numbers in decreasing order, and always put the next number in the smaller subset, before putting it in the larger subset. This algorithm produces the greedy solution first, and continues to search for better solutions, until an optimal solution is eventually found and verified.

Several additional optimizations deserve mention. One is that the first number should only be assigned to one subset, cutting the search space in half. The second is that whenever the current subset sums are equal, the next number should only be assigned to one subset, cutting the remaining subtree in half. The first optimization is in fact a special case of the second, but in practice only this special case yields significant performance improvements. Finally, when only one unassigned number remains, it should be assigned only to the smaller subset. Figure 3 shows the resulting binary tree for the numbers $(4, 5, 6, 7, 8)$, where the number in front of the set is the difference between the current subset sums, and the numbers below the leaf nodes represent the corresponding final partition differences.

2.4 Pseudopolynomial-Time Algorithm

There is also another rather different algorithm for finding optimal solutions. This is a dynamic programming approach which assumes that the original numbers are integers, or can be mapped to integers, such as a set of rational numbers. It requires an array whose size is half the sum of all the numbers. The algorithm enumerates all possible subset sums, as opposed to all possible subsets, to determine the achievable subset sum closest to half the total sum. Unfortunately, the space complexity of this algorithm makes it impractical for numbers of even moderate size.

3 Complete Karmarkar-Karp

Similar to the extension of the greedy heuristic to a complete algorithm, the main contribution of this paper is to extend the KK heuristic to a complete algorithm. While the idea is extremely simple, it doesn't appear in Karmarkar and Karp's original paper[5], and apparently escaped a number of other researchers who worked on the problem subsequently[6; 4; 2].

At each cycle, the KK heuristic commits to placing the two largest numbers in different subsets, by replac-

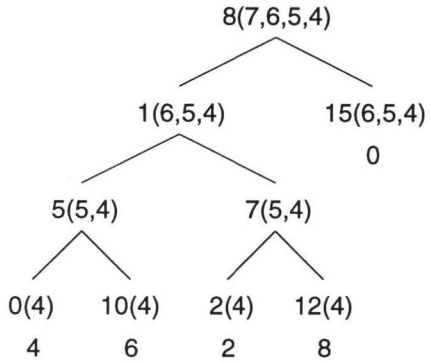

Figure 3: Example tree generated by standard algorithm

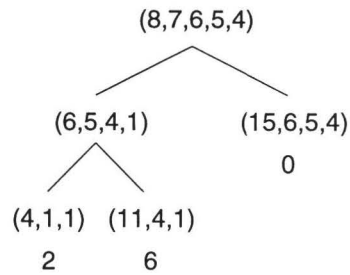

Figure 4: Example tree generated by CKK algorithm

ing them with their difference. The only other option is to place them in the same subset. This is done by replacing them by their sum. The resulting algorithm, which we call Complete Karmarkar-Karp (CKK), also searches a binary tree, where each node replaces the two largest numbers. The left branch replaces them by their difference, while the right branch replaces them by their sum. The difference is inserted in sorted order in the remaining list, while the sum is simply appended to the head of the list, since it will always be the largest element. Thus, the first solution found by CKK is the KK solution, but as it continues to run it finds better solutions, until an optimal solution is eventually found and verified.

In the worst case, the time complexity is still $O(2^N)$. However, the same pruning rules apply as in the standard algorithm, with the largest element of the set playing the role of the current subset difference. In other words, a branch is terminated when the largest element is greater than or equal to the sum of all the remaining elements. Figure 4 shows the resulting binary tree for the numbers $(4, 5, 6, 7, 8)$. Note that the tree in Figure 4 is smaller than that in Figure 3, even though both find optimal solutions to the same problem instance.

There are two reasons why CKK is more efficient than the standard complete algorithm, depending on whether or not a perfect partition exists. If there is no perfect partition, then both algorithms must search the whole tree. This is illustrated by the left subtrees in Figure 3 and Figure 4, where both algorithms place the 8 and 7 in different subsets. This state is represented by $1(6, 5, 4)$ in Figure 3, where 1 is the current subset difference, and by $(6, 5, 4, 1)$ in Figure 4B. The distinction between these two representations is that in the latter case, the difference of 1 is treated like any other number, and inserted at the end of the sorted order, instead of having the special status of the current subset difference. Thus, at the next level of the tree, represented by nodes $(4, 1, 1)$ and $(11, 4, 1)$ in Figure 4, the largest number is greater than the sum of the remaining numbers, and these branches can be pruned. In the standard algorithm, however, the two children of the left subtree, $5(5, 4)$ and $7(5, 4)$ in Figure 3, do not have this property, and have to be expanded further. Thus, CKK allows more pruning than the standard algorithm.

The second reason that CKK is more efficient occurs when a perfect partition does exist. In that case, for the same reason that the KK heuristic produces better solutions than the greedy heuristic, the CKK algorithm finds better solutions sooner, including the perfect solution. This allows it to terminate the search much earlier than the standard algorithm, on average.

4 Experimental Results

To demonstrate these effects, we implemented both the standard complete algorithm and the CKK algorithm, which both find optimal solutions. The results for two-way partitioning are shown in Figure 5. We chose random numbers uniformly distributed from 0 to 10 billion, which have ten significant decimal digits. Each data point is the average of 100 random problem instances. To make the algorithms more efficient, the CKK algorithm directly computes the optimal partition when there are four numbers left, since the KK heuristic is optimal in that case. The equivalent stopping point for the standard algorithm is when there are three unassigned numbers remaining, with the current subset difference playing the role of the fourth number. The horizontal axis shows the number of values partitioned, with data points for sets of size 5, 10, 15, ..., 95, 100. The vertical axis shows the number of nodes generated by the two algorithms. The descending line shows the average optimal partition difference on the vertical axis, fortuitously representable on the scale.

Both algorithms were coded in C, and generate approximately 15 million nodes per minute on a SUN SPARC 2. Thus, the whole vertical axis represents about an hour of computation. While CKK would seem to require more time per node to insert the computed difference in the remaining list, this amounts to only a constant factor, since most of the nodes in the tree are near the bottom, where the lists are very short. This constant is made up for by the fact that when there are only four numbers remaining, the final partition difference can be computed more efficiently by CKK, since they are in sorted order. For the standard algorithm, the last subset difference is not in sorted order with respect to the remaining three numbers.

There are clearly two different regions of this graph, depending on how many values are partitioned. With less than 30 numbers, no perfect partitions were found, while with 40 or more numbers, a perfect partition was found in every case. The optimal solution quality aver-

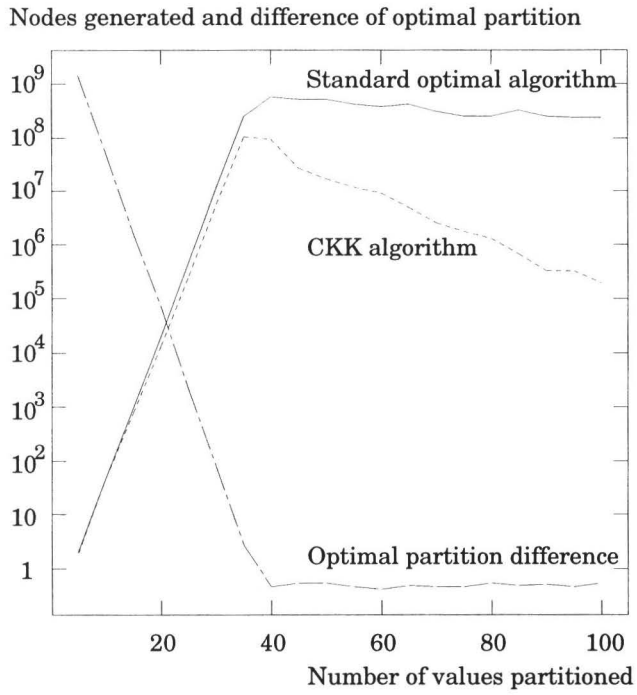

Figure 5: Nodes generated by optimal algorithms

ages .5 beyond 40 numbers, since there are roughly equal numbers of final differences of zero and one.

Figure 5 shows that CKK outperforms the standard algorithm over the entire range. Without a perfect partition, there is a small asymptotic improvement. The ratio of the number of nodes generated by the standard algorithm compared to CKK grows linearly with the number of values partitioned. This suggests that CKK is asymptotically more efficient than the standard algorithm.

The performance improvement is much more dramatic when a perfect partition exists. In that case, CKK finds the perfect partition much sooner than the standard algorithm, and hence terminates the search earlier. As the problem size increases, the running time of the standard algorithm drops gradually, but the running time of CKK drops precipitously, resulting in orders of magnitude improvement. We have run CKK on 10-digit problems up to size 300, and the trend seen in Figure 5 continues, bottoming out at about 37 milliseconds to partition 300 numbers. At that point, the KK solution is almost always optimal, and the running time is dominated by the $O(N \log N)$ time to find this first solution.

The data in Figure 5 is for numbers with ten decimal digits of precision, to allow running many trials with different numbers of values. Arbitrary-size single problem instances with up to twelve digits of precision can be solved in practice, however. Since no physical quantities are known with higher precision, any two-way partitioning problems that arise in practice can be optimally solved, regardless of problem size. While all our experiments were run on uniformly distributed values, we believe that the same results will apply to other naturally occurring distributions as well.

Most of the work on number partitioning, however, has focussed on problems without perfect partitions. To generate large such problem instances, numbers with up to 36 decimal digits have been used[6]. As long as there is no perfect partition, the performance of both CKK and the standard algorithm is largely independent of the precision of the numbers being partitioned, except for a constant based on whether single-precision, double-precision, or multiple-precision arithmetic is used. The data above was collected with double-precision arithmetic. To optimally partition 40 15-digit double-precision numbers with CKK requires an average of about 3 hours and 42 minutes, while the standard algorithm requires an average of 9 hours and 16 minutes.

For larger problems with very high precision, we must settle for approximate solutions. In that case, we can run CKK for as long as time allows, and return the best solution found. The first solution found is the KK solution, and as the algorithm continues to run, it finds better solutions. This technique is very effective, since much of the improvement in solution quality occurs early in the run. Figure 6 shows the improvement as a function of running time for partitioning 40 15-digit numbers. The horizontal axis is the number of nodes generated on a logarithmic scale, and the vertical axis is the ratio of the initial KK solution to the best solution found for a given number of node generations, also on a logarithmic scale. The entire horizontal scale represents less than a minute of real time, and shows almost five orders of magnitude improvement, relative to the original KK solution. Almost three orders of magnitude improvement is obtained within a second of running time.

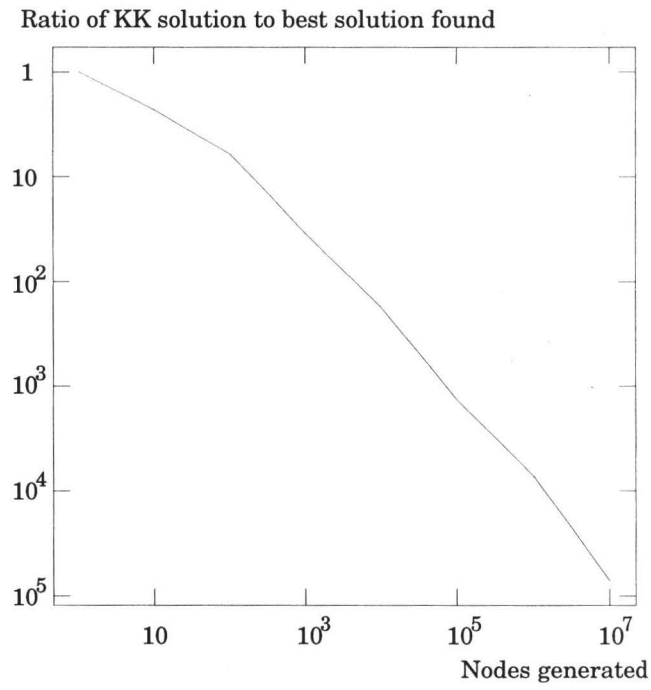

Figure 6: Solution quality relative to KK solution

5 Multi-Way Partitioning

So far, we have discussed partitioning a set of numbers into two subsets. Here we briefly discuss the generalization of all these techniques to partitioning into multiple subsets, omitting the details due to space limitations. The general task is to partition a set of numbers into K mutually exclusive and collectively exhaustive subsets, so that the difference between the largest subset sum and the smallest subset sum is minimized.

5.1 Greedy and Standard Algorithms

The generalizations of the greedy heuristic and the standard optimal algorithm to K-way partitioning are straightforward. We sort the numbers in decreasing order, and maintain K different subsets. For the greedy heuristic, we always place the next number in the subset with the smallest sum so far. The complete algorithm places each number in each different subset, in increasing order of their subset sums, searching a K-ary tree. By never placing a number in more than one empty subset, we avoid generating duplicate partitions that differ only by a permutation of the subsets, and produce all $N^K/K!$ distinct K-way partitions of N elements. We prune the tree when the sum of the remaining elements is small compared to the differences between the current subset sums. We only have to maintain $K-1$ subset sums, since we can always subtract the smallest subset sum from each of the others without affecting the final difference. A perfect partition has a difference of zero if the sum of the original numbers is divisible by K, and a difference of one otherwise.

5.2 Karmarkar-Karp Heuristic

Karmarkar and Karp generalize their set differencing method to K-way partitioning as follows. Every element represents a partial partition, with potentially K subset sums. The initial numbers each represent a partial partition with the number itself in the largest subset, and the remaining subset sums equal to zero. For example, a three-way partition of the set $(4,5,6,7,8)$ would initially be represented by the subpartitions $((8,0,0),(7,0,0),(6,0,0),(5,0,0),(4,0,0))$, which are sorted in decreasing order. Then the two largest numbers are combined into a single subpartition by putting them in different subsets, resulting in the list $((8,7,0),(6,0,0),(5,0,0),(4,0,0))$. Since the combined subpartition still has the largest subset sum, the next smaller subpartition, $(6,0,0)$, is combined with it by placing the 6 in the smallest subset, resulting in the subpartition $(8,7,6)$. Since we are only interested in the difference between the largest and the smallest subset sums, we subtract the smallest sum, 6, from each of the subsets, yielding the subpartition $(2,1,0)$. This subpartition is then inserted into the remaining sorted list in decreasing order by largest subset sum, resulting in $((5,0,0)(4,0,0)(2,1,0))$. Again, the two largest are combined, yielding $((5,4,0)(2,1,0))$. Finally, these last two subpartitions are merged by combining the largest subset sum with the smallest, the smallest subset sum with the largest, and the two medium subset sums together, yielding $(5,5,2)$. Subtracting the smallest from all the subset sums results in the final subpartition of $(3,3,0)$, which has a difference of 3, and is optimal in this case. While we have shown all three subset sums for clarity, our actual implementation only maintains the two non-zero values for each subpartition. Additional bookkeeping is required to recover the partition itself.

5.3 Complete Karmarkar-Karp Algorithm

The CKK algorithm also generalizes to multi-way partitioning. As in the case of two-way partitioning, instead of combining subpartitions in only one way, to make the algorithm complete we must combine them together in all possible ways. Again consider three-way partitioning. A particular subpartition represents a commitment to keep the elements in the different subsets separate. There are three cases to consider in combining a pair of subpartitions. In the first case, both subpartitions have only a single non-zero subset sum, say $(X,0,0)$ and $(A,0,0)$. We can combine these in two different ways, resulting in $(X,A,0)$ or $(X+A,0,0)$. In the second case, one subpartition has a single non-zero subset sum and the other has two non-zero subset sums, say $(X,Y,0)$ and $(A,0,0)$. In this case we can combine them in three different ways, resulting in the subpartitions (X,Y,A), $(X,Y+A,0)$, and $(X+A,Y,0)$. Finally, both subpartitions can have two non-zero subset sums, say $(X,Y,0)$ and $(A,B,0)$. In this case, there are or six different ways to combine them: $(X,Y+B,A)$, $(X,Y+A,B)$, $(X+B,Y,A)$, $(X+B,Y+A,0)$, $(X+A,Y,B)$, and $(X+A,Y+B,0)$. In each case, the combined subpartitions are searched in increasing order of largest subset sums, so that the first solution found is the KK solution.

The resulting search tree has depth N, and nodes with branching factor two, three, and six. The number of leaf nodes, however, is no greater than that in the standard tree. This is proven by showing that each of the three groups of combinations above is mutually exclusive and collectively exhaustive, and hence each distinct partition is represented by exactly one leaf node.

Pruning works by comparing the subset sums in the largest subpartition to the sum of all values in the remaining subpartitions. A complete partition with difference zero or one is optimal, and terminates the search.

We implemented the above algorithms for three-way partitioning, obtaining similar results to those for two-way partitioning. Namely, there is a small asymptotic improvement when no perfect partition exists, and orders of magnitude improvement with perfect partitions.

While the constant factors for CKK and the standard algorithm are roughly the same for two-way partitioning, the three-way version of CKK is more complex. As a result, CKK runs about 33% slower per node generation than the standard algorithm, on three-way partitioning problems. While this reduces the absolute performance of CKK, it appears to be asymptotically more efficient than the standard algorithm, and runs faster in practice.

In order to run large numbers of three-way partitioning problems of different sizes, we used numbers with five significant decimal digits. Single instances of arbitrary size with six digits of precision can be solved in practice, however. Three-way partitioning is computa-

tionally more difficult than two-way, and partitioning into more subsets is likely to be harder still, since the number of distinct K-way partitions is $O(K^N/N!)$.

6 Stochastic Approaches

There have been at least three studies applying stochastic algorithms to number partitioning, none of which can guarantee optimal solutions. Johnson et al.[2] applied simulated annealing to the problem, but found that it was not competitive with the Karmarkar-Karp heuristic solution. Ruml et al.[6] applied various stochastic algorithms to some novel encodings of the problem, but their best results outperform the KK solution by only three orders of magnitude, compared to the five orders of magnitude CKK achieves in a minute. Jones and Beltramo[3] applied genetic algorithms to the problem, but don't mention the Karmarkar-Karp heuristic. Their technique fails to find an optimal solution to the single problem instance they ran, while the KK solution to this instance is optimal.

7 0/1 Knapsack Problem

Given a set of integers, and a constant C, the 0/1 knapsack problem asks if there exists a subset of the integers whose sum is exactly C. We can reduce this problem to the two-way partition problem, and hence apply CKK to the 0/1 knapsack problem as well. Let S be the sum of all the integers. Assume that $C \geq S/2$, and otherwise use $S - C$ for C. Add a new integer D such that $(S + D)/2 = C$, or $D = 2C - S$. If this augmented set can be perfectly partitioned with a difference of zero, then the subset of the perfect partition that does not contain D is a subset of the original numbers whose sum is exactly C, and hence a solution to the 0/1 knapsack problem. Conversely, if this augmented set cannot be perfectly partitioned, then there is no subset of the original numbers that sum to exactly C, and hence no solution to the 0/1 knapsack problem.

8 Summary and Conclusions

The main contribution of this paper is to extend a very effective polynomial-time approximation algorithm due to Karmarkar and Karp, to a complete algorithm (CKK). The first solution it finds is the KK solution, and as it continues to run it finds better solutions, until it eventually finds and verifies an optimal solution. For problems without perfect partitions, CKK appears to be asymptotically more efficient than the standard optimal algorithm. When a perfect partition exists, CKK outperforms the standard algorithm by orders of magnitude. We showed results for two-way partitioning, and have obtained similar results for three-way partitioning. In practice, two-way partitioning problems of arbitrary size can be solved if the numbers are restricted to no more than twelve significant digits of precision, while arbitrary-sized three-way partitioning problems can be optimally solved with six significant digits. For large problems with very high precision, CKK can be run as long as time is available, returning the best solution found when time runs out. CKK outperforms every algorithm we could find in the literature.

What contribution does this work make beyond the specific problem of number partitioning? First, CKK is directly applicable to the 0/1 knapsack problem, and may apply to other related problems as well. Secondly, it presents an example of an approach that may be effective on other combinatorial problems. Namely, we took a good polynomial-time approximation algorithm, and made it complete, so that the first solution found is the approximation, and then better solutions are found as long as the algorithm continues to run, eventually finding an optimal solution. Thirdly, it represents an example of another approach that may be more broadly applicable. Most algorithms for combinatorial problems construct a solution to a problem incrementally, adding one element at a time to a single partial solution. This is the case with the standard algorithm for number partitioning. The CKK algorithm, on the other hand, constructs a large number of partial solutions, and combines them together in all possible ways. In this case, this latter strategy is much more effective, and may be for other problems as well.

9 Acknowledgements

Thanks to Wheeler Ruml for introducing me to number partitioning, and to the Karmarkar-Karp heuristic. Thanks to Wheeler, Ken Boese, Alex Fukunaga, and Andrew Kahng for helpful discussions concerning this research, and to Pierre Hasenfratz for comments on an earlier draft. This work was supported by NSF Grant IRI-9119825, and a grant from Rockwell International.

References

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

[2] Johnson, D.S., C.R. Aragon, L.A. McGeoch, and C. Schevon, Optimization by simulated annealing: An experimental evaluation; Part II, graph coloring and number partitioning, *Operations Research*, Vol. 39, No. 3, 1991, pp. 378-406.

[3] Jones, D.R., and M.A. Beltramo, Solving partitioning problems with genetic algorithms, in Belew, R.K., and L.B. Booker (Eds.), *Proceedings of the Fourth International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, Ca., 1991, pp. 442-449.

[4] Kahng, A., personal communication, 1993.

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

[6] Ruml, W., J.T. Ngo, J. Marks, S. Shieber, Easily searched encodings for number partitioning, Technical Report TR-10-94, Center for Research in Computing Technology, Harvard University, Cambridge, Mass., 1994.