University of California

Los Angeles

Optimal Vector Packing in Multiple Dimensions with Heuristic Search

A thesis submitted in partial satisfaction

of the requirements for the degree Master of Science

in Computer Science

by

Bryan G. Simpkins

2012

Abstract of the Thesis

Optimal Vector Packing in Multiple Dimensions with Heuristic Search

by

Bryan G. Simpkins

Master of Science in Computer Science

University of California, Los Angeles, 2012

Professor Richard E. Korf, Chair

We study an extension of the well-known bin-packing problem to multiple dimensions, resulting in the "vector packing" problem. The problem is to find the minimum number of multidimensional bins to pack a set of vectors into without exceeding the bin capacity in any dimension of any bin. While we use approximate methods to inform our search, we ultimately return optimal solutions. This work is an attempt to extend the results of [Kor03] to multiple dimensions and combine it with some new techniques. A hybrid DFBnB algorithm which searches in two separate problem spaces is used as the final algorithm. Our hybrid algorithm also uses a heuristic to assist the search, and an approximate algorithm to initially bound the cost of an optimal solution. The resulting algorithm can be run on vector-packing problems with any number of dimensions, and good results are shown for up to five-dimensional problems. An attempt is also made to define a method to compare these results with future algorithms, as consensus on appropriate test suites for vector-packing problems seems to be lacking.

The thesis of Bryan G. Simpkins is approved.

University of California, Los Angeles

2012

To my father –

who taught me the lessons

which have mattered most to me

and has been willing to walk with me

through both the best times of my life

. . . and the worst.

Table of Contents

List of Figures

List of Tables

Acknowledgments

There are a number of people without which this thesis would not have been possible. First, I would like to thank my advisor, Dr. Richard Korf, for his patient guidance, his willingness to share ideas, and the editing and drive he brought to making this thesis better. I am also grateful for his prior research in bin packing which inspired me to consider the multi-dimensional approach taken in this thesis. Education is a collaborative effort, and I have greatly appreciated the guidance and support of my committee members Professors Adnan Darwiche and Michael Dyer during my time at UCLA. I also want to thank my father for his editing, LaTeX expertise, technical support in maintaining the linux server all the data was generated on, and tireless patience in discussing ideas. Additionally, I want to thank my mother and my sister for all of the little miracles that have made graduate school possible for me: from helping me put together furniture for temporary apartments to listening to me discuss my thesis and schoolwork until late at night. I am in debt to Keith Stevens for his assistance in maintaining the LaTeX style files used for the UCLA computer science department. Finally, I want to thank my good friend Henry Hsieh for being available throughout my writing process to discuss ideas.

SECTION 1

Introduction to Vector Packing

The problem of vector packing can be formally described as follows. We want to find a way to “pack” a list of items into bins such that we use the minimum number of bins. Each item is represented as a vector where the components of the vector correspond to how much space the item takes up in that dimension. The list of items that we need to pack is of finite size. We are given an infinite set of bins represented by vectors where each component represents the capacity of the bin in each dimension. A solution is a finite set of bins and a mapping of each item into one of these bins. In this thesis, we are looking for a solution that uses the minimum number of bins. We call such a solution an optimal solution. This solution is not necessarily unique; there may be multiple optimal solutions of equal cost.

As a simple example, consider a problem with the two-dimensional items (2,63), (1,37), (8,27), and (5,28) with bin capacities of (10,100). First, one scales the first dimension by multiplying by ten so that both dimensions now have a capacity of one hundred. We denote a solution by showing all of the items packed into a given bin by enclosing them in curly braces. Two possible optimal solutions are {(20,63), (80,27) BinSum: [100, 90]}, {(10,37), (50,28) BinSum: [60, 65]} and {(20,63), (50,28) BinSum: [70, 91]}, {(80,27), (10,37) BinSum: [90, 64]}. Both solutions have a cost of two bins, so the two solutions are both optimal. Any solution to our modified problem can be converted into a solution of the original problem by simply dividing the first dimension by ten. The problem of finding an optimal solution automatically is the subject of this thesis.

Throughout the packing process, it is important that the *space constraint* not be violated. The space constraint states that the sum of all vectors packed into the bin may not exceed the bin capacity for any dimension. Thus, it is not always possible to place a given item into a particular bin, as the bin may be too "full" already. Additionally, while practical problems often have different units for each of the dimensions, it is possible to map any such problem into a problem where all capacities are the same as shown in the previous example. As a result, one typically speaks of the "bin capacity" as a scalar quantity that is identical for all dimensions.

1.1 Motivation and Applications

Vector packing has a number of applications. In the paper industry, for example, paper comes on a long roll called a "stock" roll. Each client generally wants smaller rolls than a stock roll, so a series of orders for smaller rolls must be filled, preferably using as few stock rolls as possible. This is both a one-dimensional vector-packing problem and a standard bin packing problem with stock rolls acting as bins and client orders for smaller rolls being items to pack into bins. The optimal vector-packing solution will be the minimum number of stock rolls required to fill all client orders. This problem generalizes to several other industries, where one might instead be cutting pipes or bars of various lengths and numerous related problems.

Practical multidimensional vector-packing examples are slightly more difficult to come by. An application from the computer industry explored by [CMB09] is the server consolidation problem. It is not uncommon for smaller companies to buy servers for each new application they need, such as a mail server, a server for financial data, and perhaps a server for an application particular to the business. This can become costly and unruly to maintain, so many companies decide to merge the applications onto a smaller number of newer, larger servers, particularly as virtualization has become more feasible. If one represents each application as a vector where the components are the required memory, disk space, cpu resources, and required network bandwidth on a virtual machine, the applications become items to pack, and the servers become bins in a vector packing problem with dimensions corresponding to their available memory, disk space, cpu resources, and network bandwidth. A similar task-scheduling application for vector packing is developed by [BS96]. Their formulation of the problem is within the context of a multicomputer framework, where there exists a pool of computing nodes and a bus that connects them. A set of tasks may arrive where each has unique application-specific resources that must be filled. In a fashion similar to server consolidation, these resources might represent hard drive space, cpu time, memory, or various other qualities a computing node possesses. The problem is to construct an assignment of nodes to tasks such that all tasks are completed with the minimum number of computing nodes used. For an additional example of a vector-packing application, the reader is directed to [CHP05] for an industrial application involving packing steel coils into special containers called "cassettes."

1.2 Prior Work

In one dimension, the problem has been quite well studied. An excellent early reference for the problem comes from [GJ79]. In general, research results on the problem divide into creating good approximate solutions to the problem and finding optimal solutions. Much of this thesis is an attempt to extend the one-dimensional algorithms of [Kor02, Kor03] to find optimal solutions in multiple dimensions, and these results are discussed in depth in the remaining sections. An earlier attempt at one-dimensional optimal bin-packing [MT90a, MT90b] created the one-dimensional version of the heuristic used in this thesis and is discussed in the subsection 2.3 below. Multidimensional optimal vector-packing results are less common, although [CT01] compares a number of existing optimal algorithms and develops several novel algorithms. In our results section (Section 5), we attempt to compare our results with [CT01] and to create a basic testing procedure that allows rough comparison between algorithms in the future.

Vector packing is not as well studied as problems which involve orientation of spatial objects, such as rectangle packing. Much of the work in vector packing finds approximate solutions, many of which involve linear programming techniques (for a reference, see [BCS06]). A recent paper produced a linear-time approximation to the problem [OPR11], but unfortunately it did not perform better than a best-fit approximation for this application.

1.3 Vector Packing as a Search Problem

This problem can be formulated as a search problem in a finite tree of states. A state represents a partial or complete packing of vectors. In a state, there exists a list of vectors representing the items remaining to be packed, a list of bin vectors representing all the current partially-packed bins, and a list of assignments of vectors to bins associated with each bin. In the initial state, all the items are in the list of vectors to be packed, there are no bins that have been packed even partially, and no assignments have been made. One creates new states via a series of legal operators. Each operation is an assignment of some nonempty set of vectors to either a new bin or an existing bin with space. All of the vectors packed into a bin are summed as vectors and become the container vector for a new bin. None of the assignments may violate the space constraint, e.g. the set of items assigned to a bin must fit completely into a bin. Another constraint is that an item may only go into one bin; we will call this the *discrete items constraint*.

A complete packing is reached whenever the list of items to pack becomes empty, as all the items will have thus been packed. The cost of a packing is simply the number of bins in the list of partially-packed bins. The problem can thus be stated as finding the minimum-cost packing using only legal operations from the initial state. This becomes a search in a tree where the edges represent legal operations that assign a set of items into a new bin, and the nodes represent problem states or partial packings.

There are at least two different problem spaces that can be searched which are differentiated by the kind of assignments which are allowed. In the item assignment problem space, a legal assignment assigns exactly one item to be packed to either an existing bin into which it fits or a new bin. In the bin-completion problem space, a legal assignment consists of the entire set of vectors which are packed into a new bin. After an assignment is made in the bin completion problem space, no new items are assigned to that bin. Both problem spaces are described in more detail below.

1.4 Item Assignment Problem Space

The item assignment problem space is the more intuitive of the two problem spaces, as it coincides with the way most people actually pack things. Each item is packed one at a time until all items have been been packed into some bin. The search branches on each assignment of an item to a bin, exploring packing the item into each bin into which it fits, as well as the possibility of adding a new bin to the list of bins and packing the item there.

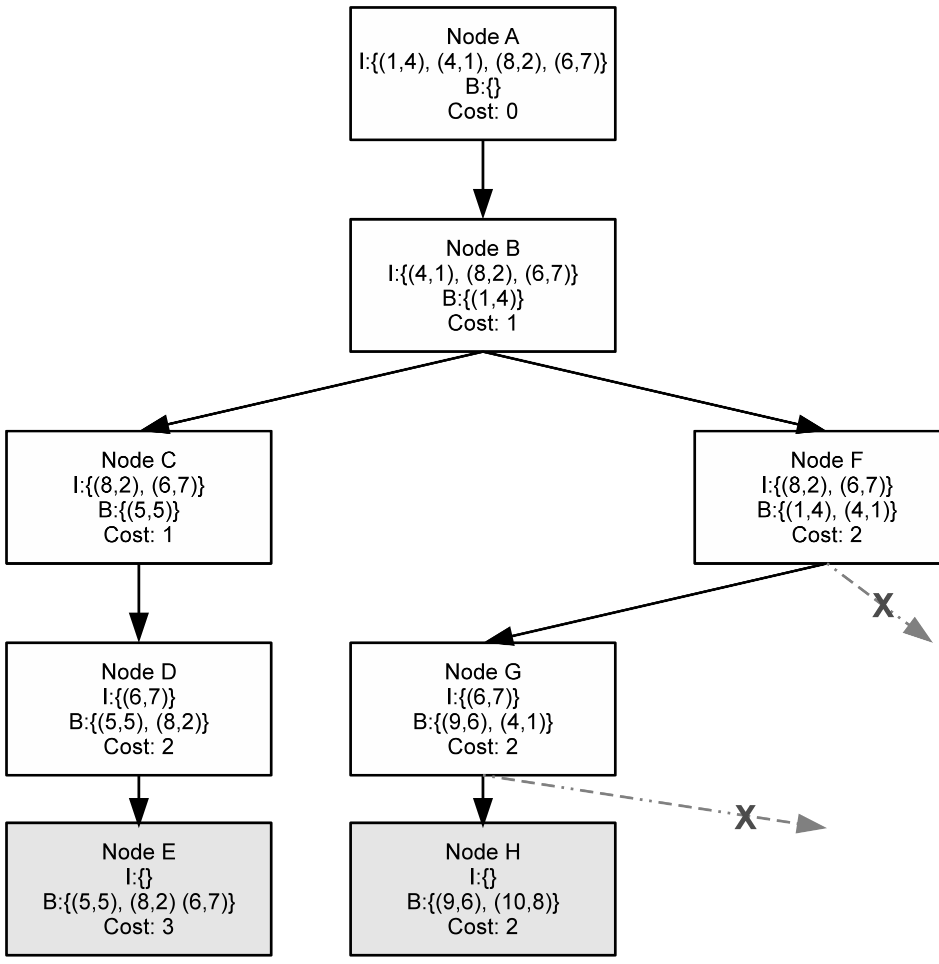

Figure 1.1: Item Assignment Example

Consider the following two-dimensional example problem, also detailed in *Figure 1.1*. The initial state is simply the set of items given by the problem, in this case the four vectors (1,4), (4,1), (8,2), and (6,7), and an empty bin list. All bin capacities are assumed to be

(10,10), terminal states are in shaded boxes and contain their cost, and arrows represent possible packings into each new bin and connect problem states. For now we will leave the vectors in their initial order, but later ordering will be shown to be an optimization.

To begin the search, the first item (1,4) is removed from the vector list and placed into a new bin by itself, since no partially-packed bins exist yet. The next item, (4,1), can either be placed into the bin with (1,4) or into a bin by itself. We represent this choice of assignment with arrows leading to the two possible assignment states. Heuristically preferring to pack items in existing bins, we first explore node C which places (1,4) in the existing bin and increases the vector representing how packed the bin is to (5,5). Our next item, (8,2), does not fit into this bin and thus must be placed into a bin by itself. Similarly, the final item (6,7) does not fit within either of the two existing bins and is thus placed into a third new bin by itself. Node E is a terminal state in this tree and represents a complete packing, as all of the vectors are now packed. To denote this, the node is shaded and its cost of 3 bins is displayed with the node. Since this node is the first terminal state reached, we initialize the best solution found so far to a cost of 3 bins.

The search then continues by returning to our first branch point and making a new choice. This occurred at node B. The search now expands the right child and places (4,1) into a bin by itself, generating node F. The next item (8,2) fits into the same bin as (1,4) or into a bin by itself. Exploring node G first, we place it into the same bin as (1,4) and have a resulting bin vector of (9,6). The final item of (6,7) fits into the same bin as (4,1) or into its own bin. We place this item into the same bin as (4,1) and end up with a bin of (10,8) and a new terminal node; node H. Node H uses 2 bins, which is fewer bins than our previous best solution found so far at node E, so we update the best solution found so far to have a cost of 2. At this point, the search returns to node G and considers placing the (6,7) item in a bin by itself.

Now this whole "branch" of the search tree contains a complete solution which uses only two bins at node G. Any solution generated below this point of the tree must use at least 2 bins. Instead of continuing the search by placing (6,7) into its own bin and exploring extraneous solutions, that branch of the tree is "pruned" and the search explores another option, in this case returning to node F. However, at node F, we again notice that there are 2 used bins and we already have a solution that uses 2 bins, so this option is pruned as well and the search returns to node B. Having explored both children of node B, the search returns to node A and finds that all options have been explored, so our saved best solution so far is returned as the optimal solution, which is represented by node H.

This is a description of a basic depth-first branch and bound (DFBnB) search of the space. A DFBnB search begins as a simple depth-first search where one explores all the way down to a complete packing before one expands other children of a node. As soon as a solution is found, however, the search creates a bound that represents the best solution found so far. In the course of any partial packing, if the current number of bins used equals or exceeds the bound, that entire branch of the search tree is "pruned," or skipped. This cannot affect the optimal number of bins returned, as a solution was already found that was at least as good as any solution possible on the path we were considering.

A useful property of a DFBnB search is that it is an "anytime" algorithm; that is, if one stops the search early due to a time constraint, DFBnB can simply return the best solution found so far. This means that it can be used as an approximate algorithm if one doesn't have enough time to find an optimal solution. Any search algorithm in this problem space likely will have its core be a DFBnB with modifications that allow for better pruning of nodes, reductions in the branching factor, or modifications to the problem space that make it smaller.

1.5 Bin Completion Problem Space

The bin completion problem space is an idea we derive from [Kor02]. Instead of packing one item at a time, the bin completion problem space packs an entire bin at a time. A valid assignment consists of a set of all items that one wishes to pack in a new bin. After a bin is packed with some items on a given branch, no new items are placed in that bin. The search proceeds by packing sets of items into bins until all the items are packed, then returns to branch on alternate sets of items that could be packed into a given bin. Instead of packing one item at a time, we pack one bin at a time.

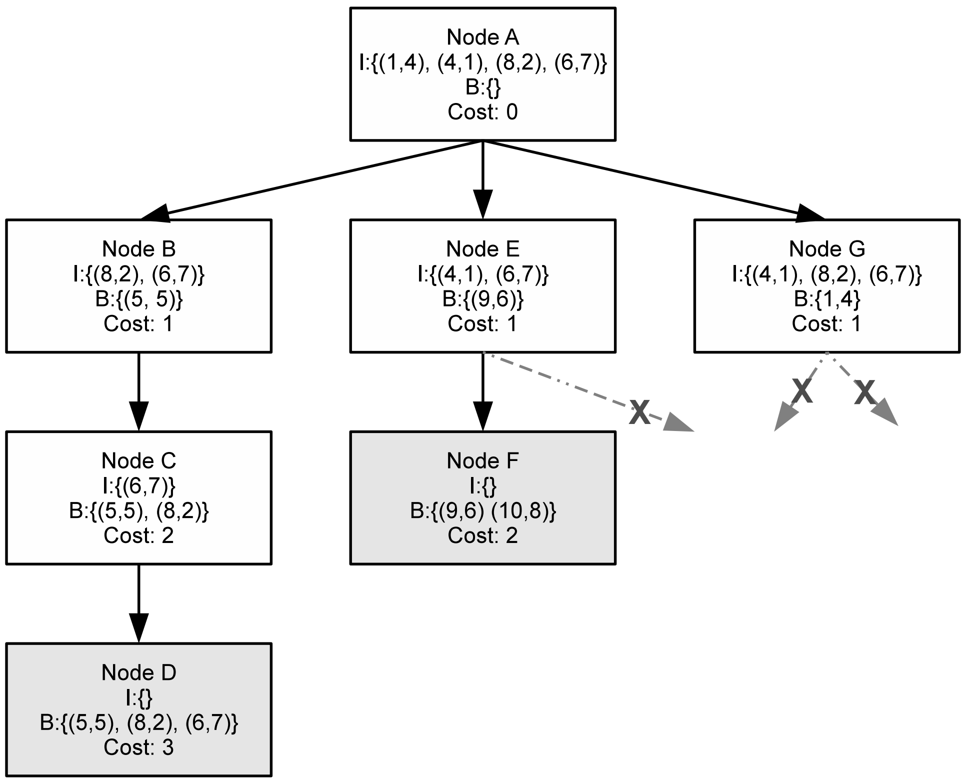

Figure 1.2: Bin Completion Example

The same two-dimensional example used to explain the item-assignment problem space is explored here with *Figure 1.2*. The initial state is identical to the initial state in the item-assignment problem space. To help avoid some branching in the search, since each vector must be placed in some bin in every solution, all completions of a given bin will have the same initial item packed in the bin. For the first bin, three possible sets can fit: either (1,4) by itself, (1,4) with (8,2), or (1,4) with (4,1). Exploring the figure from left to right, we first move to node B and place items (1,4) and (4,1) together resulting in a bin vector of (5,5). The next two items cannot fit together, so we move through two additional nodes and end up with a node D as a terminal node with cost 3, since 3 bins are used in this complete packing. DFBnB is still used here to help prune the search space, so we initialize our best solution found so far to node D with a cost of 3.

Returning all the way to node A, the next branch we explore moves to node E, placing items (1,4) and (8,2) together for a resulting bin vector of (9,6). The next bin could contain either (4,1) by itself or (4,1) together with (6,7). We explore placing the two remaining items together in a bin first, resulting in a terminal node at node F with bin vectors of (9,6) and (10,8). This node has cost 2, which is better than our existing best cost of 3, so we update our best solution found so far to node F and return to our first branch point at node E. For the bin completion problem space, each level of the tree adds a new bin and thus increases the current solution cost. Since the current solution has a cost of 1 and we already have a solution of cost 2, any children of node E can be at best of equal cost to our known solution and they are thus pruned from consideration. The search thus returns to node A and explores node G, placing (1,4) in a bin by itself. At this point, the current cost of the partial solution is 1 and the packing is not complete, so all children of node G are pruned. All of the children of the root node have thus been explored, so the search terminates and returns node F as the optimal solution with a cost of 2 bins.

This example highlights several important features of the search trees generated by this problem space. The trees are of finite depth; in the case of the item-assignment problem space the trees are always of a depth equal to the number of items to be packed. In the bin-completion problem space, the trees are of a depth at most equal to the number of bins required by the initial approximate solution (see Section 2 below. The branching factor of the tree varies with the item to be packed; while some items can only be packed by themselves, some items can be packed with many other items, leading to a high branching factor. Later in the thesis we will explore ways to reduce this branching factor by attempting to remove redundant or inferior packings from consideration. Many of these methods are more effective in the bin completion problem space, however. The key advantage that bin completion has as a problem space is that the final packing for each bin is known as soon as the bin has any items in it. This allows more aggressive analysis of a partial solution, as will become apparent in a later section on dominance checking (see subsection 3.3). For the remainder of the thesis, the bin completion problem space is the default search space used, as it will prove to scale better with the number of items. For small numbers of items in higher dimensional problems, the item-assignment problem space is more efficient, however. Accordingly, a "hybrid" algorithm is developed which runs the bin-completion problem space algorithm until the number of items falls below an experimentally determined threshold value for a given dimension, and then finishes each problem with a search in the item-assignment problem space. The hybrid algorithm is discussed in more detail in a later section (Section 4).

1.6 Overview

The remainder of this thesis is divided into four main sections. Section 2 is devoted to an explanation of all of the various forms of preprocessing used in our algorithms that can improve performance in any search solution to vector packing. After the preprocessing techniques are established, Section 3 defines the core of the actual search algorithm used in this thesis. Section 4 develops a hybrid algorithm that improves upon the core algorithm. Finally, Section 5 discusses how our experimental results were obtained, the actual results themselves, and the conclusions we draw from them. After the final section, there is a brief appendix which discusses how our random trials were generated.

SECTION 2

Preprocessing

2.1 Variable Ordering

It turns out that packing “larger” objects first is extremely helpful in reducing the runtime of most searches, as one gets good answers earlier in the search for use in pruning the tree throughout the remainder of the search. In addition, packing these larger items earlier in the search generally causes pruning to occur on higher branches of the search tree, also acting to reduce the runtime. This is consistent with most individual's experiences when packing a car for a long trip; if you first pack the larger items, it's much easier to fit the smaller items in the spaces which remain than it is to first pack the small items and try to fit in the large items later. The natural question that arises in multidimensional vector packing is how one ought to calculate “larger” with vector items as opposed to the simple integer comparison possible in one dimension. Two natural orderings are max dimension ordering and sum dimension ordering. In *max dimension ordering*, one insists that the vector with a higher value in any dimension should come first. For example, if one compares the vectors (1,9) and (6,6), (1,9) should come before (6,6) due to having a higher maximum value of 9 along any dimension. In *sum dimension ordering*, one instead orders vectors instead by the sum of all of their dimensions. In the above example, (6,6) would precede (1,9) due to its higher sum of all dimensions (12 versus 10).

These orderings are used for sorting the vectors in multiple contexts. In the case of preprocessing, the entire list of vectors is initially sorted using the variable ordering strategy. Several other places in the search use the same idea, including the order that children are explored after being generated below a given node. Additionally, when we need to order the set of bins, we order them using sum dimension ordering on the sum of all items packed into a bin. Initial testing suggests that both orderings get very similar results, even as the number of items or dimensions increases. For the remainder of this thesis, we use sum dimension ordering, as it appeared to have slightly superior runtimes in more of our tests than max dimension ordering.

2.2 Initial Approximation: Best-Fit Decreasing

Since we are using a DFBnB search, starting the search with a good solution can reduce the overall runtime if there is a fast way of generating such solutions. The key features of the approximation should be a fast runtime so that the approximation doesn't take too much time away from the search, and that the approximation be as close in cost to the optimal solution as possible. There are several papers focused on generating good approximate solutions to vector packing, and any approximation could be useful here. Generally there is a tradeoff in how close a solution comes to the optimal solution and the amount of time the approximation takes to return an answer. However, one algorithm in particular sticks out for this application as being extremely easy to implement, very fast, and capable of generating fairly good quality results. This algorithm is called best-fit decreasing and works as follows. For each vector in the search, the vector is placed into the bin in which it fits “best.” In one dimension, this is simply the bin that currently has the most capacity filled in which the item still fits. In multiple dimensions, “fullness” is determined by an ordering among the bins such as sum dimension ordering, and the item is placed into the fullest bin in which it fits. The fullest bin in which an item fits is found by a simple linear search, and the item placed in the bin. Then the bin is moved to its new sorted position in the list, which is also found by a linear search. Both searches can instead be implemented as binary searches, but we implemented this and found little change in the overall runtime.

In an attempt to find a better approximation, the algorithm described by [OPR11] was implemented. While its O(n) runtime is faster than best-fit decreasing, we were only able to get an improvement in the approximation quality vs. best-fit decreasing in 0.3% of cases.

Unfortunately, this did not increase or decrease speed significantly even when both algorithms were run and the better chosen. If run as the only approximation, this increased the overall runtime due to the poorer quality of the approximation in most cases. In the end, we chose not to use this algorithm as the approximation for any of the searches, and instead relied fully on best-fit decreasing. We also tried the more simple first-fit decreasing, where one places each item into the first bin in which it fits instead of the fullest. This approximation is no better than best-fit decreasing by itself, and doesn't improve the runtime even if both are run

2.3 Heuristics

Another notion that can be used to aid any search algorithm is that of a heuristic, which is an estimate of how costly any solution must be on the current search path. In the context of vector packing, a heuristic estimates how many additional bins must be added to pack all the remaining items. A heuristic is said to be *admissible* if and only if the estimate is a strict lower bound on the remaining cost of any solution. Put another way, a heuristic is admissible if it is a lower-bound estimate on the cost remaining along any solution path to a goal. During preprocessing, the heuristic is used to generate a lower bound on the number of bins required to solve the complete problem. This can allow us to skip the search entirely if the best-fit decreasing solution happens to use the same number of bins, and also allows us to stop a search early if we ever find a solution that is equal to this lower bound on bins, thus potentially saving a lengthy search.

A more formal definition of a heuristic typically uses a variant of the following notation, derived from discussions in [Kor11]. A *terminal node* is one where the search tree terminates because a solution has been found. For vector packing, a terminal node is defined as any state where all of the items are packed into bins in which they fit. In a search tree, a heuristic function h assigns a cost to a given node along the tree n. The heuristic cost of the path from node n to a goal state is denoted by *h(n).* If we define $h^*(n)$ to be the exact cost from a node n to a goal state, a heuristic is *admissible* if and only if for all nodes n, $h(n) \leq h^*(n)$.

For a node n in a vector packing search tree, we define g*(n) as* the number of bins used so far in the solution and *h(n)* as an admissible heuristic estimate of how many additional bins must be added to the problem in order to pack all remaining unpacked items. We prune the current branch of the search tree whenever $g(n) + h(n) \geq b$, where b is the number of bins used in the solution that we have found that uses the fewest number of bins so far.

The most basic admissible heuristic for vector packing comes from relaxing the idea that each item must be fully packed into only one bin. One can imagine that the items to be packed are liquids instead of distinct items and one can subdivide them as necessary. The minimum number of bins in one-dimensional bin packing is thus the ceiling of the sum of all the items to be packed divided by the bin capacity; one must use at least enough bins so that there is enough total space to pack all the items, even if the individual items may not be packable into that many bins. While this heuristic is a reasonable initial estimate, there are many examples that require more bins than what this heuristic would suggest. A simple concrete example from one-dimensional bin packing is three items of size 6 and a bin capacity of 10. While the sum of the items (18) suggests that only two bins are necessary, clearly three bins are required, as no two items can be placed into the same bin. To generalize this heuristic to multiple dimensions, one simply computes this sum and resulting minimum for each dimension separately, and takes the largest minimum number of bins for any dimension as the minimum number of required bins. We call this heuristic the *liquidity* heuristic.

A better but more difficult to calculate heuristic is the *wasted space* heuristic. In the previous example, no two items of size 6 fit together into a bin. In a sense, each item implies at least 4 units of wasted space, as no other item can fit into the space left over by each of the items. This idea was first proposed by [MT90a, MT90b] as the L2 lower bound. Our implementation relies on the discussion in [Kor02] of the idea. While attempts were made to create a vectorized version of the wasted space heuristic, it turns out that the additional constraints imposed by multiple dimensions causes these versions to become non-admissible. Instead, the one-dimensional wasted space heuristic is run separately for each dimension, creating a wasted space amount for each dimension. We then calculate the minimum bins required for each dimension separately, returning the maximum number of required bins for any dimension.

An example of this calculation for a single dimension is the following. Suppose the bin capacity is 100 and the items that must be packed are the items 99, 98, 97, 94, 51, 47, 5, 4, 2, and another 2. The 99 cannot be placed with any other item, so its bin will have 1 unit of wasted space. The 98 can be packed with the 2 and completely fill a bin, so those items are removed from consideration without any additional wasted space. The 97 can be only be placed with the remaining 2, so that bin will have 1 unit of wasted space. The 94 could be placed with either the 4 or the 5. While one can only fit one of these items into the bin, to avoid branching in our estimate we place as much as their sum of 9 can fit into this bin into the bin, so 6 units are placed into this bin and 3 units are carried over into the next bin. The next bin contains the 3 carried over and the item 51, so it cannot fit the 47 and has 100 - 54 = 46 wasted space. The last item goes into a new bin, and since there are no remaining items, this bin has 100 - 47 = 53 units of wasted space. The total wasted space is 1 + 1 + 46 + 53, or 101 units of wasted space. To calculate the total number of bins required to pack these objects, one sums all of the items to be packed (99 + 98 + 97 + 94 + 51 + 47 +5 + 4 + 2 +2 = 499) and the wasted space (101) and divides this total by the bin capacity of 100, taking the ceiling of the quotient to get the number of required bins. In this case, the wasted space heuristic calculates a minimum number of bins of the ceiling of 600 / 100 = 6 bins to pack the items. In comparison, the liquidity heuristic calculates the ceiling of 499/100 = 5 bins required for this example. If this were a multidimensional example, the algorithm would then be called on all remaining dimensions and the maximum number of required bins for any one dimension would be returned as the minimum number of bins required by the problem.

Since the wasted space heuristic always calculates the same number of minimum bins as the liquidity heuristic or more, it is always run as the preprocessing heuristic to generate the minimum number of bins required for the entire search. However, heuristics can also be used in the middle of a search to calculate the minimum number of additional bins required to reach a goal state. If the current number of bins used in a partial packing plus a heuristic estimate of the number of bins required to pack all remaining items equals or exceeds the best cost found so far, one can prune the current node from the search tree. While the linear cost of the wasted space heuristic is acceptable as a preprocessing step, it unfortunately turns out to be too expensive to run at every search state (see [Kor02]) and the liquidity heuristic is used instead once the search has actually begun. However, as the liquidity heuristic is run only on remaining items, the empty space already contained in packed bins is actually accounted for. Thus, as the heuristic is run on deeper and deeper states in the tree, the liquidity heuristic converges to both the wasted space heuristic results and the actual cost of solutions.

2.4 Initial Partial Packing

The final preprocessing idea which proves useful is the idea of beginning the search with as many items as possible that cannot be packed together already packed into some bin. Recall from the example in the introduction that every item ultimately must be packed into some bin, so each bin can be given an initial item. The idea is to choose the largest set of items such that no two fit together in the same bin and “prepack” these items into their own bins. This idea is best demonstrated through an example. Let us assume a three-dimensional vector-packing problem with a bin capacity of 10 and the five items (9,4,4), (3,7,3), (3,3,7), (6,0,0), and (6,0,0) to pack. The first thing to notice is that the items are already sorted in the proper ordering, whether one uses sum dimension ordering or max dimension ordering. If one attempts to pack these vectors using item-assignment without prepacking, one packs each item one at a time in the same order they are presented. (9,4,4) goes into its own bin. (3,7,3) doesn't fit with the first item, so it also goes into its own bin. Since (3,3,7) fits with the (3,7,3), we place them initially together into the same bin but retain the ability to return and place them in separate bins. The last two items cannot fit with (9,4,4) or in the bin that contains the second and third items, and so they each go into their own bins. This solution uses 4 bins and thus has a cost of 4. At this point, the algorithm returns to examine the option of putting (3,3,7) into its own bin instead of with (3,7,3). It will eventually discover the optimal solution that uses only 3 bins. Now consider what would happen if we prepacked the items first.

If we “prepack” the items, we first determine which dimension has the most items which are greater than half the bin capacity. In this case, the first dimension has three items which have values greater than five, the second dimension has one item, and the third dimension also has one item. As a consequence, the first dimension is chosen as “special” and all the items that are greater than five are prepacked into their own bins. After that, item-assignment is run as usual. At this point, items (9,4,4), (6,0,0), and (6,0,0) are all in their own bins. Since item-assignment explores packings that place items together before it places items into their own bins, after a prepacking step item assignment places items (3,7,3) and (3,3,7) in the original ordering into the bins for the identical items (6,0,0) and (6,0,0), resulting in an optimal packing with three bins as the very first packing. In other words, prepacking causes item-assignment to prune a suboptimal packing from ever being considered in this case.

In general, the prepacking helps item-assignment for multidimensional examples. It doesn't help one-dimensional item-assignment, as prepacking ends up generating the same results as normal item-assignment in one dimension. It also actually hurts bin-completion at all dimensions studied; this may be related to it altering the proper ordering for no real benefit, as bin completion prunes sub-optimal packings through a dominance check discussed in Section 3.3 below. In summary, prepacking should only be used to speed up multidimensional item-assignment runs so that both the item-assignment and hybrid algorithms complete faster.

SECTION 3

Bin Completion

3.1 Primary Search

While this algorithm is in many ways the most important part of the search, it is also not terribly complicated. The primary search in the bin-completion problem space has several parts which will be discussed in their own sections below. Whenever it is called, a new bin is packed. The next item in the list of items is placed in the bin, as this is the item which is judged largest according to whatever ordering is used. The primary search then calls the subset generation algorithm (discussed below), which will generate all possible complete packings of items into this bin, pruning some "dominated" or inferior options through the "dominance check," which is explained in Section 3.3. After the subsets have been generated, the primary search continues the search with each packing option in turn, potentially pruning a few more through "nogood pruning," which is explored in Section 3.4. Whenever a search returns from a particular packing option, the best solution found so far is updated if necessary, and the search terminates if there are either no more options left to pack the current bin, or the solution that is found happens to use the same number of bins as the minimum number of bins calculated by the heuristic.

3.2 Subset Generation

Generating all possible "feasible" (sets which fit into a bin) subsets of a set of elements can be implemented as a tree traversal. A binary tree is generated with each level corresponding to a particular element (usually the next element in the sorted order). The left branch of a node includes the element in all subsets below it, and the right branch excludes the element from all subsets below it. The left branch is only available if the element fits with the rest of the subset. The leaf nodes will have a set of included elements that collectively constitute the subset being considered. When a subset is generated completely, it is subjected to the dominance check and placed into the list of generated sets if it is undominated. Several interesting optimizations were used to speed up this tree traversal. While the subsets are being generated, a vectorized sum of all remaining elements is kept current; if the sum of all remaining elements to be considered would fit into the bin, that branch of the tree is terminated and all the items are simply placed into the subset, creating a leaf node. If the vectorized sum of all remaining elements is not more than the lower bound generated by dominance checking in at least one dimension, that branch is pruned, as every subset generated must equal or exceed the lower bound (lower bounds and dominance are discussed below). All resulting subsets from this subset generation step are subjected to the dominance and lower-bound checks to ensure that only undominated subsets will be selected for node expansion in the main search.

3.3 Dominance Checking

A key innovation from [Kor02, Kor03] is the idea of applying dominance relations as a way of pruning potential packings of a given bin. A packing or set of items is said to be dominated if some of the items in it can be swapped with either existing or added items in another set so that the number of bins remains constant or decreases. If a packing is dominated, it is pruned from consideration, as it cannot possibly lead to a solution better than a packing which dominates it. All of the following examples work both as one-dimensional and multidimensional examples.

Perhaps the most basic example is a set that could contain additional elements but doesn't, such as when items x and y would fit into the bin but only x is in the bin. As a slightly more interesting example, suppose that we have two elements x and y that sum to exactly the bin capacity. Any set which contains x but not y is dominated. For a set which contains x but not y, since the set must fit in the bin and $x + y$ is exactly the bin capacity, all elements except x in the bin must sum to y or less. Consequentially, one can swap all these items with y and end up using the same or fewer bins in the final solution. Another example involves an x such that no remaining two elements can be added together with x; in this case one ought to pack the largest single element that fits with x. As a final example, if two items x and y sum to z, and both $x + y$ and z fit separetly into a bin, z as an item would dominate the pair of items x and y. In other words, sets with fewer items dominate sets with more items. This is because there is more freedom with fewer items, as one also can pack them into separate bins. Packing one large item is harder than packing several smaller items.

Ultimately, we want to prune all sets which are dominated by any other set during subset generation. Whenever a new candidate subset is generated, it is run through the dominance check and only added as a new subset if it is not dominated. For this to work, there must be a way to check for dominance as sets are being generated. Fortunately, [Kor03] found such a method; it relies on two core ideas. The first idea is that excluding an element which could fit in a set imposes a lower bound on the final sum of items in that set so that it is not dominated by the excluded element. The second idea is an explicit check discussed below which covers all other forms of dominance. These ideas are each explored individually below.

The lower bound idea derives from the basic definition of dominance; if an item that could fit into a subset is excluded from that subset, then the sum of all elements which are placed into the subset to replace that item must exceed the excluded item in at least one dimension, or the subset will be dominated. This generates a lower bound on the final set which is equal to the excluded item's values in each dimension plus one. As an example of the lower-bound check in action, suppose that one is examining a bin in a two-dimensional bin packing problem with a bin capacity of (10,10). Suppose the initial item packed into the bin that is common to all bin completions is the item (6,4). Let us assume there are two possible bin completions for this bin, the first being the item (3,5) and the second being the pair of items (1,1) and (1,2). Technically, one can also pack (1,1) by itself or (1,2) by itself as well, but the lower-bound check will remove these completions; this should be clear from the following discussion. After the first bin completion (3,5) is considered, it is excluded from alternative completions of this bin. Thus, the lower bound for all brother nodes becomes (4,6), which is one greater in each dimension than the now "excluded" item (3,5). A completion with (1,1) and (1,2) will not be accepted, as (3,3) does not equal or exceed (4,6) in any dimension. However, (4,0) would be accepted, as it is sufficient to meet the lower bound in one dimension.

The final dominance check can be demonstrated with the following example. Let us assume that the common largest item in a bin is the item (1,7). If this two-dimensional bin packing example has a bin capacity of (10,10), the residual capacity r of the bin is the vector capacity of the bin minus the size of the largest item in the bin in each dimension, or (9,3) in this case. Let us suppose that we have already considered but excluded the item (7,2). Since this item is excluded, all undominated subsets which include this item have already been considered. In general, there might be a set of such excluded items which would contain all single remaining items less than or equal to r (in all dimensions) that are not in the set. If the subset we are considering to complete this bin is {(1,1), (4,0), (2,2)}, the vector sum of the items in this set t would be (7,3). Now consider the subset of our included items {(1,1), (4,0), (2,2)} consisting of just the items {(4,0), (2,2)}. The vector sum s of the items in this subset is (6,2). There are two conditions which are necessary for a subset to be dominated by an excluded item. First, swapping the excluded item x with the subset s must result in a feasible packing (the new packing must fit in the bin). This occurs for a given excluded item x and a given subset sum s if and only if $x - s \leq r - t$ in every dimension. If we let the excluded item x be the item (7,2), the equation becomes:

$$(7,2) - (6,2) \leq (9,3) - (7,3)$$

Which simplifies to:

$$(1,0) \leq (2,0)$$

Now since this equation is true in both dimensions, one can substitute (7,2) for the items (4,0) and (2,2) in our bin completion and get the feasible subset {(1,1), (7,2)}. The second condition is that the new feasible subset must actually dominate s. This occurs if $x - s \geq$

$(0, 0, 0, ...)$ in all dimensions. Since $(1, 0) \geq (0, 0)$ in all dimensions, the resulting subset {(1,1), (7,2)} dominates the subset {(1,1), (4,0), (2,2)}. These two checks are performed for every excluded item and every possible subset sum of the generated subset, and if it fails any check the subset is not included in the list of all subsets. Generating all possible subsets in this algorithm is done in a very similar fashion to the method described above for subset generation, and the current sum of all included items is kept up to date throughout subset generation.

3.4 Nogood Pruning

At this point, while the list of potential packings has been pruned significantly, some redundant packings will still be explored. Here, we again rely on an example described in [Kor03] and extend it to multiple dimensions. Let us suppose there are two undominated feasible completions of a given bin. As described in the initial packing preprocessing step, each bin can be uniquely identified by the first item in the bin, so all possible completions will share this item. Suppose our possible packings thus contain the items $\{w,x,y\}$ and $\{w,z\}$. Now since these are undominated completions, a number of relations must hold. It must be the case that the items x and y cannot be packed into a bin with capacity z or they would be dominated. In one dimension, this would mean that $x + y > z$, but in multiple dimensions it merely means that z is smaller than $x + y$ in some dimension. Also, since both these completions actually fit, it must be the case that both $w + x + y$ and $w + z$ are less than or equal to the bin capacity in all dimensions.

Now let us further suppose that we first completely explore $\{w,x,y\}$ as a packing and return to explore the search tree generated with $\{w,z\}$ as a packing. Somewhere in this new subtree, suppose that we find the possible packing $\{v,x,y\}$ for a different bin. Thus, in this tree we have two separate bins that are being packed with the items $\{w,z\}$ and $\{v,x,y\}$, respectively. In one dimension, one could argue that you could swap z with x and y since we know w,x, and y fit together and $z < x + y$, resulting in the packing $\{w,x,y\}$ and $\{v,z\}$. Now this packing has already been explored previously, as choosing $\{w,x,y\}$ as a packing explores all possible packings of remaining items and thus considered $\{v,z\}$ as well. This swap is valid because $z < x + y$, or more precisely z actually fits inside of a bin with capacity $x + y$. In multiple dimensions, however, it may be the case that z cannot be swapped with $x + y$, as z might be smaller only in one dimension and not in all dimensions. Thus, this swap is only valid in multiple dimensions if it turns out that z is less than or equal to $x + y$ in all dimensions. If the swap is valid, we have a new packing $\{w,x,y\}$ and $\{v,z\}$, which has been already explored when we explored the $\{w,x,y\}$ packing above this node but which uses the same number of bins. Thus, the packing $\{v,x,y\}$ is redundant and can be pruned.

The implementation for this is slightly different from the implementation in [Kor03] as one must apply a stricter condition on "Nogood" sets, as alluded to in the previous paragraph. This is implemented by keeping a primary list of nogood sets at each search level. Whenever a packing is completely explored, the subset of the packing that does not contain the common packing element originally in all bins at that level is added to a temporary nogood set list. When one explores a new packing, a new primary nogood list is created. This list contains all nogood sets that were passed from the previous level and every set in the temporary nogood list whose sum is greater than or equal to the new packing in every dimension. Put another way, the new primary nogood list will contain the previous primary nogood list and every temporary nogood that the new packing can "fit" into. If a potential packing below this node contains a nogood set as a subset, the packing is pruned as it is redundant due to the possible swap discussed above. Checking a potential packing for any subsets which are nogood can be done a number of ways; in our implementation the nogood set list is a simple list of all current nogood sets, which necessitates an O(n) walk through the nogood set list for every potential packing to check for subsets. While it is possible to avoid the O(n) via hashing and more complex data structures, this was implemented with little effect on the overall runtime; implementing no good pruning at all seems to be much more important than how it is implemented. An optimization suggested by [Kor03] was also used here and may be part of the reason the O(n) walk isn't prohibitively expensive. To keep the nogood set list from becoming too large, if any potential packing contains only part of a nogood set, the nogood set is removed from the list. This is valid, as having only part of a nogood set packed splits up the set and prevents any child node from ever containing the no good set as a subset. This helps keep the nogood set list smaller, saving both memory and time.

SECTION 4

Hybrid Algorithm

As will become evident in the results and conclusions found in Section 5, bin completion is not a clear win in all cases over even a simple DFBnB search in the item-assignment problem space. In the bin-completion problem space, at each node one chooses among children that represent all possible packings for a given bin, thus "completing" a bin at each level of the search. In the item-assignment problem space used for comparison, however, each node represents a choice of either creating a new bin to contain an item or packing the item into any of the preexisting bins into which it fits. Thus, each new state in the item assignment problem space packs a single item instead of a whole bin.

While many optimizations are possible with the bin completion problem space that are not in item assignment (such as dominance checking), the brute force search tree size on average for item-assignment is smaller. To see this, consider a simple problem with four items that all fit together. In this discussion, we choose to neglect the fact that each bin starts with an item in it for both item-assignment and bin-completion for clarity. In *Figure 4.1*, the four-item tree is shown. The leaf nodes are the final packings and are the nodes in each shaded area, namely nodes I-W. Not counting the root node, the figure shows that the four-item brute force search tree for item assignment has 23 nodes.

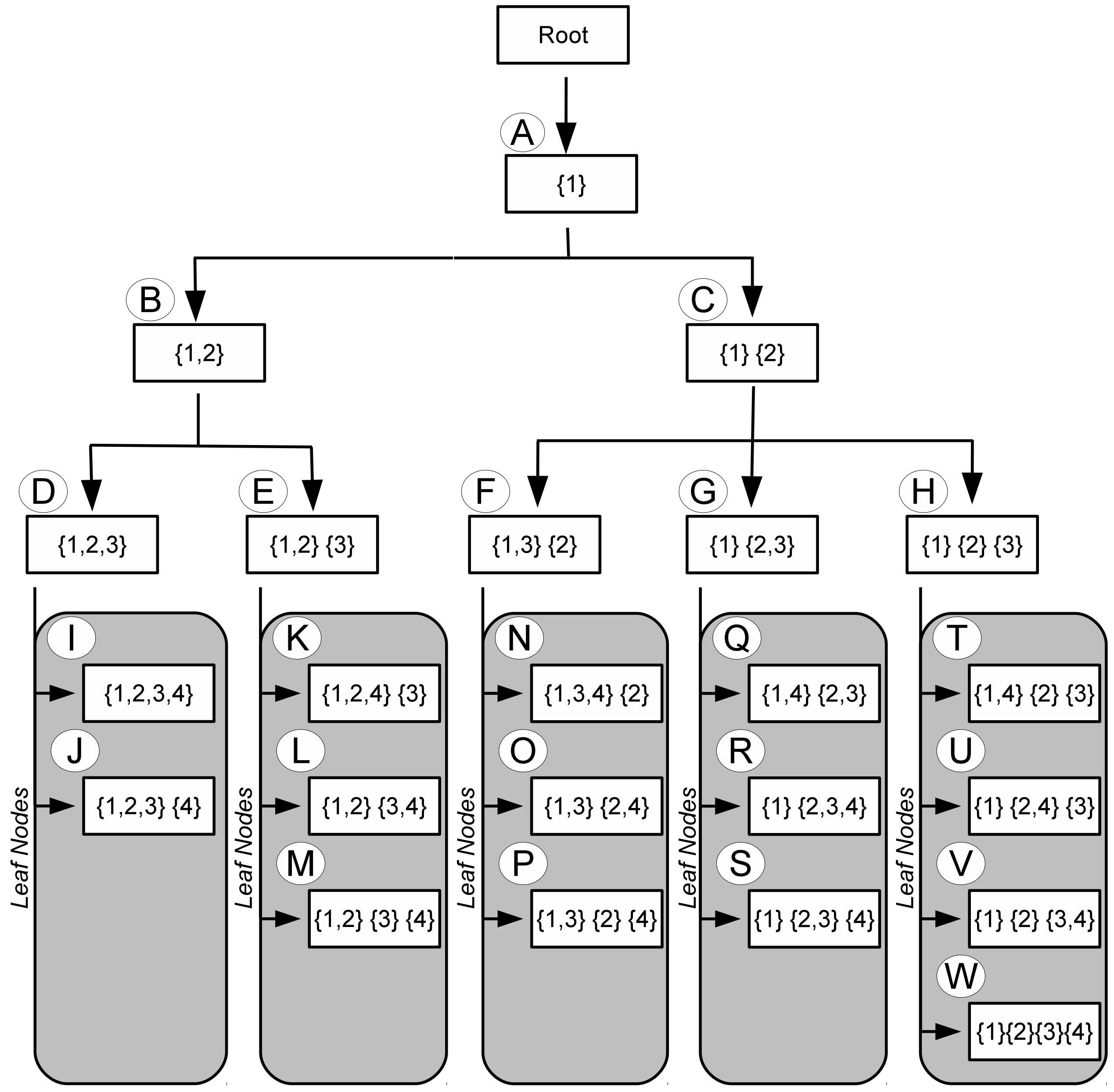

Figure 4.1: Item Assignment Brute Force Tree

For the bin completion problem space, the first level of the tree contains all ways to pack the first bin, so it has:

$$\binom{4}{1} + \binom{4}{2} + \binom{4}{3} + \binom{4}{4} = 4 + 6 + 4 + 1 = 15 \text{ nodes}$$

If one merely considers the depth two nodes that are the children of the four nodes that pack only one item into the first bin, they add:

$$\left(\binom{3}{1} + \binom{3}{2} + \binom{3}{3}\right) * 4 = (3 + 3 + 1) * 4 = 28 \text{ nodes}$$

In other words, this subset of depth two nodes in the bin-completion tree has more nodes than the entire tree for the item-assignment on this problem. Additionally, bin completion scales somewhat more poorly with the added constraints imposed by higher dimensions than item-assignment does. However, bin completion scales with an increasing number of items to pack much better than item-assignment does: the check for set dominance substantially prunes the tree.

The consequence of these effects together is that for a sufficiently small number of items that is dependent on the dimensionality, a basic item-assignment search is faster than the optimized bin completion search we have described throughout this thesis. However, when the number of items exceeds some threshold value, the bin completion search becomes faster. In order to take advantage of both algorithms, it is possible to switch between algorithms within the same search. In short, a hybrid approach is superior to either algorithm alone.

This is accomplished fairly simply within the bin completion algorithm. Whenever a new bin is packed and before the algorithm recurses with the new packing, a check is performed that checks if the new search space generated by the packing has fewer remaining items than a predetermined number of items. If it does, instead of recursing with bin completion, the basic item-assignment search is called on the remaining items to be packed instead of bin completion. Since both algorithms have a similar view of the problem that consists of a list of bins that are partially-packed, a list of vectorized items that still need to be packed, and similar additional data structures, the cost of this switch is virtually negligible. The hybrid algorithm is thus able to take advantage of the faster speed of bin completion on sufficiently large numbers of items and switch to the item-assignment space when it becomes advantageous.

In order to decide the appropriate number of items where one should switch between the two algorithms (the *item threshold*), we performed a series of experiments. This item threshold is dependent on the dimension, as the two algorithms scale differently with increasing dimensionality. Additionally, since multiple items are packed at once by bin completion, the actual switch generally occurs slightly below the threshold value, so thresholds within two or three of each other are virtually identical in practice. As a result, thresholds were determined by running the hybrid algorithm with different threshold values in multiples of five and seeing how the hybrid algorithm scaled with increasing number of items. For each dimension, we ran several versions of the hybrid algorithm that differed only in their threshold values on the same trials and increased the number of items as described in the testing methodology section below. The hybrid algorithm which performed best as the number of items increased was chosen as having the appropriate threshold value for that dimension. The threshold values for each dimension are reported in Section 5.

SECTION 5

Results and Conclusions

For the final section of this thesis, we discuss our actual experiments. The first subsection concerns how we determined the appropriate number of trials for this problem. We then discuss our methodology for actually performing comparisons between different algorithms. The results themselves are in their own subsection, followed by our conclusions.

5.1 Number of Required Trials

The question of the appropriate number of trials required to get accurate performance characteristics has not been studied extensively previously. [Kor03] performed one million randomized trials for each number of items and search strategy, whereas [CT01] only performed ten trials for each number of items and search strategy. In order to discover an appropriate number of trials, we ran our hybrid search algorithm on a sequence of item sizes from 5 to 50 with the same number of randomized trials multiple times, each time changing the random seed. While the random seed can influence the average time per trial, if one runs sufficiently many randomized trials the average time per trial rarely fluctuates by more than 50% with different random seeds for high numbers of random trials. At low numbers of random trials, however, the number of "hard" problem instances can greatly alter the average trial time. For instance, when running only ten randomized problem instances, it is possible to have average trial time fluctuations of up to a factor of one hundred times between different random seeds, as the number of "hard" problems generated dominates the average. This would be fine if all problems were equally hard for different algorithms, but unfortunately what is hard for one algorithm may be easy for a different algorithm. However, ten thousand, one hundred thousand, and one million randomized trials all have very close average trial times, even when one varies through several different randomized seeds. If one runs too many trials, the overall running time can become prohibitively long. For this reason, we chose to run ten thousand randomized trials for each problem size, and we recommend at least that many trials for future algorithm comparisons in vector-packing problems.

5.2 Testing Methodology

All comparisons of different search types, heuristics, and variable ordering strategies were tested in a similar format. We will call the particular set of all of these choices used to run a search a search strategy. For each search strategy, ten thousand randomized trials were performed where the required output was the correct number of bins used in the optimal packing. Each trial consisted of a set number of vectors whose values in each dimension were all chosen uniformly at random from 0 up to the bin capacity of that dimension. We used a bin capacity of one million to allow for more complex packings than smaller precision allows. The random seed was reset to the same number for each new search strategy to ensure that all search strategies pack exactly the same set of vectors in all trials. This was done for two reasons. First, the exact numbers for the item vectors can have a dramatic effect on search performance, as described in the previous subsection. On average, the difference in time between the shortest trial and the longest trials in each battery of tests was a factor of about 2000. In other words, the hard instances of the problem are MUCH harder than easier instances, so it is important to ensure that the same instances are run for all algorithms. While this is comforting in ensuring a fair comparison between all the search strategies, the second reason to compare the same items exactly for all algorithms is to verify the correctness of the implementation of each search strategy. The total number of bins used by each search in all trials was calculated and compared amongst the search strategies. Since the optimal number of bins does not depend on the strategy used to find it, this number should be identical for all of the strategies, and serves as a basic check for correctness of an implementation of any particular search strategy.

5.3 Results

The tables below show the total runtime of all ten thousand trials in seconds with the specified algorithm and the number of items specified by the each row. In Table 5.1, the one-dimensional results show that the bin completion algorithm initially has a similar runtime to item assignment, and then becomes significantly faster than the item-assignment algorithm for all numbers of items beyond 15. This result is not very surprising, and validates the one-dimensional results of [Kor03] even with some optimizations specific to one dimension removed. The cutoff time for the final size of item assignment in one dimension and several other dimensions is 86,400 seconds, or 24 hours. If the set of ten thousand trials required more time than 24 hours, we simply stopped that experiment and report it as in excess of 24 hours. We do not report hybrid results for one and two dimensional data, as bin completion's runtime is not significantly improved by using the hybrid algorithm for those dimensions.

Table 5.1: 1-Dimensional Data

Number of Items	Item Assignment	Bin Completion
5	0.03	0.03
10	0.06	0.06
15	0.12	0.11
20	0.40	0.19
25	10.44	0.29
30	112.14	0.43
35	3,764.80	0.59
40	>86,400	0.82

Table 5.2: 2-Dimensional Data

Number of Items	Item Assignment	Bin Completion
5	0.04	0.05
10	0.11	0.16
15	0.21	0.35
20	0.43	0.70
25	0.98	1.39
30	6.79	3.17
35	141.01	9.25
40	1,073.33	31.06
45	6,988.28	107.09
50	77,206.37	609.64

In Table 5.2, the two-dimensional results show that bin completion is slower than item assignment until size 25, after which point it becomes much faster than item assignment. This is most dramatically shown for 50 packed items, where item assignment takes nearly 80,000 seconds seconds and bin completion takes about 610 seconds.

The three-dimensional data is shown in Table 5.3. An interesting feature in this data

Table 5.3: 3-Dimensional Data

Number of Items	Item Assignment	Bin Completion	Hybrid *(n = 25)*
5	0.06	0.07	0.07
10	0.15	0.24	0.17
15	0.28	0.49	0.31
20	0.47	0.84	0.51
25	0.79	1.42	0.92
30	1.38	2.48	1.56
35	3.03	5.27	3.08
40	10.60	11.84	7.44
45	34.31	31.39	17.23
50	90.34	144.78	59.92
55	288.76	505.29	267.14
60	10,185.61	2,931.89	1,799.77
65	5,466.17	6,373.22	5,351.92
70	>86,400	32,651.95	25,143.04

occurs at size 60 for item-assignment. At this point, item-assignment actually takes about twice as long as the time required for size 65, and much longer than any of the other algorithms take to complete size 60. While the ten thousand trials have until now helped to smooth out the effects of individual hard trials, for this problem size there is a single trial that takes item-assignment two-thirds of the time of the entire run. Such extremely hard trials are somewhat rarer and less extreme when they do occur in bin completion because dominance checking helps to prune some of the truly terrible sections of a search tree. As for the rest of the data, bin completion is now slightly worse than item-assignment at all other trial sizes until size 70, where bin completion finally catches up to item assignment and has a much better runtime.

The hybrid algorithm first becomes relevant in three dimensions, taking advantage of bin completion's superior performance on higher numbers of items and item-assignment's efficient packing of lower numbers of items. The hybrid algorithm is slightly faster than item-assignment at all trial sizes beyond size 35, and lacks the extreme jump in trial time for size 60 that item-assignment features. Below size 25, the hybrid algorithm is technically identical to item assignment since the threshold value is 25, but timer resolution issues cause it to have some variation in the times relative to item-assignment's run. However, there is a slight “warm-up” period between sizes 25 and size 35 where the hybrid is slightly worse and is actually running bin completion when there are 25 or more items left to pack. While this might suggest that one should use a higher threshold value, experimental results suggest that 25 is actually the correct threshold value for faster performance on sizes beyond 40. As the hybrid is run on trials with more than 40 items, it has superior runtimes to both bin completion and item assignment.

Table 5.4: 4-Dimensional Data

Number of Items	Item Assignment	Bin Completion	Hybrid *(n = 20)*	Hybrid *(n = 25)*	Hybrid *(n = 30)*	Hybrid *(n = 35)*
5	0.08	0.10	0.09	0.08	0.08	0.09
10	0.26	0.31	0.22	0.21	0.22	0.22
15	0.40	0.57	0.39	0.38	0.39	0.39
20	0.58	0.89	0.65	0.60	0.61	0.60
25	0.84	1.29	1.04	0.95	0.90	0.90
30	1.31	1.85	1.54	1.49	1.44	1.39
35	2.07	2.64	2.16	2.16	2.24	2.32
40	4.23	4.20	3.13	3.29	3.65	4.23
45	7.56	5.92	4.65	4.67	5.15	6.74
50	23.40	9.41	7.00	7.17	8.46	12.01
55	50.93	18.66	13.29	13.01	15.66	21.83
60	149.32	43.50	26.95	25.19	30.09	37.49
65	336.01	83.45	60.99	46.40	59.88	65.90
70	823.86	352.06	245.72	317.60	498.19	752.03
75	36,484.29	1,106.40	1,417.63	1,673.77	950.98	739.14
80	30,143.67	1,750.62	1,283.40	862.63	724.20	975.71
85	38,409.70	3,842.01	3,646.38	3,267.15	3,225.57	2,826.41
90	$>$ 86,400	59,108.75	65,654.48	44,905.85	53,043.78	17,168.37

Shaded cells indicate lowest threshold value (n) of fastest hybrid runtime

10,000 Trials : Total Time in Seconds

The four-dimensional data in Table 5.4 is somewhat confusing. Bin completion is faster than item assignment for trials where one is packing at least 45 items. While this is a positive result for bin completion, the hybrid threshold values for four dimensional data vary significantly as far as which threshold value is optimal for a given item size. The shaded table values highlight the hybrid with the best performance for a given number of items. A threshold value of 25 is close to being at least an improvement on bin completion, but the extreme jump in runtime for item assignment at size 75 seems to affect the hybrid at size 75 as well, and it becomes worse than bin completion for that size only. Another reasonable choice appears to be 35, which is better than Bin Completion for all values where one is packing at least 75 items, but unfortunately it is twice as slow as bin completion for size 70 and at least slightly worse than bin completion for sizes 55, 50, 45, and 40. If one looks only at the trend for sizes 85 and 90, it appears that higher threshold values are to be preferred, although clearly this has a limit as item assignment for sizes 85 and 90 is much worse than any of the explored options. The best conclusion to draw from the four-dimensional data is that while bin completion clearly scales better with increasing items than item assignment, properly assigning threshold values could use additional research. Until further research proves otherwise, we recommend simply running bin-completion without the hybrid in four dimensions.

In five dimensions (see below), bin completion outperforms item assignment for trials where one is packing at least 45 items. Additionally, a clear threshold value of 20 works well at all numbers of items, and it is able to at least marginally outperform bin completion at all trial sizes, as shown in Table 5.5. Both bin completion and the hybrid algorithm outperform item assignment for trials where one is packing at least 45 items. In five dimensions, it seems ideal to simply run the hybrid algorithm with an item threshold of 20 items. This is significant primarily because it is a lower threshold than the threshold for three dimensions. This may suggest that the advantage item assignment has over bin completion at low numbers of items for higher dimensional trials actually begins to diminish. If that were the case, for sufficiently high dimensions, bin completion would be superior to item assignment at all or most numbers of items.

Table 5.5: 5-Dimensional Data

Number of Items	Item Assignment	Bin Completion	Hybrid *(n = 20)*
5	0.09	0.12	0.10
10	0.22	0.36	0.25
15	0.38	0.64	0.44
20	0.59	0.96	0.73
25	0.85	1.34	1.13
30	1.23	1.79	1.59
35	1.74	2.31	2.10
40	2.63	2.96	2.74
45	4.03	3.84	3.46
50	6.68	4.72	4.37
55	12.12	5.87	5.46
60	33.72	7.58	6.88
65	54.77	9.92	8.96
70	118.04	14.61	12.49
75	242.55	17.61	15.73
80	767.19	31.78	28.49
85	1,797.07	55.89	43.85
90	3,636.89	74.98	63.53
95	24,689.76	123.57	114.95
100	26,062.81	251.75	212.34
105	86,150.78	348.01	261.26

10,000 Trials : Total Time in Seconds

5.4 Conclusions

The results show that bin completion ultimately scales better with increasing numbers of items than item assignment does. However, for some dimensions item assignment performs better than bin completion for small numbers of items to pack, so the hybrid results are better than either algorithm in three and five dimensions, albeit sometimes only marginally so. In general, the approach has validity; we can pack vectorized items for any number of dimensions. It is possible that better approximations might help with all runtimes, however, and thus push bin completion into a useful range sooner. There also may be cleverer ways of deciphering when one should switch between the item-assignment and bin-completion algorithms based on the actual characteristics of a given packing problem, as opposed to a static threshold value. Both of these areas could be fruitful areas for further research, and might extend the number of dimensions for which this technique is computationally useful. In general, bin completion scales better than item-assignment with increasing numbers of items for all dimensions studied, and can often have even faster runtimes by using a hybrid approach.

As a final note, we have established a method by which future vector-packing algorithms can be reasonably compared and created a standard experimental methodology for comparison in any dimension. As long as similar experimental comparisons are used, valid improvements can be measured regardless of hardware changes or exotic algorithms. We hope that further research will use these results as a baseline to compare against and the methodology we developed to create a standard for comparison.

APPENDIX A

Random Number Generation

All random numbers were generated with the Java class Random. We were using Java version 1.7.0_03. For each algorithm, the random seed was initially set to -430238454. We generated ten thousand vectors one at a time, generating all components of each vector before moving to the next vector.

Each component was generated by the following calculation for the random number generator r:

component = 1 + r.nextInt(1000000 - 1)

This should make it possible to replicate any of our actual trials, should the reader be interested.

References

[BCS06] Nikhil Bansal, Alberto Caprara, and Maxim Sviridenko. "Improved approximation algorithms for multidimensional bin packing problems." *2006 47th Annual IEEE Symposium on Foundations of Computer Science FOCS06*, **410**(44):697–708, 2006.

[BS96] James Beck and Daniel Siewiorek. "Modeling Multicomputer Task Allocation as a Vector Packing Problem." In *Proceedings of the 9th international symposium on System synthesis*, ISSS '96, pp. 115–, Washington, DC, USA, 1996. IEEE Computer Society.

[CHP05] Soo Y. Chang, Hark-Chin Hwang, and Sanghyuck Park. "A two-dimensional vector packing model for the efficient use of coil cassettes." *Comput. Oper. Res.*, **32**(8):2051–2058, August 2005.

[CMB09] Manogna Chebiyyam, Rashi Malviya, Sumit Kumar Bose, and Srikanth Sundarrajan. "Server consolidation Leveraging the benefits of virtualization." *SETLabs Briefings*, **7**(1):65–74, 2009.

[CT01] Alberto Caprara and Paola Toth. "Lower bounds and algorithms for the 2-dimensional vector packing problem." *Discrete Appl. Math.*, **111**(3):231–262, August 2001.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979.

[Kor02] Richard E. Korf. "A new algorithm for optimal bin packing." In *Eighteenth national conference on Artificial intelligence*, pp. 731–736, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[Kor03] Richard E. Korf. "An improved algorithm for optimal bin packing." In *Proceedings of the 18th international joint conference on Artificial intelligence*, IJCAI'03, pp. 1252–1258, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

[Kor11] Richard E. Korf. "Heuristic Search." Course materials for Spring 2011 CS261A Course (*Problem Solving and Search*) at University of California Los Angeles, 2011.

[MT90a] Silvano Martello and Paolo Toth. *Knapsack problems: algorithms and computer implementations.* John Wiley & Sons, Inc., New York, NY, USA, 1990.

[MT90b] Silvano Martello and Paolo Toth. "Lower bounds and reduction procedures for the bin packing problem." *Discrete Applied Mathematics*, **28**(1):59 – 70, 1990.

[OPR11] Ekow J. Otoo, Ali Pinar, and Doron Rotem. "A Linear Approximation Algorithm for 2-Dimensional Vector Packing." *The Computing Research Repository (CoRR)*, **abs/1103.0260**, 2011.