An Improved Algorithm for Optimal Bin Packing

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

Abstract

Given a set of numbers, and a set of bins of fixed capacity, the NP-complete problem of bin packing is to find the minimum number of bins needed to contain the numbers, such that the sum of the numbers assigned to each bin does not exceed the bin capacity. We present two improvements to our previous bin-completion algorithm. The first speeds up the constant factor per node generation, and the second prunes redundant parts of the search tree. The resulting algorithm appears to be asymptotically faster than our original algorithm. On problems with 90 elements, it runs over 14 times faster. Furthermore, the ratios of node generations and running times both increase with increasing problem size.

1 Introduction and Overview

Given a set of numbers, and a fixed bin capacity, the bin-packing problem is to assign each number to a bin so that the sum of the numbers assigned to each bin does not exceed the bin capacity. An optimal solution uses the fewest number of bins. For example, given the set of numbers 6, 12, 15, 40, 43, 82, and a bin capacity of 100, we can assign 6, 12, and 82 to one bin, and 15, 40, and 43 to another, for a total of two bins. This is an optimal solution to this instance, since the sum of all the numbers, 198, is greater than 100, and hence at least two bins are required. An example application is given a set of orders for wire of varying lengths, and a standard length in which it is manufactured, how to cut up the minimum number of standard lengths to fill the orders.

Bin packing was one of the earliest problems shown to be NP-complete[Garey & Johnson, 1979]. The vast majority of the literature on this problem concerns polynomial-time approximation algorithms, such as first-fit decreasing (FFD) and best-fit decreasing (BFD), and the quality of the solutions they compute. First-fit decreasing sorts the numbers in decreasing order, orders the bins, and assigns each number in turn to the *first* bin in which it fits. Best-fit decreasing sorts the numbers in decreasing order and then assigns each number in turn to the *fullest bin* in which it fits. First-fit decreasing requires three bins to pack the set of numbers above, while best-fit decreasing packs them into two bins. Both algorithms run in O(nlogn) time.

In this paper we are concerned with finding optimal solutions, for several reasons. In applications with small numbers of bins, even one extra bin is relatively expensive. In addition, being able to find optimal solutions to problem instances allows us to more accurately gauge the quality of approximation algorithms. Furthermore, an anytime algorithm for finding optimal solutions, such as that presented in this paper, can make use of any additional time available to find better solutions than those returned by polynomial algorithms. Finally, optimal bin packing is computationally challenging, and may lead to insights applicable to other problems.

First we review previous algorithms for optimal bin packing, including our bin-completion algorithm [Korf, 2002]. We then describe two improvements to bin completion. The first is an algorithm for generating undominated bin completions more efficiently, reducing the constant time per node generation. Next we describe a method to reduce the branching factor of the search, by eliminating branch points that are dominated by previous branches. In experiments on millions of problem instances with uniformly distributed random numbers, our new algorithm appears to be asymptotically faster than our original bin-completion algorithm. This is based on the observation that both the ratio of nodes generated by the two algorithms, and also their running times, grow with increasing problem size. On problems of size 90, our new algorithm runs over 14 times faster than our original bin-completion algorithm. We also report mixed results on a common set of benchmark problems.

2 Previous Work

2.1 Martello and Toth

A well-known algorithm for optimal bin packing [Martello & Toth, 1990a; 1990b] is based on depth-first branch-and-bound. The numbers are first sorted, and are considered from largest to smallest. It first computes an approximate solution as an initial upper bound, using the best solution among first-fit, best-fit, and worst-fit decreasing. Then, for each number, the algorithm places it in each partially-filled bin that it fits into, or in an empty bin. Thus, the algorithm branches on the different bins that a number can be placed in. It also uses a lower-bound function to prune the search.

2.2 More Recent OR Approaches

An anonymous reviewer pointed out some more recent references on optimal bin packing from the operations research literature, based on integer programming formulations, with linear-programming relaxations. [Valerio de CarvaJho, 1999; Scholl, Klein, & Jurgens, 1997; Vanderbeck, 1999; DeGraeve & Schrage, 1999; Vance *et a/.*, 1994; Vance, 1998] While we have not yet fully digested this work, we report below some comparison results on a standard set of benchmarks.

2.3 Bin Completion

In [Korf, 2002], we described our *bin completion* algorithm, which uses a branching structure different from that of Martello and Toth. A *feasible set* is a set of numbers whose sum does not exceed the bin capacity. Initially, we sort the numbers in decreasing order of size. We then generate feasible sets that include the largest number. If there is more than one such set, the search may branch at that point. Each node of the search tree, except the root node, represents a complete assignment of numbers to a particular bin. The children of the root represent different ways of completing the bin containing the largest number. The nodes at the next level represent different feasible sets that include the largest remaining number, etc. The depth of any branch of the tree is the number of bins in the corresponding solution.

Bin completion is also a branch-and-bound algorithm. It starts with the best-fit decreasing solution as an upper bound, and applies a lower-bound heuristic function to prune the search. Rather than assigning numbers one at a time to bins, it branches on the different feasible sets that can be used to complete each bin. Bin completion appears asymptotically faster than the Martello and Toth algorithm, and outperforms it by a factor of a thousand on problems of size 60. The key property that makes it more efficient is a dominance condition on the feasible completions of a bin that allows us to only consider a small subset of them.

2.4 Set Dominance

Some sets of elements assigned to a bin cannot lead to solutions that are any better than those achievable by assigning other sets of elements to the same bin. We begin with some simple examples of these dominance relations, and then consider the general formulation.

First, consider two elements x and y whose sum is exactly the bin capacity c. Assume that in one solution, x and y are in different bins. In that case, we can swap y with all other elements in the bin containing x, without increasing the number of bins. This gives us an equally good solution with x and y in the same bin. Thus, given a problem with two values x and y such that $x + y = c$, we can always put x and y in the same bin, resulting in a smaller problem [Gent, 1998]. Unfortunately, this does not extend to three or more elements that sum to exactly the bin capacity.

As another example, consider an element x such that any two remaining elements added to x will exceed c. In other words, at most one additional element can be added to the bin containing x. Let y be the largest remaining element such that $x + y \leq c$. Then, we can place y in the same bin as x without sacrificing solution quality. The reason is that if we placed any other single element z with x, then we could swap y with z, since $z \leq y$ and $x + y \leq c$.

As a final example, again assume that y is the largest remaining element that can be added to x such that $x + y \leq$ c, and that y equals or exceeds the sum of *any* set of remaining elements that can be added to x without exceeding c. In that case, we can again put x and y in the same bin, without sacrificing solution quality. The reason is that any other set of elements that were placed in the same bin as x could be swapped with y without increasing the number of bins.

To illustrate the general form of this dominance relation, let A and B be two feasible sets. If the elements in B can be partitioned into subsets, and the subsets can be matched to the elements of A such that the sum of the elements in each subset doesn't exceed the corresponding element of A, then set A *dominates* set B. In other words, if the elements of B can be packed into bins whose capacities are the elements of A, then set A dominates set B. For example, let $A = \{20, 30, 40\}$ and let $B = \{5, 10, 10, 15, 15, 25\}$. Partition B into the subsets $\{5, 10\}$, $\{25\}$, and $\{10, 15, 15\}$. Since $5 + 10 \leq 20$, $25 \leq 30$, and $10 + 15 + 15 \leq 40$, set A dominates set B.

Given all the feasible sets that contain a common element x, only the undominated sets need be considered for assignment to the bin containing x. The reason is that if we complete the bin containing x with a dominated set, then we could swap each subset of numbers in the dominated set with the corresponding element of the dominating set, and get another solution without increasing the total number of bins.

Martello and Toth use this dominance relation to some extent. In particular, they take each element x, starting with the largest element, and check if there is a *single* completion of one or two more elements that dominates all feasible sets containing x. If so, they place x with those elements in the same bin, and apply the reduction to the remaining subproblem. They also use dominance relations to prune some element placements as well.

Our bin-completion algorithm, however, makes much greater use of this dominance condition. In particular, when branching on the completion of any bin, it only considers undominated completions.

3 Finding Undominated Completions

The first contribution of this paper is a faster algorithm to generate all the undominated completions of a bin.

3.1 Previous Algorithm

The simplest algorithm is to generate all feasible sets that include a particular element, and then test each pair of sets for dominance. Our original implementation improved on this by only generating a subset of all feasible completions, and testing for pairwise dominance only among those. In particular, if only one additional number can be added to a bin, only the largest such number is added. If only two additional numbers can be added to a bin, the set of two-element undominated completions can be found in linear time, as follows.

Each undominated two-element completion must have a sum greater than the largest feasible single number, and less than or equal to the remaining capacity of the bin. The remaining values are kept in sorted order, with two pointers, one initially assigned to the largest value and the other to the smallest value. If the sum of these two numbers exceeds the remaining capacity, the pointer to the larger number is moved to the next smaller number. If the sum of the two numbers is less than or equal to the largest single feasible number, the pointer to the smaller number is moved to the next larger number. If the sum of the two numbers is within this range, the pointer to the smaller number is increased to the largest number for which the sum of the two is still in range, and these two values form an undominated two-element completion. Then the pointer to the larger number is moved to the next smaller number, and the pointer to the smaller number is moved to the next larger number. This process continues until the two pointers meet.

If the sum of the numbers in two feasible sets are unequal, only the one with the larger sum can dominate the other. If two sets have the same sum, only the one with the smaller cardinality can dominate the other. Once a subset of the feasible completions is found, each pair was tested to see if either dominates the other. This was done by trying to pack the numbers of the potential dominated set into bins whose capacities were the numbers of the potential dominating set. These small bin-packing problems were solved by brute-force search.

There are two drawbacks to this approach. The first is the time to generate and then test the dominated feasible sets. The second is the memory needed to store the dominated feasible sets before they are pruned.

3.2 Generating Undominated Sets Faster

Ideally, we would like to generate all and only undominated bin completions, without pairwise testing of feasible sets for dominance. We describe such an algorithm in three stages: 1) how to generate all subsets of a universe, 2) how to generate only feasible subsets, and 3) how to generate only undominated feasible subsets.

The easiest way to generate all 2^n subsets of n elements is to recursively traverse a binary tree, where each node represents a collection of subsets. The root node represents the entire power set, and the leaf nodes represent individual subsets. Each interior level of the tree corresponds to a different element. At each node, the left branch includes the corresponding element in all subsets below it, and the right branch excludes the same element from all subsets below it. The tree is traversed depth-first using only linear memory.

To generate feasible completions of a bin containing a number *x,* we add an upper bound on the sum to each recursive call, which is initially the residual capacity, or the bin capacity minus *x.* We sort the remaining unassigned numbers in decreasing order of size. We only include numbers that are less than or equal to the upper bound. When we include a number by taking the left branch from a node, we subtract it from the upper bound of all recursive calls below that branch. When the upper bound drops to zero, we prune the tree below that node, since no further numbers can be added. This generates all feasible completions of a bin containing *x.*

Generating undominated feasible completions requires a little more work. When we exclude a number that exceeds the upper bound, by taking the right branch from the corresponding node, nothing additional is required, since it can't be a member of any feasible set below that node. If we exclude a number that equals the upper bound, we can terminate that branch of the binary tree immediately, because any feasible subset of included elements below that node cannot sum to more than the excluded element, and any subset with a smaller sum would be dominated by the excluded element.

What happens when we exclude a number that is less than the upper bound? To prevent the excluded clement from dominating any included subset below it, the sum of the numbers in any such subset must exceed the excluded element. This generates a lower bound on the sum of the elements in any included subset below this node, which is equal to the excluded element plus one, assuming that the numbers are integers. As with the upper bound, the lower bound is reduced by any subsequently included elements in the recursive calls below the corresponding nodes. The lower bound generated by an excluded element is only used if it exceeds the current lower bound on that node.

Thus, we perform a depth-first traversal of the binary tree representing all possible subsets of numbers remaining to be assigned. This traversal is pruned by the upper and lower bounds propagated down the tree as described above, and generates complete subsets at the leaf nodes. These will include all undominated feasible sets, but may include some dominated feasible sets as well.

To eliminate these, we perform an additional test on the feasible bin completions generated. The residual capacity r of a bin is the bin capacity c minus the largest number in the bin, which is common to all feasible completions of a given bin. Let *t* be the sum of all the numbers in a feasible set *A,* excluding the common largest number. The excluded numbers are all remaining numbers less than or equal to r that are not included in *A.* Set *A* will be dominated if and only if it contains any subset whose sum *s* is less than or equal to an excluded number x, such that replacing the subset with *x* will not exceed the bin capacity. This will be the case if and only if $x - s \leq r - t$. Thus, to guarantee that a feasible set

A is undominated, we check each possible subset sum s, and each excluded number x, to verify that $x - s > r - t$.

This algorithm generates feasible sets and immediately tests them for dominance, so it never stores multiple dominated sets. It tests for dominance by comparing subset sums of included elements to excluded elements, rather than comparing pairs of sets for dominance.

Pseudo-code for this algorithm is given below. The *feasible* function takes a set I of included elements, a set E of excluded elements, a set R of remaining elements, a lower bound / and an upper bound u. It generates all feasible sets of remaining elements whose sum is within the two bounds, and calls the *test* function on each. In the initial call, I and E are empty, R contains all remaining elements less than or equal to the residual capacity r of the bin, u is set to r, and l is set to the largest single element that can feasibly be added to the bin, plus one. *Test* is the test described above, and is a function of the included elements J, the excluded elements E, and the residual capacity r.

```
feasible($I$, $E$, $R$, $l$, $u$)
  if $R$ is empty or $u = 0$, test($I$, $E$, $r$)
  else
    let $x$ be the largest element of $R$
    if $x > u$, feasible($I$, $E$, $R \setminus \{x\}$, $l$, $u$)
    else if $x = u$, feasible($I \cup \{x\}$,$E$,$R \setminus \{x\}$,$l - x$,$u - x$)
    else
      feasible($I \cup \{x\}$, $E$, $R \setminus \{x\}$, $l - x$, $u - x$)
      feasible($I$, $E \cup \{x\}$, $R \setminus \{x\}$, $max(l, x + 1)$, $u$)
```

To improve this algorithm, we use the same optimizations used in our original algorithm to generate feasible sets. Namely, if only one more number can be added to a bin, we only add the largest such number, and if only two more numbers can be added, we generate all undominated two-element completions in linear time. This algorithm speeds up the generation of all undominated sets, without affecting the number of bin completions considered. For that, we turn to our next contribution.

4 Pruning the Search Space

Consider a number w in a bin, with a capacity of c. Assume that two undominated feasible completions of the bin are $\{w, x, y\}$ and $\{w, z\}$. For these completions to be undominated, it must be the case that $z > x$, $z > y$, $x + y > z$, $w + x + z > c$, and $w + y + z > c$. Assume that our search explores bin completions in decreasing order of subset sum, so in this case we consider $\{w, x, y\}$ before $\{w, z\}$. Furthermore, assume that after exhausting the subproblem below the assignment $\{w, x, y\}$, and while exploring the subproblem below the assignment $\{w, z\}$, we find a solution that assigns x and y to the same bin, say $\{v, x, y\}$. Since $w + x + y \leq c$, $v + x + y \leq c$, and $z < x + y$, we could swap z with x and y, resulting in a solution with the same number of bins, but including the bin assignments $\{w, x, y\}$ and $\{v, z\}$. However, all possible solutions below the node representing the bin assignment $\{w, x, y\}$ have already been explored. Thus, this solution is redundant, and doesn't need to be considered again. In particular, below the $\{w, z\}$ branch of the search tree, any solution that assigns x and y to the same bin will be redundant and can be pruned.

In general, given a node with more than one child, when searching the subtree of any child but the first, we don't need to consider bin assignments that assign to the same bin all the numbers used to complete the current bin in a previously-explored child node. More precisely, let $\{N_1, N_2, ..., N_m\}$ be a set of brother nodes in the search tree, and let $\{S_1, S_2, ...S_m\}$ be the sets of numbers used to complete the bin in each node, excluding the first number assigned to the bin, which is common to all the brother nodes. When searching the subtree below node N_t for $i > 1$, we exclude any bin assignments that put all the numbers in Sj in the same bin, for $j < i$. Thus, no bin completion below node Ni can have as a subset the numbers in S_j, for $j < i$. By rejecting these bin assignments as redundant, the number of node generations is reduced.

4.1 Current Implementation

Our current implementation of this pruning rule propagates a list of *nogood sets* along the tree. After generating the undominated completions for a given bin, we check each one to see if it contains any current nogood sets as a subset. If it does, we ignore that bin completion.

To keep the list of nogood sets from getting too long, occupying memory to store them and time to test them against bin completions, we prune the list as follows. Whenever there is a non-empty intersection between a bin completion and a nogood set, but the nogood set is not a subset of the bin completion, we remove that nogood set from the list that is passed down to the children of that bin completion. The reason is that by including at least one but not all the numbers in the nogood set in a bin completion, we've split up the nogood set, guaranteeing that it can't be a subset of any bin completion below that node in the search tree.

This implementation could probably be improved with more sophisticated data structures for representing arid manipulating sets of elements.

5 Experimental Results

We tested our algorithm on large sets of problems with uniformly-distributed high-precision numbers, and on a set of benchmark problems of relatively low precision.

5.1 Uniform High Precision Numbers

We compared our original algorithm to our new bin-completion algorithm on the same problem instances and on the same machine. Since high-precision numbers are often more difficult to pack than low-precision numbers, we used a bin capacity of one million, and random numbers uniformly distributed from one to one million. Given the enormous variation in the difficulty of individual problem instances, we ran one million instances of each problem size, which ranged from 5 to 95 numbers, in increments of 5. Table 1 shows the results.

[N	Optimal	Original	Without Pruning		With Nogood Pruning		Ratios	
	Bins	Time	Nodes	Time	Nodes	Time	Nodes	Time
5	3.215	6	.064	6	.064	5	1.000	1.200 1
10	5.966	13	.119	13	.119	12	1.000	1.083
15	8.659	19	.362	20	.360	19	1.006	1.000
20	11.321	27	.727	28	.716	26	1.015	1.038
25	13.966	36	1.249	37	1.204	35	1.037	1.029
30	16.593	44	2.046	46	1.878	43	1.099	1.023
35	19.212	55	3.376	57	2.827	52	1.194	1.058
40	21.823	73	6.325	71	4.452	65	1.421	1.123
45	24.427	103	13.346	94	7.338	81	1.819	1.272
| 50	27.026	189	29.414	136	12.364	104	2.379	1.817
55	29.620	609	124.476	367	28.931	174	4.303	3.500
60	32.210	2,059	391.847	1,097	108.527	518	3.611	3.975
65	34.796	28,216	7,984.196	15,694	649.553	2,658	12.292	10.616
70	37.378	41,560	9,408.125	22,628	786.126	3,549	11.968	11.710
75	39.957	194,851	57,529.770	119,928	5,308.159	21,739	10.838	8.963
80	42.534	408,580	113,746.144	233,367	7,560.130	30,972	15.046	13.192
85	45.108	412,576	129,618.988	282,851	8,697.441	36,098	14.903	11.429
90	47.680	2,522,993			38,176.160	171,778		14.688
95	50.253				324,811.294	1,343,092		

Table 1: Experimental Results for Uniformly Distributed, High-Precision Numbers

The first column gives the problem size, which is the number of values being packed. The second column shows the average number of bins needed in the optimal solution. Since the numbers range uniformly from zero to the bin capacity, the expected value of any number is half the bin capacity, and the expected value of the sum of the numbers is the half the bin capacity times the number of values. As expected, the average minimum number of bins is slightly more than half the number of values, due to the inevitable wasted space in the bins.

The third column gives the average running time of our original bin-completion algorithm [Korf, 2002], in microseconds. This is also the total time in seconds to solve all one million problem instances. All implementations are on a 440 Megahertz Sun Ultra 10 workstation.

The next two columns, labelled "Without Pruning", give the average node generations and running times in microseconds for our implementation of bin completion with our new method of generating undominated feasible sets, but without pruning nogood sets. While this program considers the same number of candidate solutions as our original one, the node generations differ from those reported in our earlier paper [Korf, 2002]. The reason is that we define a node as a recursive call to the search routine, and our current implementation checks terminating conditions before making a recursive call, rather than at the beginning of the search function. Our new program outperforms our original one by a factor of up to 1.84 in running time. We didn't run it on problems of size 90 or 95 due to the time that would be required.

The next two columns, labelled "With Nogood Pruning", give the average number of nodes generated and average running time in microseconds for our full algorithm, including nogood pruning. Comparing the number of node generations to the corresponding column without pruning shows the effect of nogood pruning.

The last two columns give performance ratios of our best program, including nogood pruning. The node ratio is the number of nodes generated without nogood pruning, divided by those generated with nogood pruning. The time ratio is the running time of our original program, divided by our current best program.

As problem size increases, nogood pruning generates increasingly fewer nodes than without pruning. On the largest problems we ran both algorithms on, the ratio of node generations is about a factor of 15. The fact that the node generation ratio increases with increasing problem size suggests that nogood pruning reduces the asymptotic time complexity of bin completion.

The ratios of the running times displays a similar trend, although the values are less than the ratios of node generations. This is due to the increased overhead of nogood pruning. On the larger problems, our new algorithm is over an order of magnitude faster than our original algorithm. Problems of size 95 take an average of only 1.343 seconds per problem to solve optimally.

Variation in Individual Problem Difficulty

There is tremendous variation in the difficulty of individual problems. For example, in 68.56% percent of the one million problems with 95 numbers, the best-fit decreasing solution uses the same number of bins as the lower bound, solving the problem without any search. Among the same problems, however, twenty instances generated more than a billion nodes, three of those generated more than ten billion nodes, and one of those generated more than a hundred billion nodes. Our program solved over 125,000 problems of size 100 in about a day, and then failed to solve the next problem in over 43 days. What distinguishes the hard problems from the easy ones?

Among all problems of size 95, the average number of bins in the optimal solution is 50.253. Among the twenty hardest problems, however, the average optimal number of bins is only 39.85. Intuitively, this makes sense, since fewer bins means more items per bin, and hence more undominated feasible sets to consider.

On the other hand, problems with a relatively small number of bins in the optimal solution are not necessarily difficult. The reason is that with smaller numbers, approximation algorithms like best-fit decreasing are more accurate. For example, eleven problems of size 95 required 36 or fewer bins. Seven of those required no search, and the average number of nodes generated to solve all eleven was only 12,704, compared to an average of 324,811 nodes for all one million problem instances.

Thus, the hard problems tend to use fewer bins, but problems that use fewer bins are not necessarily hard, since often the lower bounds agree with the solutions returned by approximation algorithms.

5.2 Benchmark Problems

We also ran our best algorithm on eight sets of twenty benchmark problems each, from the operations research library maintained by J.E. Beasley at Imperial College, London. These problem instances were originally generated by Falkenauer [Falkenauer, 1996], and have been used by a number of other researchers [Valerio de Carvalho, 1999; Vanderbeck, 1999]. We compare our results to [Valerio de Carvalho, 1999], since he reports the most detailed results. He ran his experiments on a 120 MHz Pentium, compared to our 440 MHz Sun workstation.

Uniform Problems

The first four problem sets, called *uniform* problems, consist of numbers chosen uniformly from 20 to 100, with a bin capacity of 150. Each set contains 20 problems, which are of size 120, 250, 500, and 1000 numbers each.

On the uniform problems of size 120, our best algorithm took 2 seconds on problem 0, 3 seconds on problem 3, and solved the rest instantly, meaning in less than a second. In eleven of these problems, either the best-fit decreasing solution, or the first solution found by bin completion, matched our lower bound, requiring no search. [Valerio de Carvalho, 1999] reports an average of 4.22 seconds for these problem instances.

For the uniform problems of size 250, our algorithm solved all but three instantaneously, solved problem 15 in 318 seconds, but failed to solve problems 7 and 13 in over ten minutes each. In twelve of these problems, the first solution found by bin completion matched our lower bound. [Valerio de Carvalho, 1999] reports an average of 5.98 seconds for these problems.

On uniform problems of size 500, our algorithm solved 14 problems instantly, took 4 and 160 seconds on problems 3 and 5, respectively, but failed to solve problems 0, 6, 7, and 8 in ten minutes each. In eleven of these problems, the first solution found by bin completion matched our lower bound. [Valerio de Carvalho, 1999] reports an average of 6.93 seconds on these instances.

For uniform problems of size 1000, our algorithm solved problems 0, 1, 5, 6, 8, 10, 11, 13 and 18 instantly, and solved problems 14 and 16 in 69 and 10 seconds, respectively. In seven of these problems, the first solution found by bin completion matched our lower bound. It failed, however, to solve the remaining 9 problems in ten CPU minutes each. [Valerio de Carvalho, 1999] reports an average of 7.45 seconds for these problems.

Triplet Problems

The other four problem sets are called *triplets,* since each bin contains exactly three elements in the optimal solution. The bin capacity is 1000, with numbers in the range 250 to 500. The first number in each bin was chosen uniformly from 380 to 490, the second was chosen from 250 to one-half the size of the first number, and the third element was chosen so that the sum of the three is exactly 1000. Thus, no space is wasted in the optimal solution.

Our algorithm solved all 20 triplets of size 60 instantly, while [Valerio de Carvalho, 1999] reports an average of 4.14 seconds on these problems.

On triplets of size 120, our algorithm solved all but four problems instantly, and required 26, 1, 3, and 2 seconds on problems 0, 4, 13, and 19, respectively, for an average time of 1.6 seconds. [Valerio de Carvalho, 1999] reports an average of 26.95 seconds on these instances.

On triplets of size 249, our algorithm solved instances 3, 4, 9, 11, and 19 instantly, required 2, 19, 206, 157, 1, 75 and 126 seconds on problems 0, 6, 7, 8, 10, 13, and 15, respectively, but failed to solve the remaining 8 problems after 10 minutes each. [Valerio de Carvalho, 1999] reports an average of 122.85 seconds on these problems.

We were unable to solve any of the triplets of size 501, with ten CPU minutes each. [Valerio de Carvalho, 1999] reports an average of 360.69 seconds on these instances.

Discussion of Benchmark Results

On the uniform problems of size 120, and on the triplets of size 60 and 120, our algorithm outperformed that of [Valerio de Carvalho, 1999] in average running time, taking into account the different clock speeds of our machines. On the remaining sets of problems, our algorithm would require a longer average running time, since it failed to solve some instances in ten minutes. On uniform problems of size 250 and 500, however, our algorithm took less time than that of [Valerio de Carvalho, 1999] on most problem instances, 17 out of 20 for size 250, and 14 out of 20 for size 500. On uniform problems of size 1000, our algorithm took less time on 9 out of 20 problems, and on triplets of size 249 our algorithm was faster on 8 out of 20 problems. Overall, our algorithm performed worse than that of [Valerio de Carvalho, 1999] on the largest problem sets, however.

One possible reason for this difference in performance is that we designed our algorithm with high-precision numbers in mind, ranging from one to a million in our experiments. With these values, there are no duplicate numbers in the same problem instance, and no pairs of numbers that sum to exactly the bin capacity. Thus, we didn't consider optimizations that are only possible with duplicate numbers. In the uniform benchmark datasets, however, the values range from 20 to 100, and in the triplets datasets they range from 250 to 500. As a result, these problem sets contain many identical numbers.

While high-precision values are usually more difficult to pack, this is not the case with the triplet datasets considered here. In particular, if they were generated in the same way, but using high-precision values instead, the first solution found by bin completion would be optimal, since there would be only one way to fill each bin completely, and that completion would be considered first. Furthermore, it would be immediately recognized as optimal, since there is no extra space in any of the bins.

6 Conclusions

We presented two improvements to our original bin-completion algorithm. The first is an algorithm for generating all undominated bin completions directly, without testing pairs of completions for dominance. More importantly, we presented an algorithm to identify and eliminate redundant bin completions, which prunes the search space. Combining these two improvements yields an algorithm that appears to be asymptotically faster than our original, and runs over 14 times faster on problems of size 90. For numbers uniformly distributed from zero to the bin capacity, we can solve a million problems of size 95 optimally in an average of 1.343 seconds per problem instance.

There is enormous variation in individual problem difficulty, with most problems being solved instantly, but some running for days or weeks. One problem of size 100 ran for 43 days without verifying an optimal solution.

On a set of standard benchmark problems, the results were mixed. Our algorithm outperformed that of [Valerio de Carvalho, 1999] on the smaller problem instances, but did worse on the largest problem instances. These benchmarks problems contain relatively low-precision values, which may provide further opportunities for improving the performance of bin completion.

It is important to note that most of the algorithms described in this paper are anytime algorithms. In other words, they produce an approximate solution immediately, and as they continue to run they produce better solutions, until they find and eventually verify the optimal solution. Furthermore, the gap between the approximate solution and the lower bound is often only a single bin, and always known throughout the computation, allowing the user to decide how much effort to expend in trying to achieve this improvement.

7 Acknowledgments

Thanks to the anonymous reviewers, particularly for pointing out the recent work on this problem in the operations research community. This work was supported by NSF under grant No. EIA-0113313, by NASA and JPL under contract No. 1229784, and by the State of California MICRO grant No. 01-044.

References

[DeGraeve & Schrage, 1999] DeGraeve, Z., and Schrage, L. 1999. Optimal integer solutions to industrial cutting stock problems. *INFORMS Journal on Computing* 11(4):406-419.

[Falkenauer, 1996] Falkenauer, E. 1996. A hybrid grouping genetic algorithm for bin packing. *Journal of Heuristics* 2:5-30.

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

[Gent, 1998] Gent, I. 1998. Heuristic solution of open bin packing problems. *Journal of Heuristics* 3:299-304.

[Korf, 2002] Korf, R. 2002. A new algorithm for optimal bin packing. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-02),* 731-736. Edmonton, Alberta, Canada: AAAI Press.

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

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

[Scholl, Klein, & Jurgens, 1997] Scholl, A.; Klein, R.; and Jurgens, C. 1997. Bison: A fast hybrid procedure for exactly solving the one-dimensional bin packing problem. *Computers and Operations Research* 24(7):627-645.

[Valerio de Carvalho, 1999] Valerio de Carvalho, J. 1999. Exact solution of bin-packing problems using column generation and branch-and-bound. *Annals of Operations Research* 86:629-659.

[Vance *et al.*, 1994] Vance, P. H.; Barnhart, C; Johnson, E. L.; and Nemhauser, G. L. 1994. Solving binary cutting stock problems by column generation and branch-and-bound. *Computational Optimization and Applications* 3:111-130.

[Vance, 1998] Vance, P. H. 1998. Branch-and-price algorithms for the one-dimensional cutting stock problem. *Computational Optimization and Applications* 9:211-228.

[Vanderbeck, 1999] Vanderbeck, F. 1999. Computational study of a column generation algorithm for bin packing and cutting stock problems. *Mathematical Programming Series A* 86:565-594.