A New 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

We consider the NP-complete problem of bin packing. Given a set of numbers, and a set of bins of fixed capacity, find the minimum number of bins needed to contain all the numbers, such that the sum of the numbers assigned to each bin does not exceed the bin capacity. We present a new algorithm for optimal bin packing. Rather than considering the different bins that each number can be placed into, we consider the different ways in which each bin can be packed. Our algorithm appears to be asymptotically faster than the best existing optimal algorithm, and runs more that a thousand times faster on problems with 60 numbers.

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 all numbers assigned to each bin does not exceed the bin capacity. An optimal solution to a bin-packing problem uses the fewest number of bins possible. For example, given the set of elements 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 problem instance, since the sum of all the elements (198) is greater than 100, and hence at least two bins are required.

Optimal bin packing one of the classic NP-complete problems (Garey & Johnson 1979). The vast majority of the literature on this problem concerns polynomial-time approximation algorithms, such as first-fit and best-fit decreasing, and the quality of the solutions they compute, rather than optimal solutions. We discuss these approximation algorithms in the next section.

The best existing algorithm for optimal bin packing is due to Martello and Toth (Martello & Toth 1990a; 1990b). We present a new algorithm for optimal bin packing, which we call *bin completion*, that explores a different problem space, and appears to be asymptotically faster than the Martello and Toth algorithm. On problems of size 60, bin completion runs more than a thousand times faster than Martello and Toth's algorithm. We are able to optimally solve problems with 90 elements in an average of 2.5 seconds per problem.

Approximation Algorithms

A simple approximation algorithm is called first-fit decreasing (FFD). The elements are sorted in decreasing order of size, and the bins are kept in a fixed order. Each element is placed into the first bin that it fits into, without exceeding the bin capacity. For example, given the elements 82, 43, 40, 15, 12, 6, and a bin capacity of 100, first-fit decreasing will place 82 in the first bin, 43 in a second bin, 40 in the second bin, 15 in the first bin, 12 in the second bin, and 6 in a third bin, for a total of three bins, which is one more than optimal.

A slightly better approximation algorithm is known as best-fit decreasing (BFD). It also sorts the elements in decreasing order of size, but puts each element into the *fullest* bin in which is fits. It can be implemented by keeping the bins sorted in increasing order of their remaining capacity, and placing each element into the first bin in which it fits. For example, given the set of elements 82, 43, 40, 15, 12, 6, best-fit decreasing will place 82 in bin a, 43 in bin b, 40 in bin b, 15 in bin b, because it is fuller than bin a, 12 in bin a, and 6 in bin a, for a total of two bins, which is optimal.

Both FFD and BFD can be implemented in $O(n \log n)$ time, but are not guaranteed to return optimal solutions. However, either algorithm is guaranteed to return a solution that uses no more than 11/9 of the optimal number of bins (Johnson 1973). On average, BFD performs slightly better than FFD. For example, on problems of 90 elements, where the elements are uniformly distributed from zero to one million, and the bin capacity is one million, BFD uses an average of 47.732 bins, while FFD uses an average of 47.733 bins. On these same problem instances, the optimal solution averages 47.680 bins. The FFD solution is optimal 94.694% of the time, and the BFD solution is optimal 94.832% of the time on these problem instances.

Optimal Solutions

Given the high quality solutions returned by these approximation algorithms, why bother trying to find optimal solutions? There are at least four reasons. In some applications, it may be important to have optimal solutions. In particular, with small numbers of bins, even a single extra bin is relatively expensive. In addition, being able to determine the optimal solutions to problem instances allows us to more accurately gauge the quality of approximate solutions. For example, the above comparisons of FFD and BFD solutions to

optimal solutions were only possible because we could compute the optimal solutions. Another reason is that an anytime algorithm for finding optimal solutions, such as those presented in this paper, can make use of any additional time available to find better solutions than those returned by BFD or FFD, which run very fast in practice. Finally, optimal bin packing is an interesting computational challenge, and may lead to insights applicable to other problems.

Estimated Wasted Space or L2 Lower Bound

A lower bound function for bin packing takes a problem instance, and efficiently computes a lower bound on the minimum number of bins needed. If we find a solution that uses the same number of bins as a lower bound, then we know that solution is optimal, and we can terminate the search.

An obvious lower bound is to sum all the elements, divide by the bin capacity, and round up to the next larger integer. A better bound starts with the sum of the elements, and adds an estimate of the total bin capacity that must be wasted in any solution, before dividing by the bin capacity. This is the L2 bound of Martello and Toth (Martello & Toth 1990a; 1990b), but we give a simpler and more intuitive algorithm for computing it below.

For example, consider the set of elements 99, 97, 94, 93, 8, 5, 4, 2, with a bin capacity of 100. There is no element small enough to go in the same bin as the 99, so that bin will have one unit of wasted space in any solution. The only element small enough to go in the same bin as the 97 is the 2, so the 2 can be placed with the 97 without sacrificing optimality, leaving a second unit of wasted space.

There are two remaining elements that could be placed with the 94, the 5 or the 4. In reality, only one of these elements could be placed with the 94, but to make our lower-bound calculation efficient and avoid any branching, we assume that we can place as much of their sum (9) as will fit in the bin with the 94, or 6 units. Thus, we assume there is no wasted space in this bin, and 3 units are carried over to the next bin, which contains the 93. The sum of all elements less than or equal to the residual capacity of this bin (7) is just the 3 units carried over from the previous bin. Therefore, at least $7 - 3 = 4$ additional units will be wasted between this bin and the previous one. Finally, there are no remaining elements to be placed with the 8, so 92 units must be wasted in this bin. Thus, the total wasted space will be at least $1 + 1 + 4 + 92 = 98$ units, which is added to the sum of all the elements before dividing by the bin capacity.

This estimated wasted-space calculation proceeds as follows. We consider the elements in decreasing order of size. Given an element x, the residual capacity r of the bin containing x is $r = c - x$, where c is the bin capacity. We then consider the sum s of all elements less than or equal to r, which have not already been assigned to a previous bin. There are three possible cases. The first is that r equals s. In that case, there is no estimated waste, and no carry over to the next bin. If $s < r$, then $r - s$ is added to the estimated waste, and again there is no carryover to the next bin. Finally, if $r < s$, then there is no waste added, and $s - r$ is carried over to the next bin. Once the estimated waste is computed, it is added to the sum of the elements, which is divided by the bin capacity, and then rounded up.

This estimated wasted space adds significantly to the sum of the elements. For example, on problems with 90 elements, uniformly distributed from zero to one million, with a bin capacity of one million, the estimated wasted-space lower bound averages 47.428 bins, while the simple sum lower bound averages only 45.497. For comparison, the average optimal solution for these problem instances is 47.680.

Martello and Toth Algorithm

The best existing algorithm for finding optimal solutions to bin-packing problems is due to Martello and Toth (Martello & Toth 1990a; 1990b). Their branch-and-bound algorithm is complex, and we describe here only the main features. Their basic problem space takes the elements in decreasing order of size, and places each element in turn into each partially-filled bin that it fits in, and into a new bin, branching on these different alternatives. This results in a problem space bounded by $n!$, where n is the number of elements, but this is a very pessimistic upper bound, since many elements won't fit in the same bins as other elements.

At each node of the search tree, Martello and Toth compute the first-fit, best-fit, and worst-fit decreasing completion of the corresponding partial solution. A partial solution to a bin-packing problem is one where some but not all elements have already been assigned to bins. A completion of a partial solution takes the current partially-filled bins, and assigns the remaining unassigned elements to bins. The worst-fit decreasing algorithm places each successive element in the partially-filled bin with the largest residual capacity that will accommodate that value.

Each of these approximate solutions is compared to a lower bound on the remaining solution that they call L3. The L3 bound is computed by successively relaxing the remaining subproblem by removing the smallest element, and then applying the L2 bound to each of the relaxed instances, returning the largest such lower bound. Martello and Toth's L2 bound equals the estimated wasted-space bound described above, but they use a more complex algorithm to compute it. If the number of bins used by any of the approximate completions equals the lower bound for completing the corresponding partial solution, no further search is performed below that node. If the number of bins in any approximate solution equals the lower bound on the original problem, the algorithm terminates, returning that solution as optimal.

The main source of efficiency of the Martello and Toth algorithm is a method to reduce the size of the remaining subproblems, which we will discuss below under "Dominance Relations". First, however, we very briefly discuss another optimal bin-packing program due to Fekete and Schepers.

Fekete and Schepers Lower Bound

Fekete and Schepers (Fekete & Schepers 1998) use the same algorithm as Martello and Toth, but with a more accurate lower-bound function. They claim that the resulting algorithm outperforms Martello and Toth's algorithm, based on

solving more problems with the same number of node generations, but didn't report running times. We implemented both algorithms, but found that the Fekete and Schepers algorithm was slower than that of Martello and Toth, because their lower bound function took longer to compute, and this wasn't compensated for by the reduced node generations.

Dominance Relations

Some sets of elements assigned to a bin are guaranteed to lead to solutions that are at least as good as 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, such that $x + y = c$, where c is the bin capacity. Assume that in an optimal 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 a new optimal 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 the smallest 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, 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.

The general form of this dominance relation is due to Martello and Toth (Martello & Toth 1990a; 1990b). Define a *feasible* set as any set of elements whose sum doesn't exceed the bin capacity. Let A and B be two feasible sets. If all the elements of B can be packed into a set of bins whose capacities are the elements of A, then 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.

Martello and Toth use this dominance relation to reduce the size of subproblems generated by their search. Given a partially-solved problem where some elements have already been assigned to bins, Martello and Toth first convert the partially-solved problem to an equivalent initial problem, where no elements have been assigned to bins, in two steps. First, any element that is unassigned, or assigned to a bin with no other elements, is left unchanged. Second, for each bin that contains more than one element, all the elements assigned to that bin are replaced with a single element equal to their sum, guaranteeing that any elements assigned to the same bin will stay together.

To reduce the size of such a problem, Martello and Toth take each element x in turn, starting with the largest element, and check to see if there is a *single* set of three or fewer elements, including x, that dominates all feasible sets containing x. If so, they place x with those elements in the same bin, and recursively apply the same reduction algorithm to the resulting reduced problem. They also use dominance relations to prune some placements of elements into bins.

Bin-Completion Algorithm

Our optimal bin-packing algorithm, which we call *bin completion*, makes more effective use of these dominance relations. Like the Martello and Toth algorithm, bin completion is also a branch-and-bound algorithm, but searches a different problem space. Rather than considering each element in turn, and deciding which bin to place it in, we consider each bin in turn, and consider the undominated feasible sets of elements that could be used to complete that bin. We sort the elements in decreasing order of size, and consider the bins containing each element in turn, enumerating all the undominated completions of that bin, and branching if there are more than one. In other words, we first complete the bin containing the largest element, then complete the bin containing the second largest element, etc.

Example Problem

To illustrate our algorithm, consider an example problem consisting of the elements {100, 98, 96, 93, 91, 87, 81, 59, 58, 55, 50, 43, 22, 21, 20, 15, 14, 10, 8, 6, 5, 4, 3, 1, 0}, with a bin capacity of 100. The 100 completely occupies a bin, and the 0 takes no space, resulting in the problem {98, 96, 93, 91, 87, 81, 59, 58, 55, 50, 43, 22, 21, 20, 15, 14, 10, 8, 6, 5, 4, 3, 1}. The only element that can go with 98 is 1, so we put them together, leaving {96, 93, 91, 87, 81, 59, 58, 55, 50, 43, 22, 21, 20, 15, 14, 10, 8, 6, 5, 4, 3}. We place 96 and 4 together, since they sum to exactly the bin capacity, leaving {93, 91, 87, 81, 59, 58, 55, 50, 43, 22, 21, 20, 15, 14, 10, 8, 6, 5, 3}. Since the sum of the two smallest remaining elements, 5 and 3, exceeds the residual capacity of the bin containing 93, we can't place more than one element in that bin, and choose the largest such element 6 to go with 93,, leaving {91, 87, 81, 59, 58, 55, 50, 43, 22, 21, 20, 15, 14, 10, 8, 5, 3}. We can complete the bin containing 91 with 8, 5, 3, or $5 + 3$. Since the 8 dominates the other single elements, and also the set $5 + 3$, we place 8 with 91, leaving {87, 81, 59, 58, 55, 50, 43, 22, 21, 20, 15, 14, 10, 5, 3}. The bin containing 87 can be completed with 10, 5, 3, $10 + 3$ or $5 + 3$. All of these are dominated by $10 + 3$, so we place $10 + 3$ with 87, resulting in {81, 59, 58, 55, 50, 43, 22, 21, 20, 15, 14, 5}. Up to this point, no branching is needed, since there is only one undominated choice for completing each of the bins considered so far.

The bin containing 81 can be completed with 15, 14, 5, or $14 + 5$. Both 15 and $14 + 5$ dominate both 14 and 5 individually. However, neither 15 nor $14 + 5$ dominate each other, so both must be considered, producing a two-way branch in

the search. Heuristically, we choose to explore the completion with the largest sum first, adding $14 + 5$ to the bin with 81, leaving $\{59, 58, 55, 50, 43, 22, 21, 20, 15\}$. We can complete the bin containing 59 with 22, 21, 20, 15, $22 + 15$, $21 + 20$, $21 + 15$, or $20 + 15$. Of these, only $22 + 15$ and $21 + 20$ are undominated by any of the others. This produces another two-way branch in the search, and we choose the alternative with the greater sum, $21 + 20$, to place first with 59, leaving $\{58, 55, 50, 43, 22, 15\}$. To complete the bin with 58, there is only one undominated choice, $22 + 15$, so we put these three elements together, leaving $\{55, 50, 43\}$.

Note that the only elements which could possibly be placed in the same bin with an element of size 58 or larger are those of size 42 or smaller. At this point, all elements of size 42 or smaller have already been placed in a bin with an element of size 58 or larger. While we could rearrange some of the smaller elements in those bins, no such rearrangement could possibly reduce the number of such bins, because each element of size 58 or larger requires its own bin. Thus, there is no need to backtrack to either of the branch points we encountered, and we can continue the search as if all the previous decisions were forced. The general form of this rule is the following: If $x > c/2$, where c is the bin capacity, and all elements $\leq c - x$ have been placed in bins containing elements $\geq x$, then any branch points generated in filling bins with elements $\geq x$ can be ignored.

The remaining problem of $\{55, 50, 43\}$ requires two more bins, yielding an optimal solution of eleven bins, as follows: $\{100\}$, $\{98, 1\}$, $\{96, 4\}$, $\{93, 6\}$, $\{91, 8\}$, $\{87, 10, 3\}$, $\{81, 14, 5\}$, $\{59, 21, 20\}$, $\{58, 22, 15\}$, $\{55, 43\}$, $\{50\}$.

General Algorithm

Our overall bin-completion algorithm is as follows. First we compute the best-fit decreasing (BFD) solution. Next, we compute a lower bound on the entire problem using the wasted-space bound described above. If the lower bound equals the number of bins in the BFD solution, it is returned as the optimal solution. Otherwise we initialize the best solution so far to the BFD solution, and start a branch-and-bound search for strictly better solutions. Once a partial solution uses as many bins as the best complete solution found so far, we prune that branch of the search. Experimentally, it was not worthwhile to also compute the first-fit decreasing (FFD) solution, since the FFD solution very rarely uses fewer bins than the BFD solution.

We consider the elements in decreasing order of size, and generate all the undominated completions of the bin containing the current element. If there are no completions or only one undominated completion, we complete the bin in that way and go on to the bin containing the next largest element. If there is more than one undominated completion, we order them in decreasing order of total sum, and consider the largest first, leaving the others as potential future branch points. Whenever we find a complete solution that is better than the best so far, we update the best solution found so far.

For a lower bound on a partial solution, we use the sum of all the elements, plus the actual space remaining in the bins completed so far, divided by the bin capacity and rounded up to the next larger integer. Equivalently, we can add the number of bins completed to the sum of the remaining elements, divided by the bin capacity and rounded up. This lower bound is more effective than the estimated wasted-space bound described above, because it includes the actual wasted space in the completed bins. Furthermore, it is computed in constant time, by just accumulating the amounts of wasted space in the completed bins. Surprisingly, given this computation, it doesn't help to additionally compute the estimated wasted space in the remaining bins, for reasons we don't have sufficient space to describe here. Thus, we replace a linear-time lower-bound function with a more accurate constant-time function, with no loss in pruning power. If the lower bound on a partial solution equals or exceeds the number of bins in the best solution so far, we prune consideration of that partial solution.

Most of the time in our algorithm is spent computing the undominated completions of a bin. Our current implementation generates a subset of the feasible completions, then tests these for dominance. Given a particular element x, we first find the largest element y for which $x + y \leq c$.

We then compute all feasible pairs w and z, such that $x + w + z \leq c$, which are undominated by the single element y, i.e. $w + z > y$, and such that no pair dominates any other pair. Given two pairs of elements, $d + e$ and $f + g$, all four elements must be distinct, or the pair with the larger sum will dominate the other. Assume without loss of generality that $d > f$. In that case, g must be greater than e, or $d + e$ will dominate $f + g$. Thus, given any two pairs of elements, neither of which dominates the other, one must be completely "nested" inside the other in sorted order. We can generate all such undominated pairs in linear time by keeping the remaining elements sorted, and maintaining two pointers into the list, a pointer to a larger element and a pointer to a smaller element. If the sum of the two elements pointed to exceeds the bin capacity, we bump the larger pointer down to the next smaller element. If the sum of the two elements pointed to is less than or equal to y, we bump the smaller pointer up to the next larger element. If neither of these cases occur, we have an undominated pair, and we bump the larger pointer down and the smaller pointer up. We stop when the pointers reach each other.

After computing all undominated completion pairs, we then compute all triples d, e, and f, such that $x + d + e + f \leq c$ and $d + e + f > y$, then all quadruples, etc. To compute all triples, we choose each feasible first element, and then use our undominated pairs algorithm for the remaining two elements. For all quadruples, we choose all possible feasible pairs, then use our undominated pairs algorithm for the remaining two elements, etc. These larger feasible sets will in general include dominated sets.

Given two feasible sets, determining if the one with the larger sum dominates the other is another bin-packing problem. This problem is typically so small that we solve it directly with brute-force search. We believe that we can significantly improve our implementation, by directly generating all and only undominated completions, eliminating the need to test for dominance.

N	Optimal	L2 bound	% Optimal		Martello + Toth		Bin Completion		Ratio
			FFD	BFD	Nodes	Time	Nodes	Time	Time
5	3.215	3.208	100.000%	100.000%	0.000	7	.013	6	1.17
10	5.966	5.937	99.515%	99.541%	.034	15	.158	13	1.15
15	8.659	8.609	99.004%	99.051%	.120	25	.440	19	1.32
20	11.321	11.252	98.570%	98.626%	.304	37	.869	27	1.37
25	13.966	13.878	98.157%	98.227%	.741	55	1.500	36	1.53
30	16.593	16.489	97.790%	97.867%	2.146	87	2.501	44	1.98
35	19.212	19.092	97.478%	97.561%	7.456	185	4.349	55	3.36
40	21.823	21.689	97.153%	97.241%	39.837	927	8.576	73	12.70
45	24.427	24.278	96.848%	96.946%	272.418	6821	20.183	103	66.22
50	27.026	26.864	96.553%	96.653%	852.956	20799	57.678	189	110.05
55	29.620	29.445	96.304%	96.414%	6963.377	200998	210.520	609	330.05
60	32.210	32.023	96.036%	96.184%	58359.543	2153256	765.398	2059	1045.78
65	34.796	34.598	95.780%	95.893%			11758.522	28216	
70	37.378	37.167	95.556%	95.684%			16228.245	41560	
75	39.957	39.736	95.322%	95.447%			90200.736	194851	
80	42.534	42.302	95.112%	95.248%			188121.626	408580	
85	45.108	44.866	94.854%	94.985%			206777.680	412576	
90	47.680	47.428	94.694%	94.832%			1111759.333	2522993	

Table 1: Experimental Results

Experimental Results

Martello and Toth tested their algorithm on only twenty instances each of sizes 50, 100, 200, 500, and 1000 elements. They ran each problem instance for up to 100 seconds, and reported how many were optimally solved in that time, and the average times for those problems. Fekete and Schepers ran 1000 problem instances each, of size 100, 500, and 1000. They ran each instance for 100,000 search nodes, and reported the number of instances solved optimally. Both sets of authors used a bin capacity of 100, and chose their values from three distributions: uniform from 1 to 100, uniform from 20 to 100, and uniform from 35 to 100.

We found these experiment unsatisfactory for several reasons. The first is that we observed over eleven orders of magnitude variation in the difficulty of individual problem instances. For most problems, the number of bins in the BFD solution equals the lower bound, requiring no search at all. The hardest problems we solved required over 100 billion node generations, however. Performance on a large set of problems is determined primarily by these hard problems. A problem set of only 20 or even 1000 problems is unlikely to include really hard problems, and furthermore, they terminated the processing of their hard problems when a certain computational budget was exhausted.

Another difficulty with this approach is the use of integer values no larger than 100. Problems with such low precision values are significantly easier than problems with high precision values. For example, the simple preprocessing step of removing all pairs of elements that sum to exactly the bin capacity will eliminate most elements from a problem with 500 or 1000 elements up to 100. In contrast, with real numbers, we would expect no such reduction in problem size.

Finally, two of the distributions used by Martello and Toth and Fekete and Schepers eliminate small values, namely 20 to 100 and 35 to 100. As their data show, these problems are significantly easier than problems in which the values are chosen uniformly from one to the bin capacity. The reason is that all elements greater than one-half the bin capacity get their own bins, and no bins can contain very many elements.

For these reasons, we performed a different set of experiments. To include hard problems, we completed the optimal solution of all problem instances. Given the enormous variation in individual problem difficulty, one must address considerably larger problem sets in order to get meaningful averages. We solved ten million problem instances of each size from 5 to 50, and one million problem instances of each larger size. To avoid easy problems resulting from low precision values, we chose our values uniformly from zero to one million, with a bin capacity of one million. This simulates real numbers, but allows integer arithmetic for efficiency. This also avoids the easier truncated distributions described above. Table 1 above shows our results.

The first column is the problem size, or the number of elements. The second column is the average number of bins used in the optimal solution. As expected, it is slightly more than half the number of elements, with the difference due to wasted space in the bins. The third column is the average value of the L2 or wasted-space lower bound. The next two columns show the percentage of problem instances in which the first-fit decreasing (FFD) and best-fit decreasing (BFD) heuristics return the optimal solution. Note that these are not all easy problems, since verifying that the heuristic solution is optimal can be very expensive. For example, in 94.832% of problems of size 90 the BFD solution was optimal, but in only 69.733% of problems of this size was the L2 lower bound equal to the number of bins in the BFD solution, meaning that the BFD solution could be returned as the optimal solution with no search. All these percentages decrease monotonically with increasing problem size.

The sixth column is the average number of nodes gener-

ated by the Martello and Toth algorithm, and the seventh column is the average running time in microseconds per problem. The eighth column is the average number of nodes generated by our bin-completion algorithm, and the ninth column is the average running time in microseconds per problem on the same problem instances. The last column gives the running time of the Martello and Toth algorithm divided by the running time of our bin-completion algorithm. Both programs were implemented in C, and run on a 440 megahertz Sun Ultra 10 workstation.

It is clear that our bin-completion algorithm dramatically outperforms the Martello and Toth algorithm on the larger problems. The ratio of the running times of the two algorithms appears to increase without bound, strongly suggesting that our algorithm is asymptotically faster. On problems of size 60, which were the largest for which we could solve a million instances using the Martello and Toth algorithm, their algorithm took almost 25 days, while our algorithm took less than 35 minutes, which is over a thousand times faster. We were able to run the Martello and Toth algorithm on 10,000 problem instances of size 65, in a total of 614,857 seconds, which is over a week. Our bin completion algorithm took 19 seconds to solve the same instances, which is 32,360 times faster. Another view of this data is that we were able to increase the size of problems for which we could solve a million instances from 60 elements with the Martello and Toth algorithm, to 90 elements with our bin completion algorithm, an increase of 50% in problem size.

Some problems were hard for both algorithms, and others were hard for one algorithm were easy for the other. In general, the difficult problems required significantly fewer bins than average problems of the same size, or equivalently, packed more elements per bin. For example, on 60 problems of size 90, the bin-completion algorithm generated more than a billion nodes. The average number of bins used by the optimal solutions to these problems was 38.4, compared to 47.68 bins for an average problem of this size.

Conclusions and Discussion

We have presented a new algorithm for optimal bin packing. Rather than branching on the number of different possible bins that an element can be assigned to, we branch on the number of undominated ways in which a bin can be completed. Our bin-completion algorithm appears to be asymptotically faster than the best existing algorithm, and runs more than a thousand times faster on problems of 60 elements. We also presented a simpler derivation of Martello and Toth's L2 lower bound, and an efficient algorithm for computing it. An additional advantage of our bin-completion algorithm is that it is much simpler and easier to implement than the Martello and Toth algorithm.

Since both algorithms use depth-first branch-and-bound, they are anytime algorithms, meaning that they can be run for as long as time allows, and then return the best solution found at that point. For large problems where solution quality is important, it may be worthwhile to spend more time than is required to run an $O(n \log n)$ approximation algorithm such as best-fit or first-fit decreasing. In that case, either the Martello and Toth or bin completion algorithms could be run as an anytime algorithm, but the above data suggest that bin completion will find better solutions faster.

Many hard combinatorial problems involve packing elements of one set into another. For example, floor planning, cutting stock, and VLSI layout problems involve packing elements of various shapes and sizes into a single region of minimum size. The usual approach to optimally solve these problems involves considering the elements one at time, in decreasing order of size, and deciding where to place them in the overall region. The analog of our bin-completion algorithm would be to consider the most constrained regions of the space, and determine which elements can be placed in those regions. A clear example of the virtue of this approach is solving a jigsaw puzzle. Rather than individually considering each piece and where it might go, it's much more efficient to consider the most constraining regions of a partial solution, and find the pieces that can go into that region.

This duality between pieces and spaces has been exploited by Knuth in exact covering problems, such as polyominoe puzzles (Knuth 2000). We have shown that this alternative problem space can also be very useful in non-exact covering problems such as bin packing. Since many other packing problems have similar features, we believe that our general approach will be applicable to other problems as well.

Acknowledgements

Thanks to Russell Knight for helpful discussions concerning this research, and to Victoria Cortessis for a careful reading of the manuscript. This research was supported by NSF under grant No. IIS-9619447, and by NASA and JPL under contract No. 1229784.

References

Fekete, S., and Schepers, J. 1998. New classes of lower bounds for bin packing problems. In Bixby, R.; Boyd, E.; and Rios-Mercado, R., eds., *Integer Programming and Combinatorial Optimization: Proceedings of the 6th International IPCO Conference*, 257–270. Houston, TX: Springer.

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

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

Johnson, D. 1973. *Near-Optimal Bin Packing Algorithms*. Ph.D. Dissertation, Dept. of Mathematics, M.I.T., Cambridge, MA.

Knuth, D. 2000. Dancing links. In Davies, J.; Roscoe, B.; and Woodcock, J., eds., *Millenial Perspectives in Computer Science*. Palgrave: Houndmills, Bashingstake, Hampshire. 187–214.

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

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