An MDD Approach to Multidimensional Bin Packing *

Brian Kell[1] and Willem-Jan van Hoeve[2]

[1] Department of Mathematical Sciences, Carnegie Mellon University
bkell@cmu.edu
[2] Tepper School of Business, Carnegie Mellon University
vanhoeve@andrew.cmu.edu

Abstract. We investigate the application of multivalued decision diagrams (MDDs) to multidimensional bin packing problems. In these problems, each bin has a multidimensional capacity and each item has an associated multidimensional size. We develop several MDD representations for this problem, and explore different MDD construction methods including a new heuristic-driven depth-first compilation scheme. We also derive MDD restrictions and relaxations, using a novel application of a clustering algorithm to identify approximate equivalence classes among MDD nodes. Our experimental results show that these techniques can significantly outperform current CP and MIP solvers.

1 Introduction

Many related problems in combinatorial optimization are collectively referred to as "bin packing problems." In the classical bin packing problem, the input is a list $(s_1, \ldots, s_n)$ of item sizes, each in the interval $(0, 1]$, and the objective is to pack the n items into a minimum number of bins of capacity 1.

In this paper we study a multidimensional variant of the bin packing problem, presented as a satisfaction problem. An instance of this problem consists of a list $(s_1, \ldots, s_n)$ of item sizes and a list $(c_1, \ldots, c_m)$ of bin capacities. Each item size and each bin capacity is a d-tuple of nonnegative integers; e.g., $s_i = (s_{i,1}, \ldots, s_{i,d})$. The objective is to assign each of the n items to one of the m bins in such a way that, for every bin and in every dimension, the total size of the items assigned to the bin does not exceed the bin capacity.

This can be viewed as a constraint satisfaction problem (CSP) with n variables and md constraints. Each variable x_i has domain $\{1, \ldots, m\}$ and denotes the bin into which the ith item is placed. The constraints require that $\sum_{i:x_i=j} s_{i,k} \le c_{j,k}$ for all $j \in \{1, \ldots, m\}$ and all $k \in \{1, \ldots, d\}$.

Note that the "dimensions" in this problem should not be interpreted as geometric dimensions. In this way the problem studied here differs from the two- and three-dimensional bin packing problems studied, for example, in [10,

* This work was supported by the NSF under grant CMMI-1130012 and a Google Research Grant.

11], in which the items and bins are geometric rectangles or cuboids. Rather, the dimensions in the problem studied in this paper correspond to independent one-dimensional bin packing constraints that must be satisfied simultaneously.

Multidimensional bin packing (MBP) problems of the kind considered in this paper appear in practice. For example, the Google ROADEF/EURO challenge 2011–2012[3] involves a set of machines with several resources, such as RAM and CPU, running processes which consume those resources. However, these problems have received relatively little attention in the literature. Current CP methods are weak on problems involving simultaneous bin packing constraints. Current MIP methods do better but are still limited in their effectiveness.

In this paper we make the following contributions. We present a new generic exploratory construction algorithm for multivalued decision diagrams (MDDs) and a novel application of the median cut algorithm in the construction of approximate MDDs. We also describe several techniques specific to the use of MDDs for the MBP problem. Our experimental results show that such techniques can yield an improvement on existing methods.

The remainder of the paper is organized as follows. In Section 2 we present several generic approaches to the construction of MDDs. The focus of Section 3 is approximate MDDs, which represent sets of solutions to relaxations or restrictions of problem instances. In Section 4 we discuss techniques that can be used to apply MDDs to the MBP problem. In Section 5 we present experimental results comparing the performance of the techniques described in this paper to that of commercial CP and MIP solvers. We conclude in Section 6.

2 MDD Construction

In this section we present a generic algorithm for the construction of an MDD representing the set of feasible solutions to a CSP. A CSP is specified by a set of constraints $\{C_1, \ldots, C_p\}$ on a set of variables $\{x_1, \ldots, x_n\}$ having domains $D_1, \ldots, D_n$, respectively. A *solution* to a CSP is an n-tuple $(y_1, \ldots, y_n) \in D_1 \times \cdots \times D_n$. A solution is *feasible* if the set of assignments $x_1 = y_1, \ldots, x_n = y_n$ satisfies every constraint C_j.

A *multivalued decision diagram* (MDD) [13] is an edge-labeled acyclic directed multigraph whose nodes are arranged in $n+1$ layers $L_1, \ldots, L_{n+1}$. The layer L_1 consists of a single node, called the *root*. Every edge in the MDD is directed from a node in L_i to a node in L_{i+1}. All of the edges directed out of a node have distinct labels. The nodes in layer L_{n+1} are called *sinks* or *terminals*. In this paper we shall primarily be interested in MDDs having a single sink (which represents feasibility), but the ideas can easily be generalized to MDDs with multiple sinks [13].

Let I be an instance of a CSP. An MDD M can be used to represent a set of solutions to I as follows [1]. The layers $L_1, \ldots, L_n$ correspond respectively to the variables $x_1, \ldots, x_n$ in I. An edge directed from a node in L_i to a node

[3] Online: `http://challenge.roadef.org/2012/en/index.php`.

in L_{i+1} and having the label y_i, where $y_i \in D_i$, corresponds to the assignment $x_i = y_i$. Therefore a path from the root to the sink along edges labeled $y_1, \ldots, y_n$ corresponds to the solution $(y_1, \ldots, y_n)$. The MDD M represents the set $\mathcal{M}$ of solutions corresponding to all such paths. Let $\mathcal{F}$ denote the set of feasible solutions to I. If $\mathcal{M} = \mathcal{F}$, $\mathcal{M} \supseteq \mathcal{F}$, or $\mathcal{M} \subseteq \mathcal{F}$, then M is said to be an *exact MDD*, a *relaxation MDD*, or a *restriction MDD* for I, respectively $[1, 3, 4, 6]$. We shall consider only exact MDDs in this section; relaxation and restriction MDDs will be considered in Section 3.

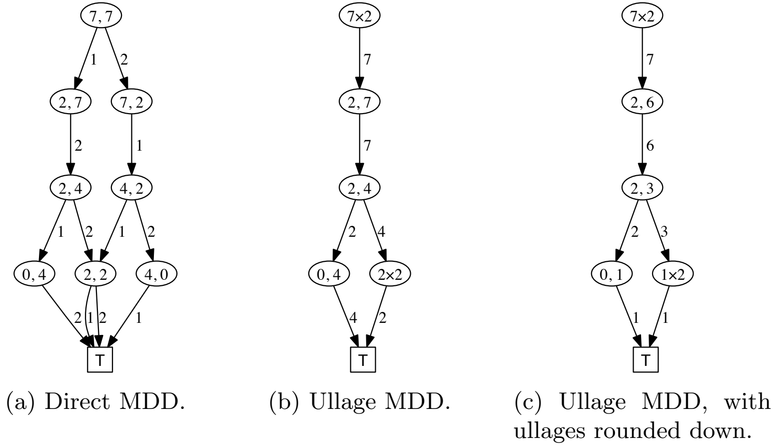
A path in an MDD from the root to a node in the layer L_{i+1} represents a *partial solution* $y = (y_1, \ldots, y_i) \in D_1 \times \cdots \times D_i$; we shall say that the *level* of this partial solution is i and write $\text{level}(y) = i$. Let $\mathcal{F}(y)$ denote the set of *feasible completions* of this partial solution, that is, $\mathcal{F}(y) = \{\, z \in D_{i+1} \times \cdots \times D_n \mid (y, z) \text{ is feasible} \,\}$. If y and y' are partial solutions with $\mathcal{F}(y) = \mathcal{F}(y')$, then we say that y and y' are *equivalent*. Note that in an exact MDD all paths from the root to a fixed node v represent equivalent partial solutions, and conversely if two partial solutions y and y' are equivalent then the paths in an exact MDD that correspond to y and y' can lead to the same node.

Direct MDD Representation for Multidimensional Bin Packing. Let I be an MBP instance, having n items and m bins. A direct MDD representation of the set of feasible solutions of I has layers $L_1, \ldots, L_n$ corresponding to the variables $x_1, \ldots, x_n$, and also the last layer L_{n+1} which contains the sink. The edge labels are elements of $\{1, \ldots, m\}$. A path from the root to the sink along edges labeled $y_1, \ldots, y_n$ represents the feasible solution $(y_1, \ldots, y_n)$, that is, the feasible solution in which item i is placed into bin y_i.

For example, Figure 1a shows the direct MDD representation for a one-dimensional bin packing instance having two bins, each of capacity 7, and four items, with sizes 5, 3, 2, and 1. There are six paths from the root to the sink, representing the six feasible solutions; for instance, the path following the edges labeled 2, 1, 1, 2 corresponds to the solution in which the item of size 5 is packed in bin 2, the items of size 3 and 2 are packed in bin 1, and the item of size 1 is packed in bin 2. The node labels are states; we discuss these next.

Equivalence of Partial Solutions. Equivalent partial solutions have the same set of feasible completions. Hence, the recognition that two partial solutions are equivalent reduces the size of the MDD, because the corresponding paths can lead to the same node.

In general, determining whether two partial solutions are equivalent is NP-hard for the MBP problem (because it is NP-hard even to determine whether a one-dimensional instance is feasible). However, we can sometimes determine that two partial solutions are equivalent by associating partial solutions with "states." A *state function* for layer i is a map σ_i from the set $Y_i = D_1 \times \cdots \times D_{i-1}$ of partial solutions at layer i into some set S_i of *states*, such that $\sigma_i(y) = \sigma_i(y')$ implies $\mathcal{F}(y) = \mathcal{F}(y')$. In other words, two partial solutions that lead to the same state have the same set of feasible completions. (A "perfect" state function would



(a) Direct MDD.

(b) Ullage MDD.

(c) Ullage MDD, with ullages rounded down.

Fig. 1. MDD representations of a one-dimensional bin packing instance having two bins, each of capacity 7, and four items, with sizes 5, 3, 2, and 1.

also allow us to say that two partial solutions that lead to different states have different sets of feasible completions, and we strive for this ideal, but for practical reasons our state function should be easy to compute, so we cannot require this.)

Consider an MBP instance. After items having sizes $s_{i_1}, \ldots, s_{i_k}$ have been placed into a bin of capacity c_j, the remaining capacity of the bin is $c_j - \sum_{l=1}^{k} s_{i_l}$. (Recall that the item sizes and bin capacities are d-tuples; here and elsewhere addition and subtraction of d-tuples is done componentwise.) We shall call this remaining capacity the *ullage* of the bin; it is a d-tuple. (The word "ullage" means "the amount by which a container falls short of being full.") Of course, the ullage of each bin is nonincreasing (componentwise) as the items are placed one by one into the bins.

A useful state function for the direct MDD representation is the map σ_i from a partial solution $y = (y_1, \ldots, y_{i-1})$ to the list $(u_1, \ldots, u_m)$ of the ullages u_j of the m bins; in other words, for $j \in \{1, \ldots, m\}$, we take $u_j = c_j - \sum_{k \in \{1,\ldots,i-1\}: y_k = j} s_k$. For example, in Fig. 1a, the path from the root along the edges labeled 1, 2, 2 represents a partial solution for which the ullages of the two bins are each 2, so the state of this partial solution is $(2, 2)$. The partial solution corresponding to the path 2, 1, 1 has the same state. Observe that if two partial solutions at layer i have the same lists of ullages, then they have the same set of feasible completions, so this is indeed a state function.

Exact MDD Construction. Behle [2] described a top-down algorithm for the construction of threshold binary decision diagrams (BDDs), which are exact representations of solution sets of instances of 0–1 knapsack problems. A general algorithm for a top-down, layer-by-layer (i.e., breadth-first) construction of an MDD is presented as Algorithm 1, "Top-down MDD compilation," in Bergman et al. [4]. The key to the top-down construction of an MDD is the identification

of a *node equivalence test*, which determines when two nodes on the same layer (each representing one or more partial solutions) have the same set of feasible completions; this is exactly what a state function does.

So far we have spoken of the states of partial solutions. We shall now extend this idea to states of nodes in an MDD. In the MDD that we construct, partial solutions at layer i that lead to the same state will correspond to paths from the root that lead to the same node; we shall associate this state with this node. Now, given a node v in layer L_i in the MDD and its state, which we shall write as state(v), and given a value $y_i \in D_i$, we can determine the state of a child node w of v if the edge (v, w) has label y_i. This is simply the state of the feasible solution (y, y_i), where y is any feasible solution corresponding to the node v. For instance, suppose v is a node in layer L_i of a direct bin packing MDD, and suppose the state of v (i.e., the corresponding list of ullages) is $(u_1, \ldots, u_m)$. Then the child state corresponding to $y_i \in D_i$ is $(u_1, \ldots, u_{y_i-1}, u_{y_i} - s_i, u_{y_i+1}, \ldots, u_m)$.

To be more precise, and to make these ideas applicable to generic CSPs, we make the following definitions. Let $i \in \{1, \ldots, n+1\}$. Let $Y_i = D_1 \times \cdots \times D_{i-1}$ denote the set of partial solutions at level i; take $Y_1 = \{\emptyset\}$, a singleton set having one element representing the empty partial solution. Let S_i be an arbitrary set whose elements are called *states* and which contains a special element $\perp$ indicating infeasibility. Recall that we say that $\sigma_i : Y_i \to S_i$ is a *state function* if $\sigma_i(y) = \sigma_i(y')$ implies $\mathcal{F}(y) = \mathcal{F}(y')$; we also require that $\sigma_i(y) = \perp$ implies $\mathcal{F}(y) = \emptyset$. We assume that we can test the feasibility of a (complete) solution, so for $y \in Y_{n+1}$ we require that $\sigma_{n+1}(y) = \perp$ if $\mathcal{F}(y) = \emptyset$. For $i \in \{1, \ldots, n\}$, we say that $\chi_i : S_i \times D_i \to S_{i+1}$ is a *child state function* if $\chi_i\big(\sigma_i(y), y_i\big) = \sigma_{i+1}(y, y_i)$ for all $y \in Y_i$ and all $y_i \in D_i$.

In order to use state information effectively in the construction of an MDD, we must maintain, for each layer L_i, a mapping from states to nodes that have already been constructed in L_i. When we seek a node in L_i having state s, we consult this mapping to see if such a node already exists. Such a mapping can be implemented with a hash table. It is often called the *unique table* because it ensures that the node representing state s in layer L_i is unique [9].

Algorithm 1 constructs an exact MDD. For each $i \in \{1, \ldots, n+1\}$ let σ_i be a state function, and for each $i \in \{1, \ldots, n\}$ let χ_i be a corresponding child state function. Let r be the root node. The algorithm maintains a collection T of nodes to be processed, i.e., nodes whose children need to be constructed. When a node v in layer i is processed, each possible domain value $y \in D_i$ is considered, and the corresponding child state s is computed. The unique table is consulted to see if a node w with state s already exists in layer L_{i+1}; if not, a new node w is constructed and added to T. Then the edge (v, w) is added to the MDD with label y. This is repeated until all nodes have been processed.

Exploratory Construction. The main difference between Algorithm 1 and the top-down exact MDD compilation algorithm of Bergman et al. is the order in which the nodes are processed. Instead of requiring that the nodes be processed layer by layer, we allow the collection T to provide the nodes in any order. This

Algorithm 1 Exact MDD Construction

```
1: L_1 := {r}
2: T := {r}
3: while T is not empty do
4:     select v ∈ T and remove it from T
5:     i := layer(v)
6:     for all y ∈ D_i do
7:         s := χ_i(state(v), y)
8:         if s ≠ ⊥ then
9:             w := unique-table(i + 1, s)
10:            if w = nil then
11:                w := new node with state s
12:                add w to L_{i+1}
13:                add w to T
14:            add edge (v, w) with label y
```

generalization permits exploratory construction of the MDD. For example, if we are constructing the MDD in order to seek a feasible solution, we can build it in a depth-first manner by taking T to be a stack. The layer-by-layer behavior of the algorithm of Bergman et al. can be achieved by using a queue for T. Note that if we do construct the MDD layer by layer, we can discard the unique table for each layer as soon as we have finished processing the previous layer.

It is useful to have a heuristic to estimate the "promise" of a partial solution, that is, the likelihood that it has a feasible completion. Such a heuristic can be used to guide the depth-first construction of an MDD in search of a feasible solution. For the MBP problem, we propose the following heuristic. Given a partial solution $(y_1, \ldots, y_i)$ describing the packing of the first i items into bins, we perform a non-backtracking random packing of the remaining items ($i+1$, $\ldots$, n) as well as we can without violating the bin packing constraints. In other words, we iterate through the remaining items in order, and we pack each item into one of the bins that has sufficient ullage, chosen at random; if no such bin exists, we put the item into a trash pile. At the end we count the total size of the items in the trash pile, along all d dimensions, and this number is the score of this packing. This random packing of the remaining items is repeated several times, and the total score of these packings is used as the heuristic value of the partial solution; a low score is better. (Occasionally, while we are computing the heuristic for a partial solution in this way, we may luckily find a feasible completion: the trash pile will be empty. In this case, if we are constructing the MDD merely to seek a feasible solution, we can immediately return the solution thus found.)

With such a heuristic, we can use a priority queue for T to select the most promising nodes to process next. Alternatively, we can use a stack for T and modify Algorithm 1 slightly so that when we process a node we construct all its children, evaluate their heuristics, and then add them to T in reverse order of their promise. This will yield a depth-first algorithm that explores the most promising child of each node first.

This depth-first MDD construction process, especially if it is being used simply to find a feasible solution, is very similar to a backtracking search. It is an improvement, however, because the MDD nodes act as a memoization technique to prevent the exploration of portions of the search tree that can be recognized as equivalent to portions already explored.

3 Approximate MDDs

In general, exact MDDs can be of exponential size, so the use of Algorithm 1 may not be practical because of space limitations. In this case we may be able to use an approximate MDD to get useful results.

The MDDs described in this section are called approximate because their structure approximates the structure of the exact MDD. An approximate MDD represents a set of solutions to a relaxation or a restriction of the problem instance. Hence, if a restriction MDD indicates that an instance is feasible, then every solution it represents (i.e., every path from the root to the sink) is an exact feasible solution to the original instance. Similarly, an indication of infeasibility from a relaxation MDD is a proof that the original instance is infeasible. In this way, relaxation and restriction MDDs can be used together to determine the feasibility or infeasibility of an instance, and to get an exact feasible solution if the instance is feasible. Of course, it is possible for a relaxation MDD to indicate that an instance is feasible while a restriction MDD indicates it is infeasible, in which case nothing is learned. In response, one could construct MDDs representing tighter relaxations or restrictions (probably at the cost of greater time and space requirements), or could embed the MDDs inside a complete search.

Approximation MDDs by Merging. MDDs of limited width were proposed by Andersen et al. [1] to reduce space requirements. In this approach, the MDD is constructed in a top-down, layer-by-layer manner; whenever a layer of the MDD exceeds some preset value W an approximation operation is applied to reduce its size to W before constructing the next layer. For this approximation, Bergman et al. [4] use a relaxation operation $\oplus$ defined on states of nodes so that, given nodes v and v', the state given by $\text{state}(v) \oplus \text{state}(v')$ is a "relaxation" of both $\text{state}(v)$ and $\text{state}(v')$; see also [8].

We can formalize this idea as follows. Let $C_i = D_i \times \cdots \times D_n$ denote the set of completions at level i (independent of any particular partial solution). For a partial solution $y \in Y_i$, the set of feasible completions of y is some subset of C_i, so $\mathcal{F}(y) \in \mathcal{P}(C_i)$, where $\mathcal{P}$ denotes the power set. Recall that a state function $\sigma_i : Y_i \to S_i$ is such that $\sigma_i(y) = \sigma_i(y')$ implies $\mathcal{F}(y) = \mathcal{F}(y')$, and $\sigma_i(y) = \bot$ implies $\mathcal{F}(y) = \emptyset$. The existence of such a function implies the existence of a *completion function* $\tau_i : S_i \to \mathcal{P}(C_i)$ such that $\tau_i(\sigma_i(y)) = \mathcal{F}(y)$ for all $y \in Y_i$. For $i \in \{1, \ldots, n\}$, we say that a binary operation $\vee_i : S_i \times S_i \to S_i$ is a *relaxation merge* if for all $y, y' \in Y_i$ we have $\tau_i(\sigma_i(y) \vee_i \sigma_i(y')) \supseteq \mathcal{F}(y) \cup \mathcal{F}(y')$. In other words, the set of feasible completions implied by the state $\sigma_i(y) \vee_i \sigma_i(y')$ contains all feasible completions implied by the state $\sigma_i(y)$ and all feasible completions

Algorithm 2 Approximate MDD Construction by Merging

```
1: L_1 := {r}
2: for i = 1 to n do
3:     L_{i+1} := ∅
4:     for all v ∈ L_i do
5:         for all y ∈ D_i do
6:             s := χ_i(state(v), y)
7:             if s ≠ ⊥ then
8:                 w := unique-table(i + 1, s)
9:                 if w = nil then
10:                    w := new node with state s
11:                    add w to L_{i+1}
12:                add edge (v, w) with label y
13:    if |L_{i+1}| > W then
14:        partition L_{i+1} into W clusters A_1, ..., A_W
15:        for j = 1 to W do
16:            w_j := new node with state ⋁A_j (or ⋀A_j)
17:            for all v ∈ A_j do
18:                change every edge (u, v) to (u, w_j) with the same label
19:        L_{i+1} := {w_1, ..., w_W}
```

implied by the state $\sigma_i(y')$. Similarly, we call $\wedge_i : S_i \times S_i \to S_i$ a *restriction merge* if for all $y, y' \in Y_i$ we have $\tau_i\big(\sigma_i(y) \wedge_i \sigma_i(y')\big) \subseteq \mathcal{F}(y) \cap \mathcal{F}(y')$. For simplicity, we shall omit the subscript and just write $\vee$ or $\wedge$. These merge operations need not be associative or commutative. However, in a slight abuse of notation, we shall write $\bigvee A$ to denote a combination of all elements $s \in A \subseteq S_i$ using the relaxation merge operation $\vee$, in any order and parenthesized in any way; likewise for $\bigwedge A$.

For the direct MDD representation of an MBP instance, in which node states are lists of ullages $(u_1, \ldots, u_m)$, an appropriate relaxation (respectively, restriction) merge is the componentwise maximum (respectively, minimum).

Bergman et al. give an algorithm to construct a limited-width MDD which iteratively merges pairs of nodes in a layer using a relaxation merge. We propose a refinement of this technique that uses a clustering algorithm to partition the nodes in the layer into W clusters; the nodes in each cluster are then merged into a single node. This is outlined in Algorithm 2.

To perform the clustering of nodes on line 14 of Algorithm 2, we adapted the median cut algorithm of Heckbert [7], which was originally designed for color quantization of images. The median cut algorithm operates on a set of points in q-dimensional Euclidean space (in the original version, $q = 3$, representing the red, green, and blue components of each pixel in the image) and partitions the points into clusters. Initially all of the points are grouped into a single cluster, which is tightly enclosed by a q-dimensional rectangular box. Then the following operation is repeatedly performed: the box having the longest length (among all boxes in all q dimensions) is selected, and it is divided into two boxes along this longest length at the median point, that is, in such a way that each of the two smaller boxes contains approximately half of the points in the original

Algorithm 3 Restriction MDD Construction by Deletion

```
 1: (lines 1–12 are the same as in Algorithm 2)
13:     if |L_{i+1}| > W then
14:         use heuristic to select most promising nodes w_1, ..., w_W ∈ L_{i+1}
15:         for all w ∈ L_{i+1} \ {w_1, ..., w_W} do
16:             delete w from L_{i+1} and delete all edges (u, w)
```

box; the two smaller boxes are then "shrunk" to fit tightly around the points they contain. This process continues until the desired number of clusters (boxes) have been generated. The median cut algorithm can be implemented to run in $O\big(K(pq + \log K)\big)$ time, where K is the desired number of clusters, p is the number of points, and q is the number of dimensions.

To apply the median cut algorithm to the nodes in a layer of an MDD, we interpret the state of each node as a point in q-dimensional Euclidean space, for some value of q. For the direct MDD representation of an MBP instance, the state of a node is a list of d-dimensional ullages, one for each of the m bins; so we view this state directly as an md-dimensional point.

If a merged MDD reports that a CSP is feasible, it is desirable to extract a (possibly) feasible solution from it. One way to do this is to maintain a representative partial solution for each node as the MDD is constructed; when two nodes are merged, either of the two corresponding partial solutions can be selected (perhaps in accordance with a heuristic) as the representative partial solution for the merged node. Then the representative (complete) solution at the sink will be a (possibly) feasible solution for the CSP. The representative partial solution can be viewed as auxiliary state information of the node.

Restriction MDDs by Deletion. Algorithm 2 can be used to construct a limited-width MDD by merging nodes when the size of a layer becomes too large. If we are constructing a restriction MDD, however, then another option is simply to delete some of the nodes in the layer [3]. The selection of nodes to keep can be guided by a heuristic. This is described in Algorithm 3.

We note that this deletion algorithm does not use a partitioning algorithm to cluster the nodes in each layer as the merging algorithm does; instead it incurs the cost of computing a heuristic for each node. So the deletion algorithm may be especially beneficial if partitioning the nodes in a layer of the MDD is slower than computing a heuristic for a node.

4 MDD Techniques for Bin Packing

In the previous sections we have presented generic MDD construction algorithms, suitable for any CSP. In this section we specialize some of these techniques to the MBP problem.

Ullage MDD Representation. Let I be an MBP instance, having n items and m bins. One difficulty with the direct MDD representation of I is that it does not take into account the possible symmetry of the bins. For example, suppose that item 1 will fit in any of the m bins. Then the root of the direct MDD will have m outgoing edges labeled 1 through m, indicating the possible bins into which item 1 can be packed. However, if the bins are all identical, these possibilities are essentially equivalent (up to a reordering of the bins). The direct MDD representation cannot recognize this equivalence, because the sets of feasible completions, corresponding to edge-labeled paths in the MDD, are different. For example, in Fig. 1a, the two edges directed out of the root node represent essentially equivalent choices.

To address the possible symmetry of the bins, we can reduce the number of distinct descriptions of feasible solutions by expressing the solutions differently. Rather than assigning items directly to bins, we assign each item to an ullage. For example, instead of saying that item 3 is packed into bin 2, we say that it is packed into a bin with ullage 4. We call this the *ullage description* of the solution; it consists of a list $(u_1, \ldots, u_n)$ of d-tuples, assigning an ullage to each item.

To specify the domains of the variables u_i in the ullage description of a solution, we define the *ullage multiset function* U. If $C = (c_1, \ldots, c_m)$ is the list of bin capacities in I, then $U\big(C, (u_1, \ldots, u_i)\big)$ denotes the multiset of ullages after the first i items have been placed into bins as described by the list $(u_1, \ldots, u_i)$. This is the same as the multiset of ullages after the first $i-1$ items have been placed, except that an item of size s_i was placed into a bin having ullage u_i; so an element u_i of the multiset should be removed and replaced by an element $u_i - s_i$. Formally, we can define U recursively as follows:

- $U(C, \emptyset) = C$ (viewing C as a multiset).
- For $i \in \{1, \ldots, n\}$, if $U_{i-1} = U\big(C, (u_1, \ldots, u_{i-1})\big)$ is defined and $u_i \in U_{i-1}$, then $U\big(C, (u_1, \ldots, u_i)\big) = (U_{i-1} \setminus u_i) \cup \{u_i - s_i\}$.

With this definition of U, the domain of the variable u_i in the ullage description of a solution is $U\big(C, (u_1, \ldots, u_{i-1})\big)$. Note that this domain depends on the values that have previously been assigned to $u_1, \ldots, u_{i-1}$.

An *ullage MDD representation* of the set of feasible solutions of I has layers $L_1, \ldots, L_{n+1}$. The label of an edge directed out of a node in layer L_i in an ullage MDD is a d-tuple, representing the ullage of the bin into which item i is to be placed (after items 1 through $i-1$ have been placed into bins). Therefore the edge labels $u_1, \ldots, u_n$ along a path from the root to the sink in an ullage MDD correspond to an ullage description $(u_1, \ldots, u_n)$ of a feasible solution to I.

Fig. 1b illustrates the ullage MDD representation for the one-dimensional bin packing instance having two bins of capacity 7 and items with sizes 5, 3, 2, and 1. At the root, the state is $\{7 \times 2\}$, i.e., a multiset containing the element 7 with multiplicity 2. The first item, of size 5, must be placed in a bin having ullage 7; this leads to the state $\{2, 7\}$. Then the second item, of size 3, must be placed in the bin that now has ullage 7, and so forth. Of course, a path from the root to the sink in this ullage MDD can easily be converted into an explicit list of bin assignments if desired.

State Function for the Ullage MDD Representation. For the ullage MDD representation, it is useful to consider the state of a partial solution having ullage description $(u_1, \ldots, u_{i-1})$ to be $U\big(C, (u_1, \ldots, u_{i-1})\big)$, that is, the multiset of ullages of the bins.

This idea can be extended to handle side constraints in the CSP. For example, the steel mill slab problem [12] is essentially a (one-dimensional) bin packing problem with the additional constraint that each item has a color and no bin can contain items of more than two colors. To handle a side constraint like this, we can simply augment the state information of a node to include the colors of items that have been packed into it so far.

A few observations can be used to identify further equivalent partial solutions. Let $u_{j,k}$ denote the ullage of bin j, in the kth dimension, after we have placed items 1 through i into bins. Let a denote the greatest possible sum of a subset of the sizes of items $i+1$ through n, in the kth dimension, that does not exceed $u_{j,k}$. If $a < u_{j,k}$, then we may consider the ullage of bin j, in the kth dimension, to be a rather than $u_{j,k}$ without changing the set of feasible completions. If the order of the items is fixed, the relevant sets of possible sums of remaining items can be computed once at the beginning of the MDD construction in $O(nc_{\max}^2)$ time, where $c_{\max}$ is the largest bin capacity in a single dimension. Using this technique of "rounding down" the ullages across all bins in all dimensions, we can sometimes identify additional equivalent partial solutions (their states may be the same after they are rounded down, even if they were not the same before). Moreover, after rounding down ullages, we may discover that the total ullage in all bins is not enough for the remaining items; then we know that the current state has no feasible completions.

If, after we have placed items 1 through i into bins, there is any bin that is so small that none of the remaining items will fit, we can declare that bin *dead* and remove it from further consideration. This is potentially stronger than rounding down, because it may be that in each dimension, considered separately, there is some remaining item that will fit into the bin; but no remaining item is small enough in every dimension to fit into the bin. Conversely, if after we have placed items 1 through i into bins, there is some bin that is large enough in every dimension that all of the remaining items will fit in it, then we know that the instance is feasible. We call such a bin *free*. Once we discover a free bin, we can immediately return a feasible solution: extend a partial solution corresponding to the current node to a complete solution by packing all remaining items into the free bin. The ideas underlying the concepts of dead and free bins are present in Behle's threshold BDD algorithm [2].

In Fig. 1c we apply the rounding-down technique to the ullage MDD. If we additionally check for dead and free bins, we will discover a free bin in the second layer (the bin with rounded-down ullage of 6).

Variable Ordering. The variable ordering used in an MDD can have a very significant effect on the size of the MDD. Behle [2] investigated the optimal variable

ordering problem for threshold BDDs. In general, the problem of determining whether a given variable ordering of a BDD can be improved is NP-complete [5].

For the MBP problem, we take a simple heuristic approach. We observe that identifying dead bins and free bins is beneficial, and we would like to make such identifications as soon as possible. If we pack the largest items first, then the total size of the remaining unpacked items will decrease quickly in the beginning, which suggests that we may reach free bins early; additionally, we will tend to fill bins quickly in the beginning, which suggests that we may exhaust the bins' capacity quickly and reach dead bins early. However, these ideas are somewhat contradictory, and the latter idea is opposed by the observation that the unpacked items will be small, so they can fit into small spaces.

We therefore use an "interleaved" ordering, in which the largest item is packed first, then the smallest item, then the second largest item, then the second smallest item, and so on, packing the median-sized item last. (Our item sizes are multidimensional, so we use the total size of the item in all dimensions: $s_i = \sum_{k=1}^{d} s_{i,k}$.) This ordering seemed to work well for our experimental instances. This straightforward approach means that we can implement variable ordering by sorting the items in this manner as a preprocessing step.

5 Experimental Results

We implemented the MDD-based algorithms described above in Java, using the exploratory construction method described in Section 2, the approximation methods from Section 3, and the ullage MDD representation and the other techniques described in Section 4.

Our test instances were generated as follows. Given the parameters d (the number of dimensions), n (the number of items), m (the number of bins), and β (percentage bin slack), we first generate a list of n item sizes $(s_1, \ldots, s_n)$, each of which is a d-tuple whose coordinates are integers chosen uniformly and independently at random from $\{0, \ldots, 1000\}$. Then the sum $t = \sum_{i=1}^{n} s_i$ is computed, and the m bin capacities are all taken to be $\lceil (1 + \beta/100)t/m \rceil$; these computations are done componentwise. (If $\beta = 20$, for example, then the total bin capacity, in each dimension, will be 20% more than the total item size.) An instance is rejected and regenerated if it contains any single item that is too large to be placed into a bin, as such an instance is obviously infeasible. Our test instances have 6 dimensions, 18 items, and 6 bins; we generated 52 such instances for each integer value of β from 0 to 35. These instances are available at `http://www.math.cmu.edu/~bkell/6-18-6-instances.txt` or by request.

By their construction, these instances have identical bins. The ullage MDD representation can exploit this symmetry effectively to reduce the number of branches in the search tree. This is especially evident in the infeasible instances, where infeasibility must be established by some kind of exhaustive search.

The experiments were run on a 32-bit Intel Pentium 4 CPU at 3.00 GHz with 1 GiB of RAM using Windows 7 Professional. The maximum Java heap size was set to 512 MiB. We used AIMMS 3.13 with CPOptimizer 12.4 as the CP

solver and CPLEX 12.4 as the MIP solver, with their default settings. The CP model has n variables $x_1, \ldots, x_n$, each with domain $\{1, \ldots, m\}$; the assignment $x_i = j$ indicates that item i is packed into bin j. These variables are subject to d independent `cp::BinPacking` constraints. The MIP model has mn binary variables $x_{i,j}$, for $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$; the assignment $x_{i,j} = 1$ indicates that item i is packed into bin j. The MIP model also has a nonnegative "overflow" variable ω_j for each bin, representing the maximum amount by which the bin is overfull in any dimension, and there is a nonnegative "total overflow" variable $\Omega = \sum_{j=1}^{m} \omega_j$. The MIP model appears below. It is formulated as a minimization problem only because that is the form the solver requires; the constraint $\Omega = 0$ means it is really just a feasibility problem.

$$
\begin{aligned}
\min\ & \Omega \\
\text{s.t. } & \sum_{j=1}^{m} x_{i,j} = 1; \\
& \sum_{i=1}^{n} s_{i,k} x_{i,j} \leq c_{j,k} + \omega_j \quad \text{for all } k \in \{1, \ldots, d\},\ j \in \{1, \ldots, m\}; \\
& \Omega = \sum_{j=1}^{m} \omega_j; \\
& x_{i,j} \in \{0, 1\}, \quad \omega_j \geq 0, \quad \Omega = 0.
\end{aligned}
$$

We compared the performance of CP and MIP to our MDD approaches: the exact MDD (using depth-first, heuristic-driven exploratory construction), a relaxation MDD using the relaxation merge operation, and restriction MDDs using the restriction merge operation or deletion. All instances were run to completion using each method. The maximum width for the approximation MDDs was set to 5000 nodes. With this width, the approximation MDDs returned "feasible" or "infeasible" correctly in all instances except two: the restriction merge MDD returned "infeasible" incorrectly for one instance with 25% bin slack and one instance with 26% bin slack. The combination of the relaxation merge MDD and the deletion (restriction) MDD was enough to correctly solve all 1872 instances.

Fig. 2 shows a clear feasibility phase transition centered around approximately 20% bin slack, with a corresponding hardness peak. In the infeasible region, on instances having bin slack between about 2% and 22%, the average run time of the exact MDD method is consistently less than that of MIP and significantly less than that of CP (by over three orders of magnitude at 20% bin slack). On the other hand, in the feasible region, on instances having bin slack more than about 25%, CP and MIP both tend to outperform the exact MDD method. A notable exception (visible as a spike in the hardness profile) occurs at 27% bin slack, for which one of the 52 generated instances happened to be infeasible; this single infeasible instance greatly increased the average run time of CP and MIP without noticeably affecting the performance of the exact MDD.

We investigated the instances at the hardness peak, i.e., those having 20% bin slack, in more detail. A performance profile for these instances appears in

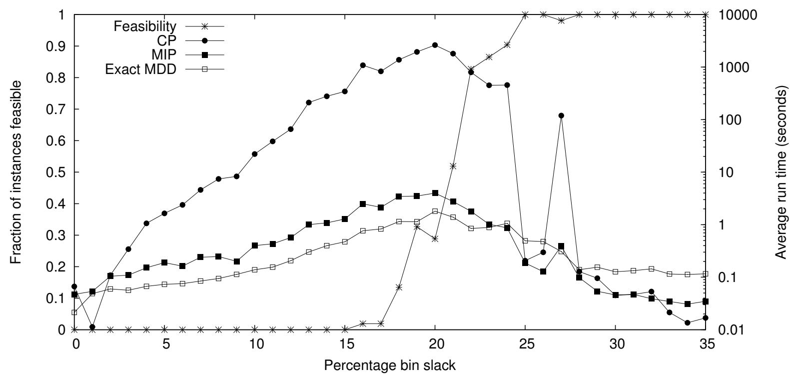


Fig. 2. Feasibility and hardness profiles for instances having 6 dimensions, 18 items, and 6 bins.

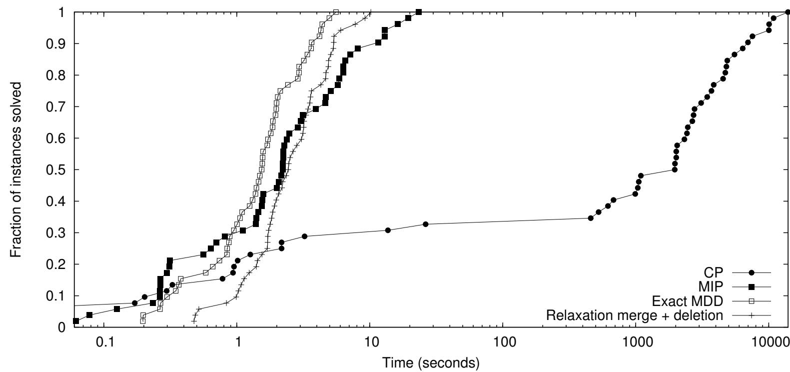
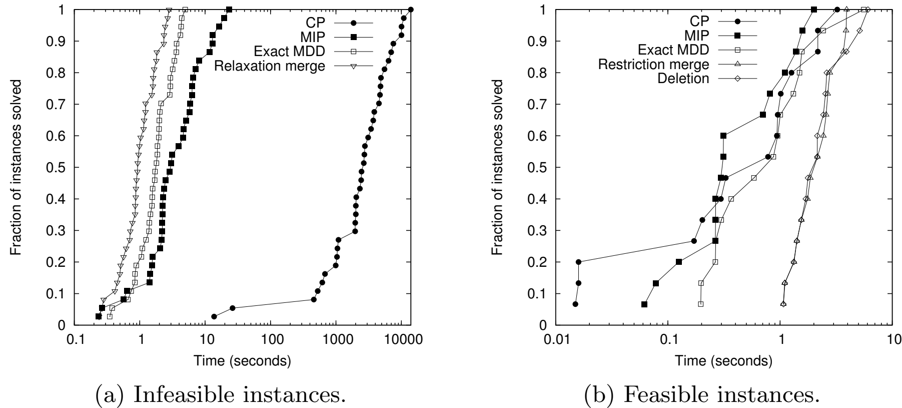


Fig. 3. Performance profile on the subset of instances having 20% bin slack.

Fig. 3, including CP, MIP, the exact MDD, and the combination of the relaxation merge MDD and the deletion (restriction) MDD. The CP solver required over 400 seconds for 35 instances (67%), taking almost 14000 seconds in the extreme case. The MIP solver did much better, solving every instance in less than 12 seconds. The exact MDD method, which solved each instance in less than 6 seconds, was faster than MIP in 32 instances (62%), while the relaxation MDD and the deletion MDD together (sufficient in all 52 instances to establish feasibility or infeasibility) were faster than MIP in 24 instances (46%).

When we look only at the 37 infeasible instances with 20% bin slack, as seen in Fig. 4a, the difference between CP/MIP and the MDD approaches becomes clearer. (Restriction MDDs do not give useful results for infeasible instances, so they are omitted from this plot merely for clarity. All of the approximate MDD methods we implemented ran about equally fast on all instances with 20% bin slack, so using a restriction MDD together with the relaxation approximately



(a) Infeasible instances.

(b) Feasible instances.

Fig. 4. Performance profiles on infeasible and feasible instances having 20% bin slack.

doubles the run time.) On the other hand, in the performance profile on the 15 feasible instances with 20% bin slack, shown in Fig. 4b (with the relaxation MDD omitted), the various methods are not as clearly separated.

The advantage of the ullage MDD representation on infeasible instances comes from its ability to exploit the symmetry among identical bins in order to reduce the number of branches taken in an exhaustive search. However, on feasible instances, our Java code, which is not particularly optimized, does not find solutions as quickly as the commercial CP and MIP solvers do. The depth-first, heuristic-driven algorithm tends to solve feasible instances more quickly than the layer-by-layer approximation algorithms, but limited-width MDDs tend to be faster than exact MDDs on infeasible instances.

6 Conclusions

Our aim was to investigate the use of MDDs for the MBP problem. We described several variations of a generic algorithm for the construction of exact and approximate MDDs representing sets of feasible solutions to CSPs, including a heuristic-driven depth-first method to construct an exact MDD and an application of a clustering algorithm to construct approximate MDDs. We also examined several techniques to work with MBP instances effectively with MDDs, including the ullage MDD representation to handle symmetry, a rounding-down technique to more reliably detect equivalent nodes, and the identification of free and dead bins to quickly recognize feasibility and infeasibility. Experimental results show that our MDD algorithms, when combined with these representation techniques, can significantly outperform currently used CP techniques and can also consistently outperform MIP.

References

1. Andersen, H.R., Hadzic, T., Hooker, J.N., Tiedemann, P.: A constraint store based on multivalued decision diagrams. Principles and Practice of Constraint Programming–CP 2007 (2007) 118–132
2. Behle, M.: On threshold BDDs and the optimal variable ordering problem. Journal of Combinatorial Optimization **16** (2008) 107–118
3. Bergman, D., Cire, A.A., Hoeve, W.-J. van, Yunes, T.: BDD-based heuristics for binary optimization. Submitted (2013)
4. Bergman, D., Hoeve, W.-J. van, Hooker, J.N.: Manipulating MDD relaxations for combinatorial optimization. In Achterberg, T., Beck, J. (eds.) CPAIOR 2011. LNCS, vol. 6697, pp. 20–35. Springer, Berlin (2011)
5. Bollig, B., Wegener, I.: Improving the variable ordering of OBDDs is NP-complete. IEEE Transactions on Computers **45**(9) (1996) 993–1002
6. Hadzic, T., Hooker, J.N., O'Sullivan, B., Tiedemann, P.: Approximate compilation of constraints into multivalued decision diagrams. In Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 448–462. Springer, Berlin (2008)
7. Heckbert, P.: Color image quantization for frame buffer display. In: SIGGRAPH '82, pp. 297–307. ACM, New York (1982)
8. Hoda, S., Hoeve, W.-J. van, Hooker, J.N.: A systematic approach to MDD-based constraint programming. In Cohen, D. (ed.) CP 2010. LNCS, vol. 6308, pp. 266–280. Springer, Berlin (2010)
9. Knuth, D.E.: The Art of Computer Programming, volume 4, fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams. Addison-Wesley (2009)
10. Lodi, A., Martello, S., Monaci, M.: Two-dimensional packing problems: A survey. European Journal of Operational Research **141**(2) (2002) 241–252
11. Martello, S., Pisinger, D., Vigo, D.: The three-dimensional bin packing problem. Operations Research **48** (2000) 256–267
12. Schaus, P., Van Hentenryck, P., Monette, J.-N., Coffrin, C., Michel, L., Deville, Y.: Solving steel mill slab problems with constraint-based techniques: CP, LNS, and CBLS. Constraints **16**(2) (2011) 125–147
13. Wegener, I.: Branching Programs and Binary Decision Diagrams: Theory and Applications. SIAM, Philadelphia (2000)