Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence

Improved Bin Completion for Optimal Bin Packing and Number Partitioning

Ethan L. Schreiber and **Richard E. Korf**
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095 USA
ethan@cs.ucla.edu, korf@cs.ucla.edu

Abstract

The bin-packing problem is to partition a multiset of n numbers into as few bins of capacity C as possible, such that the sum of the numbers in each bin does not exceed C. We compare two existing algorithms for solving this problem: bin completion (BC) and branch-and-cut-and-price (BCP). We show experimentally that the problem difficulty and dominant algorithm are a function of n, the precision of the input elements and the number of bins in an optimal solution. We describe three improvements to BC which result in a speedup of up to five orders of magnitude as compared to the original BC algorithm. While the current belief is that BCP is the dominant bin-packing algorithm, we show that improved BC is up to five orders of magnitude faster than a state-of-the-art BCP algorithm on problems with relatively few bins. We then explore a closely related problem, the number-partitioning problem, and show that an algorithm based on improved bin packing is up to three orders of magnitude faster than a BCP solver called DIMM which claims to be state of the art. Finally, we show how to use number partitioning to generate difficult bin-packing instances.

1 Introduction & Overview

Given a multiset $S = \{s_1, s_2..., s_n\}$ of n elements and a bin capacity C, all positive integers, the bin-packing problem is to separate the elements of S into as few subsets (bins) as possible, such that the sum of each subset is no greater than C. For example, given $S = \{1, 2, 2, 3, 5, 6, 8\}$ and $C = 10$, there is an optimal packing into 3 bins: $\{2, 8\}, \{1, 2, 5\}$ and $\{3, 6\}$. An example application is packing computer files of varying size into memory blocks of fixed size. Bin packing is NP-Complete (Problem SR1 of [Garey and Johnson, 1979]).

Given S and a positive integer $k \geq 2$, the number-partitioning problem is to find the smallest capacity C such that the elements of S can be packed into k bins. For example, with $S = \{1, 2, 2, 3, 5, 6, 8\}$ and $k = 3$, the optimal partitioning is $\{1, 8\}, \{2, 2, 5\}$ and $\{3, 6\}$ requiring a capacity of 9. The classic application is partitioning a set of processes onto k processors such that the total time to complete all jobs (the makespan) is minimized. In fact, number partitioning is also called multiprocessor scheduling and is also NP-Complete (Problem SS8 of [Garey and Johnson, 1979]).

First, we describe lower and upper bounds for bin packing. We then present two optimal bin-packing algorithms: bin completion (BC) [Korf, 2003], a heuristic search algorithm; and branch-and-cut-and-price (BCP) [Valério de Carvalho, 1999; Vanderbeck, 1999; Belov and Scheithauer, 2006], an integer linear programming algorithm which is considered the current state of the art. We also present algorithmic improvements to BC which we call improved bin completion (IBC).

We run a set of bin packing experiments and show that the difficulty in solving an instance depends on n, the number of bins in the solution and the precision of the input elements. We then describe the relationship between bin packing and number partitioning and how to use bin-packing algorithms to solve number-partitioning problems. Furthermore, we show how to use solutions to the number-partitioning problem to find hard bin packing instances.

2 Bounds for Bin Packing

There is a large literature on both upper and lower bounds for bin packing. Since this paper focuses on optimal solutions and not approximation algorithms, we describe two lower bounds and one upper bound, refering the reader to external sources for more information on bounds. For example, see [Fekete and Schepers, 2001; Coffman Jr *et al.*, 1996].

2.1 L_1 and L_2 Lower Bounds

A bin packing lower bound function efficiently computes the minimum number of bins of capacity C needed to pack the input set S. If we find a packing of this many bins while searching for a solution, we can terminate our search and immediately return that solution as optimal.

Martello and Toth introduce two lower bounds [1990b]. The lower bound L_1 sums the elements, divides by the bin capacity and rounds up. L_2, also referred to as the wasted space heuristic, is explained clearly by Korf [2002]. L_2 calculates the amount of bin capacity that must be wasted (unused) in any packing and adds this to the sum of all input elements before dividing by the bin capacity and rounding up.

Let w be the wasted space, initialized to 0. While there are still elements left in S, remove the largest element s_1 and calculate the residual bin capacity $r = C - s_1$ left after s_1 is

placed in a bin. Then, remove from S all elements with value less than or equal to r and store their sum in the variable sum. If $sum \leq r$, add $r - sum$ to w. If $sum > r$, then $sum - r$ is carried over and added to the sum computed for the next bin. The algorithm terminates when no elements are left in S. L_2 is then calculated as:

$$L_2(S) = \left\lceil \frac{1}{C} \left(w + \sum_{i=1}^{n} s_i \right) \right\rceil$$

2.2 First-Fit & Best-Fit Decreasing Upper Bounds

A feasible packing is an assignment of elements to bins such that the sum of the elements in each bin is not greater than C. Any feasible packing of S provides an upper bound. First-fit decreasing (FFD) and best-fit decreasing (BFD) [Johnson, 1973] are classic approximation algorithms for bin packing. For both, we sort the elements of S in decreasing order and consider the items one at a time. For FFD, each element is placed into the first bin it fits into. For BFD, each element is placed into the most full bin it fits into. Consider S={15,10,6,4,3,2} and C=20. The FFD and BFD solutions are $[\{15, 4\}, \{10, 6, 3\}, \{2\}]$ and $[\{15, 3, 2\}, \{10, 6, 4\}]$.

3 Bin Completion (BC)

Bin-packing algorithms by Eilon and Christofides [1971] and Martello and Toth [1990a] consider input elements one at a time and assign them to bins. In contrast, bin completion [Korf, 2002; 2003] considers the bins one at a time and assigns a complete set of elements to them. We first describe Korf's original algorithm [1], then our extensions to it.

3.1 The Original BC Algorithm

BC is a branch-and-bound algorithm with initial lower and upper bounds computed using L_2 wasted space and BFD. If these bounds are equal, the BFD solution is returned as optimal. Otherwise, a tree search is performed with the variable *best* initialized to the number of bins in the BFD solution.

A feasible set is a set of input elements with sum no greater than C. We call the assignment of a feasible set to a bin a bin completion. Each node of the tree except the root corresponds to a bin completion. The children of the root correspond to the completions of the bin containing the largest element. The grandchildren of the root correspond to the completions of the bin containing the largest remaining element, and so forth. We include the largest element for two reasons. First, to avoid duplicates that differ only by a permutation of the bins. Second, to shrink the remaining capacity of the bin which results in fewer feasible bin completions to consider.

The waste of a bin completion is the capacity remaining after it has been packed with feasible set F. For example, if C=10 and F={4,5}, the waste is 1. BC keeps track of the sum of the waste in all bins completed so far. At any point during our search, we have already found a solution requiring $best$ bins. Therefore, we only consider solutions of $best$-1 or fewer bins. To achieve such a solution, the sum of the waste in all bins must be no greater than $[(best - 1) * C] - sum(S)$, which we call the total allowed waste or W.

At each node of the search tree, BC generates all feasible sets which include the largest remaining element and whose waste when added to the previous bins' waste does not exceed the total allowed waste. These sets are sorted by their sums in decreasing order. BC then branches on each feasible set F, removing the elements of F from S for the subtree beneath the node corresponding to F. If all elements are packed in fewer than $best$ bins, $best$ is updated to the new value. If this new value equals the lower bound, BC terminates, returning this packing. Otherwise, the search terminates when all completions have been exhaustively searched.

3.2 Generating Completions

At each node in the BC search tree, we branch on a set of feasible completions. As mentioned above, we always include the largest remaining element. Call w the sum of the waste so far in all completed bins, which cannot exceed W, the total allowed waste. The sum of each completion must be in the range $[C - (W - w), C]$.

Call R the set of elements remaining. The simplest way to exhaustively search for all subsets in a range is using an inclusion/exclusion binary search tree. The nodes at depth d of the tree correspond to the d^{th} element of R. The left edge leaving a node corresponds to including the element in all subsets below it and the right edge corresponds to excluding the element. As we exhaustively search the tree, we keep a running sum of all included elements at a node. At any terminal node, if the sum is in range, we add the corresponding subset to our list of completions. If the sum is $\geq C$, we can prune as all distinct subsets below would have sum $> C$. We can also prune if the sum of the remaining elements on the path plus the running sum are smaller than $C - (W - w)$.

3.3 Dominance

Some completions are dominated by others and need not be considered. Given feasible sets F_1 and F_2, F_1 dominates F_2 if the optimal solution after packing a bin with F_1 is at least as good as the optimal solution after packing the bin with F_2.

Martello and Toth [1990a] present the following dominance relation for bin packing. If all the elements of feasible set F_2 can be packed into bins whose capacities equal the elements of feasible set F_1, then F_1 dominates F_2. For example, if $F_1 = \{6, 4\}$ and $F_2 = \{6, 3, 1\}$, then F_1 dominates F_2 since the elements of F_2 can be packed into bins whose capacities equal the elements of F_1 as follows: $\{\{6\}, \{3, 1\}\}$. Intuitively, this makes sense since any place where we can fit the element 4, we can also fit the elements 3 and 1. However, there are places where 3 or 1 can fit in which 4 cannot fit. We use dominance relations to prune parts of the search space, dramatically cutting down the size of the search tree. Efficiently generating only undominated completions is similar but more involved than described in section 3.2. We refer the reader to [Korf, 2003] for details.

3.4 Our Improvements to Bin Completion

We have implemented three improvements to bin completion. While the changes are simple conceptually, the experimental

[1] Korf wrote two papers on bin completion, when we say original, we are referring to his 2003 paper which had some implementation improvements over his 2002 paper.

results are dramatic, speeding up the BC algorithm by up to five orders of magnitude.

Incrementally Generated Completions

The original BC algorithm generates all completions at each node before sorting them. At any node, if there are n elements remaining, then there are 2^n possible subsets. If a large fraction of these subsets have sums within the valid range $[C-(W-w), C]$, then there are an exponential number of completions to generate. This situation tends to arise when the number of elements per bin in an optimal solution is large. In this case, BC spends all of its time generating completions.

To avoid generating an exponential number of completions, we generate and buffer up to b undominated feasible completions at a time, sort them and branch. If we don't find an optimal solution recursively searching the subtrees of each of these b completions, we then generate the next b completions.

The implementation of the inclusion/exclusion tree for the original BC Algorithm used a recursive depth-first search (DFS) with pruning rules as described in section 3.2. To perform the search over the inclusion/exclusion tree incrementally, we use an iterative DFS that maintains its own explicit stack data structure. This way, we can generate as many completions at a time as necessary and keep our explicit stack in memory so we can come back later to resume the search.

Variable Ordering

As mentioned in the last section, after all feasible completion sets are generated, they are sorted in decreasing order of subset sum. However, this is not a total order. If two subsets have the same subset sum, it is arbitrary which completion will be tried first. This can cause a large disparity in search time depending on the implementation of the sort algorithm.

With our new sort comparator, the subsets are still sorted by sum in decreasing order, but ties are broken with smaller cardinality subsets coming first. The rationale is that with fewer elements filling the current bin, there is more flexibility filling the remaining bins. If both the sum and cardinality are equal, the subset with the smallest unique element comes second. For example, for $A = \{9, 7, 4, 1\}$ and $B = \{9, 7, 3, 2\}$, B comes before A.

Limited Discrepancy Search (LDS)

We have also implemented limited discrepancy search [Harvey and Ginsberg, 1995] which searches for solutions that mostly agree with heuristic choices before solutions that disagree. In our case, the heuristic choice is the value ordering of bin completions. Suppose we are searching a complete binary tree with each left branch corresponding to a heuristic choice and each right branch corresponding to a discrepancy. Standard DFS explores the entire left subtree of the root before any nodes in the right subtree. The path that follows the left branch at the root then right branches down to a leaf node has $d-1$ discrepancies where d is the depth of the tree. The path that follows the right branch at the root, then left branches to a leaf node has only one discrepancy. Nonetheless, DFS searches the path with $d-1$ discrepancies before searching the path with only one. LDS orders the heuristic search by number of discrepancies. Since we do not have a binary tree, we treat either a reduction in sum or an increase in cardinality from the previous branch as an additional discrepancy. This technique only helped with the DIMM dataset of Table 3 and so it is turned off for the other experiments.

4 Branch-and-Cut-and-Price (BCP)

Branch-and-cut-and-price is a method for solving integer linear programs (ILP) with an exponential number of variables. BCP is a combination of linear programming [Chvatal, 1983], branch and bound, cutting planes [Gomory, 1958] and column generation [Gilmore and Gomory, 1961].

The LP model for bin packing is based on the Gilmore and Gomory [1961] model proposed for the cutting stock problem, a problem closely related to bin packing, the difference being in spirit. Bin packing assumes that there are relatively few duplicate elements in the input set S, while the cutting stock problem expects many duplicates.

BCP has traditionally been used to solve the cutting stock problem. In the past 15 years, many groups have applied BCP to bin packing as well, for example, see [Vanderbeck, 1999; Valério de Carvalho, 1999; Belov and Scheithauer, 2006].

4.1 Gilmore and Gomory Bin-Packing Model

For input elements $S = \{s_1, ...s_n\}$, bin capacity C and m feasible packings, the following is the Gilmore and Gomory linear programming bin-packing model:

Minimize	**Subject to**
$\sum_{j=1}^{m} x_j$	$\sum_{j=1}^{m} a_{ij}x_j \geq 1, \quad \forall i \in \{1, ..., n\}$

where:

- x_j is 1 if the set of elements in feasible packing j are assigned to a bin, 0 otherwise.
- a_{ij} is 1 if item i appears in packing j, 0 otherwise.

The objective function minimizes the number of packed bins. The constraints require that each element is assigned to at least one bin. In order to be feasible, packing j must satisfy the following constraint that the sum of the elements in the packing is not greater than the bin capacity:

$$\sum_{i=1}^{n} s_i a_{ij} \leq C, \quad \forall j \in \{1, ..., m\}$$

We test bin completion against Gleb Belov's BCP model because he graciously provided his source code to us, and as far as we are aware, it is considered the current state of the art for bin packing. Belov's algorithm is a BCP algorithm using Gomory mixed-integer cuts. There are many details to Belov's algorithm beyond the scope of this paper. We refer the reader to [Belov and Scheithauer, 2006] for details.

5 Number Partitioning and Bin Packing

Recall that given S and $k \geq 2$, the number-partitioning problem is to find the smallest capacity C such that the elements of S can be packed into k bins. Coffman, Garey and Johnson [1978] proposed an approximation algorithm for solving number partitioning using bin packing and binary search. Dell'Amico *et al.* [2008] proposed an optimal algorithm using BCP as the bin-packing solver.

We can solve a number-partitioning problem on S by solving a sequence of bin-packing problems on S, varying the bin capacity C. We search for the smallest C such that the optimal number of bins required is equal to k, and call this smallest capacity C^*. Starting with lower and upper bounds on C^*, we perform a binary search over the bin capacities between these two values. At each probe of the binary search, we solve a bin-packing problem with fixed capacity C. This can be done with any bin-packing solver. If the optimal solution requires fewer than k bins, the lower half of the remaining capacity space is searched, and otherwise the upper half is searched. Using Improved BC as the bin packer, we call this algorithm BSIBC (binary-search improved bin-completion). With BCP as the bin packer, we call it BSBCP.

5.1 Lower Bounds for Number Partitioning

Dell'Amico and Martello [1995] define three lower bounds for number partitioning. L_0: Relax the constraint that an item cannot be split between multiple bins. L_1: C^* must be at least as large as the maximal element of S. L_2: Relax the problem by removing all but the largest $k+1$ input elements. The smallest remaining are the k^{th} and $k+1^{st}$ elements, so they must go in the same bin. Assuming S is in decreasing order:

$$L_0 = \left\lceil \frac{\sum_{i=1}^{n} s_i}{k} \right\rceil$$
$$L_1 = \max\{L_0, s_1\}$$
$$L_2 = \max\{L_1, s_k + s_{k+1}\}$$

5.2 Upper Bound for Number Partitioning

We use a very simple upper bound on C^* called longest processing time (LPT) [Graham, 1966]. It sorts the input elements in decreasing order and assigns each element to the subset with smallest sum so far. Please refer to [Dell'Amico *et al.*, 2008] for a more complete treatment of upper bounds.

6 Experiments

There have been a number of published benchmarks for bin packing. One common feature of the benchmarks we have encountered is the number of elements per bin is relatively small. For example, the triplet set of [Falkenauer, 1996], the hard28 set of [Schoenfield, 2002], and experiments by [Fekete and Schepers, 2001], [Martello and Toth, 1990b] and [Korf, 2003] all average three elements per bin or fewer. In fact, [Falkenauer, 1996] claims, "Similarly, BPP instances are easier to approximate when the number of items per bin grows above three, so 'triplets' are the most difficult BPP instances." We disagree and show contrary results. A second common feature of these benchmarks is that the precision of the elements is relatively small, with the input elements being sampled from the ranges $[1, 10^2]$, $[1, 10^3]$ or $[1, 10^4]$.

It is not obvious how to generate good benchmarks for bin packing. Most randomly generated bin-packing problems have equal lower and upper bounds and thus are trivially solved without search. Our experiments[2] suggest that the difficulty of a bin-packing instance is dependent on the number and precision of the input elements, the number of bins in an optimal solution, and the amount of empty space in an optimal packing. We also find that problems which are easy for BC can be hard for BCP and vice-versa. The following experiments were run on an Intel Xeon X5680 CPU at 3.33GHz. The IBC algorithm is run with the improved sort comparator, a completion buffer of size 50 but no LDS (see section 3.4). Korf's BC implementation generates segmentation faults on our benchmarks because it uses a fixed size buffer for bin completions and the sometimes exponential number of bin completions overflows this buffer. To approximate BC, we use our implementation based on Korf's with a buffer of size 1,000,000 and the original sort comparator. For all experiments, the algorithm with the best time for a particular set of instances is shown in bold.

[2]All our benchmarks can be downloaded from http://www.cs.ucla.edu/~korf/binpacking/benchmarks

6.1 Bin Packing Results

Table 1 shows results for a set of bin-packing experiments comparing both the original and improved BC algorithms as well as Belov's BCP algorithm. All the instances have $n = 100$ input elements. We tested three bin capacities, $C \in \{10^4, 10^6, 10^{15}\}$ and sampled the elements of the problem instances uniformly at random over various ranges. After generating an instance, we computed both the L_2 lower and BFD upper bounds. If these values were equal, thus making the problem trivial, we rejected that instance. We rejected between 88.56% and 99.96% of all generated problems depending on the experimental parameters. For each combination of range and bin capacity, we generated 100 non-trivial problem instances and gave each algorithm a maximum of 24 hours to solve them. The results are reported in seconds and are the average over the 100 instances. If all problems were not solved within the 24 hours, we report a '-' in the table.

The first row reports the maximum value of the sample range of the input elements. We sample between the value 1 and this range. For example, the last column's range max is $.50C$, so we sampled in the range $[1, .50C]$. For bin capacity $C = 10^4$, this range is $[1, 5000]$. For $C = 10^6$, the range is $[1, 500000]$. The second row, $\boldsymbol{E}[B]$ reports the expected number of bins in an optimal partition given the range. $E[B] = \frac{(\text{Range Max}) \times n}{2C}$. The next three rows report the percent of randomly sampled problems which were rejected as trivial because their lower and upper bounds were equal. The next 15 rows are broken down into three groups, one for each bin capacity $C \in \{10^4, 10^6, 10^{15}\}$. In each group, there are five rows. The first row reports results for original bin completion, the second branch-and-cut-and-price, and the third improved bin completion with both incremental completion generation and the new variable order. The fourth row reports IBC with only incremental completion generation and the fifth with only the new variable order.

Our IBC algorithm dominates the original BC algorithm for all bin capacities and all ranges. The IBC algorithm is up to five orders of magnitude faster than the original algorithm for instances which the original algorithm successfully solved in time. For $\boldsymbol{E}[B]$ between 2 and 20, IBC outperforms BCP. For $\boldsymbol{E}[B] \geq 22$, IBC no longer completes all it's experiments within the 24 hour limit while BCP successfully solves all problems. There is a trend for BCP that higher precision

	Range Max →	.04C	.08C	.12C	.16C	.20C	.30C	.40C	.42C	.44C	.46C	.48C	.50C
	E[B] →	2	4	6	8	10	15	20	21	22	23	24	25
% Trivial	$C = 10^4$	99.96	99.80	99.55	99.14	98.55	96.88	93.91	92.75	91.47	89.93	90.02	86.79
	$C = 10^6$	99.94	99.78	99.65	99.25	98.75	96.80	92.49	92.88	91.69	91.28	88.12	87.71
	$C = 10^{15}$	99.94	99.78	99.59	99.21	98.49	96.53	93.81	93.61	92.44	92.11	90.09	88.56
$C = 10^4$	BC	-	-	-	-	-	-	23.73	12.71	3.863	9.909	67.82	-
	BCP	13.96	4.118	1.712	1.050	.4618	.2832	.1180	.1053	.1216	.1177	.1479	**.2359**
	IBC	**.0003**	**.0001**	**.0012**	.0011	.0013	.0018	**.0024**	**.0024**	**.0023**	**.0020**	**.0020**	-
	Incremental	.0005	.0010	.0014	**.0010**	**.0011**	**.0017**	.0146	.0852	.0042	.0852	.0221	-
	Variable	-	-	-	-	-	-	78.56	25.98	15.52	6.98	4.713	1.475
$C = 10^6$	BC	62.10	109.6	157.3	198.9	173.1	122.2	26.91	-	-	-	-	-
	BCP	56.13	13.38	4.874	4.113	3.852	4.138	6.135	**5.578**	**4.754**	**3.551**	**2.869**	**1.671**
	IBC	**0001**	.0004	**.0007**	**.0008**	.0011	.0016	**.0870**	6.792	-	-	-	491.3
	Incremental	**.0001**	**.0003**	.0008	.0010	**.0010**	**.0015**	.1899	6.797	-	-	-	494.4
	Variable	59.39	141.3	221.1	347.7	427.8	53.20	39.91	31.75	-	-	-	-
$C = 10^{15}$	BC	50.69	94.61	145.74	175.20	171.43	93.50	-	-	10.12	103.69	-	-
	BCP	187.7	53.75	29.29	23.53	19.22	12.59	8.639	7.103	5.514	3.689	**12.32**	**1.926**
	IBC	**0001**	**.0004**	**.0006**	**.0008**	**.0012**	**.0016**	6.165	**.3889**	.3029	**.3238**	-	-
	Incremental	.0002	.0005	.0009	**.0008**	**.0012**	.0017	**6.132**	**.3889**	**.2938**	.3245	-	-
	Variable	48.11	94.44	142.3	169.3	17.38	93.33	-	-	1.26	57.20	-	-

Table 1: Bin Packing Results in seconds for n=100 using improved and original bin completion and branch-and-cut-and-price.

numbers are harder to pack than lower precision. There is no clear precision trend for IBC.

However, this table does not tell the whole story. For example, consider the entry for IBC, $C = 10^6$ and $\boldsymbol{E}[B] = 25$. The average time is 491.3s though 98 of the 100 non-trivial instances are solved in less than one second. The remaining two problems take 46680.3s and 2459.5s. In fact, for all the distinct experiments in this table, IBC solved all but a few instances in less than one second. For example, for $C = 10^{15}$ and $\boldsymbol{E}[B] = 25$, 88 of the first 91 problems are solved in less than one second each. The 92nd problem took so long to solve that the experiments ran out of time. Finally, consider $\boldsymbol{E}[B] = 21$ and $C = 10^6$. The average times of IBC and BCP are 6.792s and 5.578s. While it appears their results are similar, the variance tells a different story. The sample variance of BCP is 2.750 as compared to 4,407.283 for IBC. For $n = 100$, most randomly sampled non-trivial problem instances are easy for IBC while a few show an exponential explosion. BCP has a more consistent behavior.

While some of the experiments have erratic results, most are statistically significant. We calculated a two-tailed paired Student's t-test comparing the IBC and BCP results for each value of C and $\mathbf{E}[B]$. All of the experiments have a p-value below $6.352e^{-6}$ except for (C,$\mathbf{E}[B]$) = (10^4,15), (10^6,21), (10^6,25), and (10^{15},20). These four sets of experiments have p-values of .008, .855, .297 and .674 respectively. That is to say that for these four sets of experiments, we cannot be confident that the difference in means between IBC and BCP are statistically significant. In the next section, we show how bin-packing problems generated from number partitioning have more consistent and statistically significant results.

For most of the experiments, the contribution of incremental completion generation to the effectiveness of IBC is much stronger than variable order. A notable exception is for $C = 10^4$, $\mathbf{E}[B]$ = 25. In this set of experiments, IBC with only the new variable order completes all of the experiments with an average time of 1.475s while IBC with incremental completions and variable ordering does not complete the experiments within our 24 hour time period. Also note that for $C = 10^4$ and $\mathbf{E}[B] \geq 20$, IBC with both techniques is significantly faster than with just incremental completion generation. The contribution of variable ordering is more significant in the results of table 3 to be discussed in the next section.

6.2 Number Partitioning Results

Table 2 shows results for both bin packing and number partitioning for $n = 45$. We choose 45 and not 50 as not enough number partitioning experiments were completed for $n = 50$ to show complete data. The problem instances are generated in the same manner as for Table 1, again rejecting trivial bin-packing instances. Rejecting the trivial bin-packing instances does not affect the difficulty of solving the input elements as number partitioning problems.

Both the bin packing and number partitioning algorithms use the exact same input elements. The bin-packing problems treat $C \in \{10^4, 10^6, 10^{15}\}$ as fixed bin capacities and so k is the expected number of bins in an optimal solution. The number-partitioning problems treat k as fixed and perform a binary search to find the smallest value of C which allows for k bins in the optimal solution. BSIBC and BSBCP are the binary search bin packing algorithm solvers described in section 5. They both use the L_2 lower bound of section 5.1 and the LPT upper bound of section 5.2.

Note that solving these instances as bin packing problems

		Range Max →	.08C	.13C	.17C	.22C	.26C	.31C	.35C	.40C	.44C	.66C	.88C
		k →	2	3	4	5	6	7	8	9	10	15	20
Bin Packing	C=10^4	BCP	.3631	.0596	.0336	.0247	.0243	.0236	.0239	.0304	.0359	**.0105**	.0047
		IBC	**.0001**	**.0002**	**.0002**	**.0004**	**.0005**	**.0005**	**.0006**	**.0006**	**.0055**	.0603	**.0012**
	C=10^6	BCP	3.281	.3881	.4061	.4356	.4725	.5642	.4587	.2240	.1280	**.0110**	.0046
		IBC	**.0000**	**.0002**	**.0003**	**.0003**	**.0005**	**.0003**	**.0006**	**.0005**	**.0020**	.0724	**.0011**
	C=10^{15}	BCP	6.18	1.064	1.055	1.018	1.005	.7815	.4727	.2703	.1347	**.0104**	.0045
		IBC	**.0001**	**.0003**	**.0002**	**.0004**	**.0004**	**.0003**	**.0005**	**.0188**	**.0493**	.0406	**.0006**
Number Partitioning	C=10^4	BSBCP	.4852	.1300	.0553	.1134	.1170	.1625	.2025	.2909	.5905	**.1101**	.0381
		BSIBC	**.0001**	**.0006**	**.0066**	**.0217**	**.0264**	**.0371**	**.0313**	**.0326**	**.0955**	2.475	**.0030**
	C=10^6	BSBCP	1.217	1.836	3.9105	7.188	16.28	278.0	32.53	**12.77**	**6.772**	**.2413**	.0728
		BSIBC	**.0038**	**.0467**	**.0699**	**.0702**	**.0919**	**2.337**	**8.382**	14.17	18.20	7.214	**.0033**
	C=10^{15}	BSBCP	-	-	-	-	-	-	481.9	**101.7**	**38.50**	**.8673**	.2248
		BSIBC	-	-	-	**433.8**	**247.5**	**179.4**	**159.3**	168.8	146.1	31.57	**.0042**

Table 2: Bin packing and number partitioning results in seconds for n=45 using identical sets of input numbers.

with fixed C is almost instantaneous for IBC for all instances. Belov's BCP is also very fast on average, solving all instances with $k > 2$ in less than one second. Not surprisingly, these instances are much easier than the $n = 100$ instances of Table 1. Both BSIBC and BSBCP solve all number partitioning instances with precision 10^4 on average in less than one second except for BSIBC and $k = 15$ which takes 2.48s on average.

However, the story for higher precision input elements is different. For input elements with precision 10^6, both BSIBC and BSBCP show an easy-hard-easy transition with the hardest problems appearing at $k = 10$ for BSIBC and $k = 7$ for BSBCP. For input elements with precision 10^{15}, the hardest problems occur with low k for both BSIBC and BSBCP and progressively get easier as k increases. For both 10^6 and 10^{15}, BSIBC is faster than BSBCP for $k \leq 8$. At $k = 9$, BSBCP becomes and stays faster until $k = 20$.

Finally, consider $k = 9$ and $C = 10^6$. BSIBC shows an average time of 14.17s and BSBCP an average time of 12.74s. This time, the variance is much closer. The sample variance of BSBCP is 53.03 as compared to 138.22 for BSIBC. We calculated a two-tailed paired Student's t-test comparing the BSBCP and BSIBC results for each value of C and k. This time, all of the experiments have a p-value below .000106 except for $k = 9, C = 10^6$. This experiment had a p-value of .342. This means that all of the experiments are statistically significant except for this lone exception. This stands to reason as the average times were so close to each other.

It appears that the high variance in problem difficulty we saw for bin completion in Table 1 is due to the large amount of unused bin capacity typically allowed in an optimal solution of a random instance. We can use the optimal capacity found by solving the number-partitioning problem as the bin capacity for a bin-packing problem over the same input elements to minimize the unused capacity and make the bin packing instances much harder, even for small n.

Korf [2009; 2011] presents recursive number partitioning (RNP), an algorithm which also solves the number-partitioning problem. Unlike the algorithms in this paper, RNP solves number partitioning directly instead of as a binary search over a series of bin-packing problems. While RNP is beyond the scope of this paper, we note that Korf and Schreiber [2013] show that for $k \leq 5$, RNP outperforms binary-search bin-packing approaches to number partitioning, but not for larger values of k.

6.3 DIMM

Dell'Amico *et al.* [2008] presents DIMM, a binary search branch-and-cut-and-price solver used to solve the number partitioning problem. They claim DIMM to be the state of the art for number partitioning. They explore a uniform number partitioning benchmark [3] created by Franca et al. [1994]. They start with 390 problem instances and are able to solve all but 19 of them optimally using lower bounds and approximation algorithms. They post results along with the lower and upper bounds they used for these 19 hard problems in Table 5 of their paper. We run eight versions of our BSIBC algorithm toggling between the old and new variable ordering; incremental and exhaustive completion generation (buffer=2500 and buffer = 10^6); and limited discrepancy search on and off. (see section 3.4). The leftmost of the eight BSIBC columns has all of our improvements toggled on. We requested but were unable to attain their software so we compare against their published results. To be fair, we ran these experiments on a 3.8 GHz Pentium IV which is comparable to their test machine.

Table 3 reports the results. Range is the sample range for the input elements, k the fixed number of partitions, n the number of input elements and index a unique index from their dataset given range, k and n. LB and UB are the lower and

[3] Available at http://www.or.deis.unibo.it/research.html

							BSIBC							
							New Variable Order				Old Variable Order			
							buf=2500		buf=10^6		buf=2500		buf=10^6	
Range	k	n	**index**	**LB**	**UB**	**DIMM**	**LDS**	**¬ LDS**	**LDS**	**¬ LDS**	**LDS**	**¬ LDS**	**LDS**	**¬LDS**
[1,10^2]	25	50	3	110	111	30.08	.00	.01	.00	.00	.00	.00	.00	.00
[1,10^3]	25	50	3	1,092	1,105	30.02	.01	.00	.00	.00	.00	.00	.00	.00
[1,10^3]	25	100	2	1,941	1,942	124.12	4.87	4.87	3.72	3.72	1.37	.94	1.78	1.16
[1,10^4]	5	10	6	11,385	11,575	.02	.00	.01	.00	.00	.00	.00	.00	.00
[1,10^4]	10	50	1	26,233	26,234	30.63	.17	.12	.16	0.11	53.01	33.9	59.56	37.36
[1,10^4]	10	50	3	27,764	27,765	221.70	.11	.13	0.1	.13	33.45	1.78	35.67	1.91
[1,10^4]	10	50	5	25,296	25,297	34.05	.12	.12	.1	0.09	.45	0.09	.51	0.09
[1,10^4]	10	50	8	32,266	32,267	82.13	.09	.08	0.07	.08	.19	.77	.21	.75
[1,10^4]	25	50	1	9,517	9,688	30.03	.01	.00	.00	.00	.00	.00	.00	.00
[1,10^4]	25	100	0	21,169	21,172	124.13	7.57	583.76	8.1	662.22	-	-	-	-
[1,10^4]	25	100	1	17,197	17,199	126.06	3.46	3.33	7.45	7.39	-	-	-	-
[1,10^4]	25	100	2	21,572	21,575	123.83	20.32	33.3	22.61	38.08	-	-	-	-
[1,10^4]	25	100	3	20,842	20,844	123.84	.95	1.0	0.85	.91	-	-	-	-
[1,10^4]	25	100	4	20,568	20,571	124.22	917.0	-	-	-	-	-	-	-
[1,10^4]	25	100	5	20,695	20,697	125.97	1.71	1.71	4.48	4.51	-	-	-	-
[1,10^4]	25	100	6	20,021	20,023	123.53	1.03	1.03	1.01	1.03	-	-	-	-
[1,10^4]	25	100	7	19,272	19,274	130.13	1.61	1.72	3.48	3.64	-	-	-	-
[1,10^4]	25	100	8	20,598	20,600	130.69	1.23	1.22	1.58	1.59	-	-	-	-
[1,10^4]	25	100	9	19,124	19,127	154.48	4.35	3.38	4.53	3.79	-	-	-	-

Table 3: Number partitioning DIMM vs BSIBC.

upper bounds they use, we use the same bounds. For all but one instance ($k = 25, n = 100, index = 4$), BSIBC with all improvements outperforms DIMM, by up to three orders of magnitude.

In contrast to our earlier experiments reported in Table 1, the variable order seems to be the most important improvement here. None of the hardest experiments with range=$[1, 10^4]$ and k=25 are solved within the time limit without the new variable order. LDS has a large affect on two problems of this class, the ones with indices 0 and 4.

7 Conclusions and Future Work

We have described three improvements to the bin completion algorithm. The first enforces a different branching order. The second incrementally generates and buffers completions so the algorithm does not get stuck generating an exponential number of completions. The third, LDS, changes the search order preferring paths that mostly agree with the heuristic choice before paths that do not. The first two changes lead to five orders of magnitude speed up compared to the original algorithm. We also compare IBC to a state of the art branch-and-cut-and-price algorithm written by Gleb Belov. Over the past 15 years, most optimal bin-packing literature has focused on BCP and there have not been experimental comparisons with artificial intelligence search techniques. We have done this comparison and find that for $n = 100$ when there are no more than 20 bins in the optimal solution, the artificial intelligence algorithm IBC outperforms BCP. However, when there are more than 20 bins in the optimal solution, BCP outperforms IBC. We show that the difficulty of solving a bin-packing instance is dependent on the number of input elements n, the number of bins in the optimal solution, the precision of the input elements and the amount of unused space in the bins of an optimal solution.

For bin-packing problems with $n = 45$, both IBC and BCP solve virtually all instances in less than 1 second. We solve these same input elements as number-partitioning problems, solving for the smallest capacity with which we can pack the numbers into k bins. We used BSIBC and BSBCP, a combination of IBC and BCP with a binary search algorithm to solve these instances. We find that for $n = 45$ and $k \leq 8$, BSIBC outperforms BSBCP, for $9 \leq k \leq 15$, BSBCP outperforms BSIBC and for $k = 20$, they perform similarly. We also note that for $k \leq 5$, recursive number partitioning outperforms both BSIBC and BSBCP. Solving the problems as number-partitioning instances is up to five orders of magnitude slower than solving them as bin-packing instances. We also compared BSIBC with limited discrepancy search to DIMM, a state of the art number partitioning solver. BSIBC outperformed DIMM by up to three orders of magnitude.

From this work, it is clear that there are bin-packing instances for which branch-and-cut-and-price is best and other instances for which bin completion is best. In future work, we will explore how to integrate ideas from BCP, IBC and RNP, creating a more effective solver. We specifically will focus on techniques for more intelligently and efficiently generating completions.

8 Acknowledgements

We thank Gleb Belov for providing his BCP source code to us along with a great deal of support running his code.

References

[Belov and Scheithauer, 2006] G. Belov and G. Scheithauer. A branch-and-cut-and-price algorithm for one-dimensional stock cutting and two-dimensional two-stage cutting. *European Journal of Operational Research*, 171(1):85–106, May 2006.

[Chvatal, 1983] V. Chvatal. *Linear programming*. WH Freeman, 1983.

[Coffman Jr *et al.*, 1978] E.G. Coffman Jr, M.R. Garey, and D.S. Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*, 7(1):1–17, 1978.

[Coffman Jr *et al.*, 1996] E.G. Coffman Jr, M.R. Garey, and D.S. Johnson. Approximation algorithms for bin packing: A survey. In *Approximation algorithms for NP-hard problems*, pages 46–93. PWS Publishing Co., 1996.

[Dell'Amico and Martello, 1995] M. Dell'Amico and S. Martello. Optimal scheduling of tasks on identical parallel processors. *ORSA Journal on Computing*, 7(2):191–200, 1995.

[Dell'Amico *et al.*, 2008] M. Dell'Amico, M. Iori, S. Martello, and M. Monaci. Heuristic and exact algorithms for the identical parallel machine scheduling problem. *INFORMS Journal on Computing*, 20(3):333–344, 2008.

[Eilon and Christofides, 1971] S. Eilon and N. Christofides. The loading problem. *Management Science*, 17(5):259–268, 1971.

[Falkenauer, 1996] E. Falkenauer. A hybrid grouping genetic algorithm for bin packing. *Journal of heuristics*, 2(1):5–30, 1996.

[Fekete and Schepers, 2001] S.P. Fekete and J. Schepers. New classes of fast lower bounds for bin packing problems. *Mathematical programming*, 91(1):11–31, 2001.

[França *et al.*, 1994] P.M. França, M. Gendreau, G. Laporte, and F.M. Müller. A composite heuristic for the identical parallel machine scheduling problem with minimum makespan objective. *Computers & operations research*, 21(2):205–210, 1994.

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

[Gilmore and Gomory, 1961] P.C. Gilmore and R.E. Gomory. A linear programming approach to the cutting-stock problem. *Operations research*, 9(6):849–859, 1961.

[Gomory, 1958] R.E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64(5):275–278, 1958.

[Graham, 1966] R.L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.

[Harvey and Ginsberg, 1995] W.D. Harvey and M.L. Ginsberg. Limited discrepancy search. In *International Joint Conference on Artificial Intelligence*, volume 14, pages 607–615. Lawrence Erblaum Associated ltd, 1995.

[Johnson, 1973] D.S. Johnson. *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

[Korf and Schreiber, 2013] Richard E. Korf and Ethan L. Schreiber. Optimally scheduling small numbers of identical parallel machines. In *Proceedings of the 23nd International Conference on Automated Planning and Scheduling (ICAPS 2013) Rome, Italy*, 2013.

[Korf, 2002] R.E. Korf. A new algorithm for optimal bin packing. In *Proceedings of the National conference on Artificial Intelligence*, pages 731–736, 2002.

[Korf, 2003] Richard E. Korf. An improved algorithm for optimal bin packing. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03) Acapulco, Mexico*, pages 1252–1258, 2003.

[Korf, 2009] Richard E. Korf. Multi-way number partitioning. In *Proceedings of the 20nd International Joint Conference on Artificial Intelligence (IJCAI-09)*, pages 538–543, 2009.

[Korf, 2011] R.E. Korf. A hybrid recursive multi-way number partitioning algorithm. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume One*, pages 591–596. AAAI Press, 2011.

[Martello and Toth, 1990a] S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations*. Chichester: John Wiley & Sons, 1990.

[Martello and Toth, 1990b] Silvano Martello and Paolo Toth. Lower bounds and reduction procedures for the bin packing problem. *Discrete Applied Mathematics*, 28(1):59–70, 1990.

[Schoenfield, 2002] J.E. Schoenfield. Fast, exact solution of open bin packing problems without linear programming. *Draft, US Army Space & Missile Defense Command*, page 45, 2002.

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

[Vanderbeck, 1999] F. Vanderbeck. Computational study of a column generation algorithm for bin packing and cutting stock problems. *Mathematical Programming*, 86(3):565–594, 1999.