Multiple Destination Bin Packing[*]

Bram Verweij[†]

July 11, 1996

Abstract

There exists ample literature about on-line and off-line algorithms to solve the two and three dimensional bin packing problem. However, non of the off-line algorithms presented in literature are designed for the case that there are constraints on the order in which the objects have to be removed from the bin. This constraint is relevant in the case that we are loading a truck with several orders that have to be unloaded at different destinations.

We present a population-based search heuristic for finding optimally packed bins, and present the data structures needed to implement the heuristic in both two and three dimensions. The algorithm is re-entrant and can be called iteratively for each destination to pack a bin with the corresponding orders. Experimental results are given to measure the average performance of the algorithm for the two- and three-dimensional case.

1 Introduction and Literature Overview

The bin packing problem is a problem that is known to be NP-hard, but has many practical applications. In its most simple form, in one dimension, it is comparable to the subset-sum problem. For example, if we have a collection of songs on several compact discs, and we want to duplicate them on a set of finite-length tapes in such a way that we use a minimal number of tapes, we have to solve the one dimensional bin packing problem. In one dimension, exact solutions can be calculated using dynamic programming in exponential time relative to the size of the input, or by applying a fully polynomial time approximation scheme like the one of Ibarra and Kim [IK75].

More serious applications of the bin packing problem involve two and three dimensional problems, such as loading pallets on the floor of a truck, packing pallets with boxes, and loading a container or a truck with a set of boxes. The literature about these higher

[*]This research was (partially) supported by ESPRIT Long Term Research Project 20244 (project ALCOM IT: *Algorithms and Complexity in Information Technology*).

[†]This work has been done while the author was connected at ORTEC Consultants (Gouda, The Netherlands). **Email:** amverwei@cs.ruu.nl

dimensions focuses on physically placing boxes in a single bin. Because off-line algorithms can add boxes in any permutation that the constraints of the particular problem allow, each box can possibly be placed in a number of places, and boxes can be placed in different orientations, the search space of possible solutions to this problem is huge indeed. The search space for a set boxes becomes smaller when the boxes only have a limited number of distinct box types.

Known heuristic algorithms start by making some assumptions about the form of a good solution, simultaneously reducing the size of the search space taken into account even more. For example, Bischoff *et al.* [BJR95] assume the optimal solution can be found by filling the bin from the bottom upwards by stacking layers consisting of one or two blocks of a single type of box on top of an available surface (an instance of the *layering* approach). Another practical assumption for the three dimensional bin packing problem is that the boxes have to be placed with full support. Layering algorithms trivially fulfil this assumption.

Let us briefly review the available literature on the bin packing problem. A number of authors considered the two dimensional packing problem. Bengtsson [Ben82] describes a heuristic algorithm that enumerates several layouts and chooses the best it finds. Beasly [Bea85] describes a method to find exact solutions for the problem, which is suitable if the number of pieces that is to be packed is small. The two-dimensional bin packing problem is related to the two-dimensional cutting stock problem, which also is richly documented. However, as this is beyond the scope of this paper, the reader is referred to Dyckhoff and Finke [DF92].

The three dimensional bin packing problem has received attention from a number of authors, with differences that are due to the background the authors found most interesting. Haessler and Talbot [HT90] describe a heuristic for rail shipment of orders. A number of algorithms are proposed for the container loading problem. The special case in which there is only one type of box is treated by Han *et al.* [HKE89] and George [Geo92]. The algorithm by Han *et al.* uses dynamic programming to derive a solution, whereas the algorithm by George is a layering heuristic. Gehring *et al.* [GMM90] present a layering algorithm for container loading; Portmann [Por91] describes an algorithm for container loading that fills the container from the bottom up. George and Robinson [GR80] describe an algorithm for container loading that seeks to construct vertical "walls" across the container.

Ngoi *et al.* [NTC94] have a different approach for the container loading problem; they present a heuristic that uses a data structure for representing a packed container. The data structure consists of a stack of two dimensional arrays; this might make the memory requirements of the data structure a bit large.

Finally, Bischoff *et al.* [BJR95] focus the attention on the problem of loading pallets. They present a most interesting layering heuristic, and a thorough experimental evaluation. The volume utilisation achieved with this algorithm is the best in the literature known to the author. However, layering algorithms have one drawback: their performance degenerates if the input forces fragmentation of the loading surface in the early stages of the algorithm.

In this paper we consider the case that we have to load a truck with a number of orders, each order consisting of a number of boxes. The orders have to be delivered one at a time; and we do not want to unload all orders to deliver only one. A rigid approach that fulfils this restriction is to force the order that has to be delivered next to be the front most one from the back of the truck. Note, however, that this approach might very well imply an instant fragmentation of the loading surface, thus rendering a layering approach invalid. Of coarse, one can use an on-line algorithm for the problem. However, this does not exploit the freedom within an order. Multiple destination bin packing asks for an approach that combines qualities of both layering- and on-line algorithms.

We present a search heuristic that does not make any assumption about the form of an optimal solution, given that the boxes have to be placed with their sides parallel to the sides of the bin. Instead of packing a bin layer by layer, we pack our bin box by box, as if we are loading a truck from the back. Packing a bin box by box has the disadvantage that decisions about where to place a box can only be made by local criteria. This greedy approach can cause the algorithm to end up with a poor packing of the bin. Our solution to this problem is to maintain a population of packings, and discard those that are no longer interesting. Although our algorithm basically is a greedy algorithm that keeps track of a number of possible packings, the population-based approach can be compared with population-based genetic algorithms in so far as it is used as a mechanism to sample a limited portion of the search space. However, the individuals in our population are very much different from the bit strings that would be used as individuals in a genetic algorithm; as is our way of generating a new population from an old one.

To represent and evaluate individuals in our population, or partially packed bins, we use techniques from computational geometry. Especially in three dimensions these techniques are far from trivial.

The rest of this paper is organised as follows. Section 2 describes our overall approach to the bin packing problem, and formulates what kind of operations we need to perform on a partial packings data structure. Section 3 describes a data structure that we use to represent two dimensional partial packings, and Section 4 does the same for three dimensional packings. Results of experimental evaluation are presented in Section 5. We conclude with our conclusions and state some possibilities for further research in Section 6.

2 Bin Packing

This section treats our approach to solving various bin packing problems. Section 2.1 we discuss how to pack a single bin. Section 2.2 treats a heuristic to pack boxes for multiple destinations in a set of non-identical bins.

2.1 Packing a Single Bin

Given a bin, and a set of boxes with a total volume exceeding the volume of the bin, we want to find a placement of a subset of the boxes that utilises the volume of the bin as good as possible. If V is the volume of the bin, and V_b is the total volume of all placed boxes, the *volume utilisation* is defined as V_b/V. For each placed box we want to know its location in the bin.

We will maintain a population of individual solutions for the problem. Each solution represents a packing of the bin, and stores the information of all boxes placed, the front of its packing, and the boxes that still have to be placed. The *front* of a packing are those faces that are visible from the front of the bin in an axis-parallel projection. Theoretically, we can limit our search process to so called *normalised* packings, in which all boxes are shifted as far to the left-back corner of the bin as possible. However, if we only maintain the front of a packing (that also encloses empty volumes), it might be hard to find out if a box touches another one with its left side. Therefore, we limit ourselves to packings that can be made by placing boxes in a front such that they touch the front with their back and left side. We divide this search space in levels; level k contains all partial packings of the bin with exactly k boxes packed. Level 0 contains only one point of the search space, which is an empty bin. Hence, our initial population consists of only one individual, namely the empty bin.

Each iteration of the algorithm, we let the individuals in the population grow one level in the search space. This is done as follows. We calculate all packings in the search space that can be reached from the current population by placing one box in the front of an individual in the current population. A *placing* in a front is a combination of a box in a certain orientation, and a place in the front were this box fits in this orientation. We preselect c_s placings per individual of our current population, except in the early stages of the algorithm when we try to fill our population. The *depth* of a placing is the distance of its place to the back of the truck. Let m denote the maximum box dimension of all boxes in the input. The depth d of a placing in a front f is said to be *too far to the front of the bin* if there exists a placing in f that has a depth smaller than $d - m$. From the preselected placings, we remove all placings that are too far to the front of the bin. Next, we calculate the new fronts for all remaining placings, giving us an intermediate population. For each individual in this intermediate population we calculate a finger print, that is distinct for packings with different fronts with high probability. Using these finger prints, we can efficiently remove all packings with duplicate fronts. After removing all packings with duplicate fronts in our intermediate population, we just select the c_P best packings to form our new population. During the entire process we keep track of the packing with the highest volume utilisation we encounter. The algorithm is summarised in Algorithm 1. It will be discussed in Sections 3.6 and 4.4 how to calculate a good finger print; we will see this is rather easy to do.

Let us take a closer look at how to select packings. We use three criteria to evaluate a packing. The first one is the measure that we want to maximise, the volume utilisation. The second criterion is an upper bound on the volume utilisation that can be achieved from a packing. We define the *empty volume* of a packing to be the volume of the space enclosed behind the front of the packing that is not occupied by boxes. If V is the volume of the bin, and e is the empty volume of a packing, than the upper bound on the volume utilisation for this packing we use is $1 - e/V$. Our third criterion is a measure for the complexity of the front of the packing (see Sections 3.5 and 4.1). As we will see, we can express all these

```
procedure SingleBin($P$, $c_s$, $c_P$)
(* $P$ is the initial population of partial packings of the same bin,
   each containing the same number of boxes. *)
begin $done := false$;
        while not $done$ and boxes left do
        begin $P' := \varnothing$; $c := \max(\lfloor c_P/|P| \rfloor, c_s)$
                forall packings $p \in P$ do
                begin $L := \varnothing$;
                        forall box types $i$ left
                        do $L := L \cup \{$place box of type $i$ at $x$ in $p$
                                        $| \, x \in$ ExtendPacking($p$, $i$, $c$) and $x$ not too large$\}$
                        $P' := P' \cup \{$best $c$ packings from $L\}$;
                end ;
                if $P' = \varnothing$ then $done := true$;
                else Remove all packings with duplicate fronts from $P'$;
                $P := \{c_P$ best packings of $P'\}$;
        end ;
        if all boxes packed
        then return $P$;
        else  return {best packing seen};
end .
```

Algorithm 1: The Bin Packing Algorithm.

criteria in well-defined numerical quantities; which we can use to compare two packings. More explicitly, comparing two fronts can be done by comparing them *lexicographically* in some order on these criteria. Then, a different permutation of these criteria corresponds to a different comparison operator.

How do we generate all possible locations for a box in a front? We will be using geometrical algorithms to do the hard work, which will receive a dedicated section later on in this paper, so let's just globally investigate the problem at hand. We can generate all possible orientations of a new box just by rotating it and checking whether the resulting orientation is allowed. If e_p is the empty volume in a packing p, e_b the volume between the back of a new box and the front of p at some location in p, and V the volume of the bin, then the upper bound on the volume utilisation that can be achieved from p after placing the box at this place in p is given by $1 - (e_p + e_b)/V$. The complexity measure can be calculated incrementally in a similar way. The procedure for calculating box placements in a packing is illustrated in Algorithm 2.

Let us analyse the running time of our single bin algorithm. Denote the time needed to inspect a front (as in ExtendPacking) with T_1, the time needed to insert a box in a front with T_2, and the time needed to finger-print a front with T_3. Furthermore, denote the number

```
procedure ExtendPacking($p$, $i$, $c$)
(* This procedure selects $c$ extensions of packing $p$ with a box of type $i$. *)
begin forall dimensions $d$ of box type $i$
        do if $d$ is allowed as height dimension
            then forall distinct orientations $x$ of box type $i$
                            with dimension $d$ as height do
                begin Calculate all candidate places for a box with orientation $x$ in
                        $p$; for each candidate place, calculate the upper bound on the
                        volume utilisation and the complexity of the resulting packing.
                end ;
        return all candidate places found;
end .
```

Algorithm 2: Extending a packing with a box.

of boxes to be placed with n, and the number of distinct box types with m. Clearly, the procedure ExtendPacking uses $O(T_1)$ time. The outer loop of the procedure SingleBin is executed at most n times. Creating the intermediate population takes $O(c_P m T_1 + c_P c_s T_2)$ time. The size of the intermediate population is at most $c_P c_s$. Removing doubles, which includes finger-printing and sorting on finger print, takes $O(c_P c_s(T_3 + \log c_P c_s))$ time. Selection can be done in linear time. As c_P and c_s are constants, the overall time taken by the algorithm is

$$O(c_P n(mT_1 + c_s(T_2 + T_3 + \log c_P c_s))) = O(n(mT_1 + T_2 + T_3)).$$

For the two-dimensional case, we will present a data structure that allows for inspecting, inserting and finger printing a front f in $T_1 = T_2 = T_3 = O(|f|)$ time in Section 3, where $|f|$ denotes the complexity of f. Depending on the input, $O(|f|)$ can be as high as $O(n)$, making the overall complexity $O(n^2 m)$ for the two dimensional case.

For the three-dimensional case, the complexity is dominated by the time for inspecting the data structure, which can be as high as $O(|f|^2) = O(n^4)$, depending on the input (see also Section 4.3). This leads to an ominous $O(n^5 m)$ algorithm. In practice, the performance will be much better, since the abysmal situations in which $|f| = O(n^2)$ and the time needed for inspecting f becomes $O(|f|^2)$ will hardly ever occur.

2.2 Packing for Multiple Destinations

In the previous section, we discussed how to solve the bin packing problem for a single bin. Since our back ground is loading trucks we still have to face three problems, namely that the boxes that are left out by the single bin algorithm have to be packed in an other bin, that the order in which we can remove items from the bin is important, and that the bins can have different dimensions.

```
procedure MDMB-Packing(bins, boxes)
(* Packs boxes in bins. *)
begin while bins left and boxes left do
        begin Get boxes for next destination;
                repeat if no bin open
                          then Open the first bin, and create the initial population
                                  containing the new bin, and the boxes of the current
                                  destination.
                          else if last bin full
                          then Open the next bin, and create a population of packings
                                  with the new bin and the remaining boxes of the
                                  packings in the population of the last bin;
                          else  Add boxes of the current destination to all packings in the
                                  population of the current bin (without placing them);
                          Use the single bin algorithm to pack the boxes in the population;
                until all boxes of the current destination packed;
        end
end .
```

Algorithm 3: Multiple Destination, Multiple Bins Packing.

Our solution for the first two problems is to iteratively call the single bin algorithm with the boxes for a single destination; if they don't fit in the current bin, we *open* the next bin and repeat the process with the remaining boxes of the best found solution for the current bin. We ensure that boxes for a destination can be removed from the bin by loading these boxes only after all boxes for later destinations (along the route of the truck) have been packed. The order in which we open new bins is just the order in which the bins are presented to the algorithm. The pseudo code for this approach can be found in Algorithm 3.

This algorithm still does not guarantee that you will find the highest volume utilisation for the whole set of bins, since the last bin will be only partly filled with boxes. If the overall volume utilisation of the bins is to be any good, it is necessary for the sum of the volumes of all bins to be close to the sum of the volumes of all boxes. So, the problem of finding the right set of bins to pack in is very similar to the subset-sum problem.

Based on this observation, we can try to find a good set of bins by using a fully polynomial approximation scheme for the subset-sum problem like the algorithm by Ibarra and Kim [IK75] to find several feasible *combinations* of bins, sorted on increasing total volume, and then use binary search in this list of combinations to find one that has a good overall volume utilisation. Of course, we only pack those combinations of bins that are needed for the binary search.

```
procedure PackSubsetBins($bins$, $boxes$)
begin $l$ := Total volume of the boxes;
    Sort the bins on non-increasing volume;
    Pack the boxes in the sorted bins using the multiple bins algorithm;
    $u$ := Total volume of bins needed;
    Use an approximation algorithm for the subset-sum problem to generate a
      list of feasible combinations of bins with total volume in the range $[l, u\rangle$;
    Binary search in the generated list of combinations to find a
      solution to the bin packing problem with good volume utilisation.
end .
```

Algorithm 4: Finding a good set of bins.

Let us inspect this approach with a bit more detail. We only want to pack combinations of bins that have enough volume for all boxes, which gives us a lower bound on the volume of all feasible combinations. Furthermore, we can sort our bins on non-increasing volume, and use the multiple bins algorithm to pack the objects in this sorted list of bins. Then we calculate an upper bound on the total volume of the combinations of bins needed by summing the volume of the bins used. After this, we use the approximation scheme for the subset-sum problem to generate a list of feasible combinations of bins that contains an element with total volume within a factor of $(1 + \epsilon)$ of the volume of any possible combination of bins, for any ϵ. The length of this list is linear in $(N \ln V)/\epsilon$, where N is the number of bins, and V is the upper bound on the total volume of the bins (see [CLR90, p. 983]). Next we just use binary search in this list to find a combination of bins with a good volume utilisation. We use the multiple bins algorithm to pack those combinations needed for comparison by the binary search. The algorithm is summarised in Algorithm 4.

3 Two-Dimensional Packing Data Structure

In this section we give a data structure that can be used to represent two-dimensional packings, and discuss its properties. Although Section 2 primarily discussed a three-dimensional problem, it degenerates to a two-dimensional problem if all boxes and bins have the same height, and all boxes must be placed with its top side up.

Section 3.1 presents the data structure; Section 3.2 shows how to insert a box in the structure; Sections 3.3, 3.4, and 3.5 show how to generate candidate places, calculate empty volumes, and calculate complexity measures, respectively. Finally, Section 3.6 treats how to make a finger print of a front.

We will see that the presented data structure requires memory linear in the size of the front of the packing and the number of boxes in the packing; and that all necessary operations can be done in time linear in the number of segments in the front of the packing.

3.1 The Data Structure

Before presenting the data structure, let us restate our requirements based on the packing algorithms in the previous section. We have to give a data structure that allows for inserting a box in a packing, and inspecting the front of a packing. We also have to be able to retrieve the locations of the boxes in the front. Furthermore, we have to give finger prints of fronts, that should be equal not only when the packings represented by the fronts are equal, but also if only the fronts are equal. The latter requirement calls for the need of a unique representation of the front. So if the front most faces of two adjacent boxes in the front of a packing have the same depth, they should be represented together by only one segment.

The data structure we present is rather straightforward. Denote the faces of the front of a two-dimensional packing by *segments*. We store all the segments in the front in a list, in such a way that segments adjacent in the front are also adjacent in the list. For each segment, we store its start coordinate, end coordinate, and its depth. With each segment in the list, we associate a subset of the boxes that are in the packing; of course, each box in the packing occurs in exactly one subset. Together with each box in such a subset we store its location in the packing represented by the front that contains the segment. The subsets of boxes can be represented by a binary tree. Figure 5 illustrates the data structure; the segments are numbered with their segment index, and the boxes are labelled with the index of the segment to which they belong.

Obviously, the memory required by the data structure is linear in the number of segments in the front of the represented packing, and in the number of boxes in the packing. Note, that although fronts itself cannot be shared between different individuals in a population, the subsets of boxes can be shared if we keep track of the number of references (or *reference count* [Col60]) to each node in the binary trees that contain the boxes, thus limiting the memory requirements of a whole population.

3.2 Inserting a Box

When we generated new packings from an old packing, we placed boxes before the old packing at different locations. Since we possibly want to re-use the old packing with different boxes, before inserting a box in a front, we first duplicate the front, and then insert the box in the duplicated front. There is no need to physically duplicate the subsets of boxes in the packing; we can just copy a pointer to each of the subsets and increase their reference count.

To insert a box in a front, we first create a new segment for the box that starts at the left edge of the box, and ends at the right edge. Next, we locate the segment that contains the start of the new segment, and the segment that contains the end of the new segment. The start coordinate of the segment containing the end of the new segment is shifted to the right to reflect the new situation. If possible, we merge the new segment with its neighbours.

All old segments that are completely overlapped by the new one are then discarded,

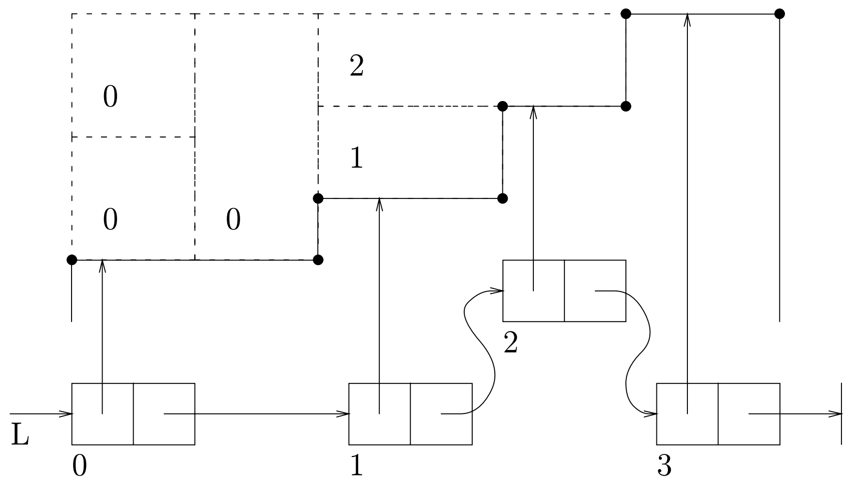
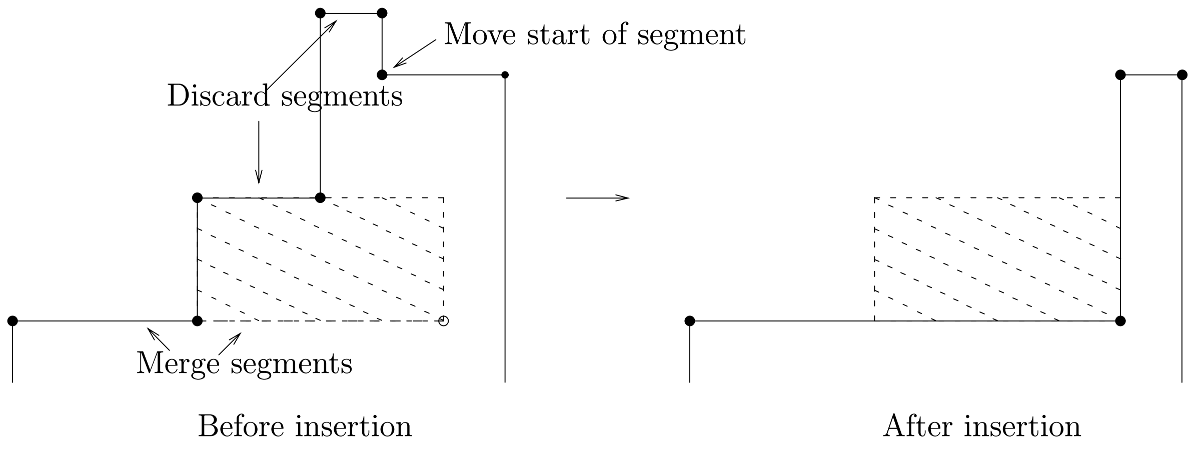


Figure 5: Data structure for representing 2D fronts.

and their subsets of boxes are added to the new segment together with the data of the placed box. The insertion procedure is illustrated in Figure 6.



On the left, the old front is shown in solid lines. The new box is shown dashed. The front of the box defines a new segment in the old front, that covers some of the old segments. The two segments that are covered completely have to be discarded; the segment that is covered partially is shortened. As the depth of the new segment equals the depth of the segment to its left, these segments have to be merged. After insertion, the situation on the right emerges.

Figure 6: Inserting a box in a front.

The running time of the insertion procedure is linear in the size of the front; this is all right since we also have to copy it, which requires at least an amount of work of equal proportion. However, it could be implemented more efficiently, if we know beforehand where in the front the new segment has to be inserted; then, only local rearrangements are needed.

3.3 Generating Candidate Places

So how do we locate all valid places, or *candidate places*, where we can place a box in a front, given the representation described above? Recall that we are interested in all places where the box has contact with its left and back side with the front, and recall that the *depth* of a segment in the front of a packing to be the distance from the segment to the back of the truck. In order to locate all candidate places, we traverse the front from left to right, and push all left-back corners on a push-up stack (which is similar to a regular stack of plates). Together with each corner, we push the depth of the neighbouring segment to the left (or the depth of the bin for the first segment), which is the *maximal allowable depth* (for the given box) at that place.

When we reach a segment with depth d, and d is larger than the depth of the previous segment, we pop all places from the stack that have a depth smaller or equal than d. For each place that is not the last popped, there are two possibilities. Either the place has room enough for the box, and it is reported as a candidate place; or the place is too small for the box, and it is discarded. For the last popped place, we have three possibilities. Either the place has room enough for the box and is reported as a candidate place, or the place is too small but the depth d is less than the maximal allowable depth for the place in which case the candidate place is moved forward to a depth of d and the new place is pushed, or the place is too small for the box and the depth d is larger than or equal to the maximal allowable depth for the place, in which case the place is discarded. When we reach the right side of the bin, we just pop the stack, and report all remaining valid places. The pseudo code is given as Algorithm 7.

The correctness from this algorithm follows from four observations: (i) All candidate places enter the stack, since all left-back corners are pushed. (ii) The process maintains a *stack invariant*: if a place is higher on the stack, it's depth in the front is smaller. If a place that is not the last popped from the stack is discarded, the stack invariant implies that we did have another place on the stack with a depth smaller than d, so moving the former place forward to depth d would invalidate it as a candidate place since the box would no longer contact the front with its left side. So, places that are discarded do not have contact with their left side. (iii) Places that are pushed on the stack do have contact with their left side, and with the most forward segment of the front that we encountered since the place was pushed. Places are only pushed, if the start of the current segment is smaller than the end of the candidate place, otherwise they are reported or discarded. (iv) After a candidate place is popped for the last time, it will be reported if it is valid. Hence, if the stack is empty after termination, all candidate places will be reported. This completes the correctness.

```
procedure ReportCandidatePlaces($f$, $b$)
(* $f$ is a front; $b$ is box that is to be placed in $f$. *)
var $S$ : stack of candidate places;
begin for $i := 0$ to #segments in $f - 1$ do
        begin if $S$ empty or depth of segment $i <$ depth top($S$)
                then Push start of segment $i$;
                else if depth segment $i >$ depth top($S$) then
                begin while $S$ not empty and depth top($S$) $\leq$ depth segment $i$ do
                        begin $x :=$ pop($S$);
                                Report $x$ if $x$ is a valid candidate place for $b$;
                        end ;
                        if $x$ not reported
                        then Push $x$ with depth of segment $i$;
                end
                else Skip segment $i$            (* Depth of segment $i =$ depth top($S$) *)
        end ;
        forall remaining candidate places $x$ in $S$
        do Report $x$ if $x$ is a valid candidate place for $b$;
end .
```

Algorithm 7: Reporting Candidate Places.

The running time is linear in the size of the front. This can be seen as follows: We traverse the front only once. If a place is pushed on the stack, we charge this to the current segment. Since we only push at most two places per segment as we traverse the front, the number of push operations is at most twice the number of segments, and the number of pop operations equals the number of push operations.

3.4 Calculating Empty Volumes

When placing boxes, we would like to place them in such a way that their back is fully against other boxes, or the back of the bin. This is because empty space that is packed behind a box is no longer available for placing boxes. In most cases it will be inevitable that there emerges empty volumes, which makes it a good criteria for distinguishing interesting box placements from less interesting ones, or even for comparing different packings. If you subtract all empty volumes in a packing from the volume of the bin, you have a solid upper bound on the volume that can be achieved with a packing. Given a front, a box, and a set of candidate places, we want to know for each candidate place how much empty volume it creates. First, we sort the candidate places on increasing order of their start segment. This can be done in time linear in the number of segments using counting sort [CLR90, pp. 175–177]. Next, for each segment, we calculate the volume behind it, and the cumulative

```
procedure CalculateEmptyVolumes($f$, $C$, $b$)
(* f is a front, $b$ a box, and $C$ a set of candidate places for $b$ in $f$. *)
var $vol[]$ : array of float;      (* Cumulative volumes for each segment. *)
begin for $i := 0$ to #segments in $f - 1$
        do $vol[i]$ := volume behind segment $0, \ldots, i$;
        for $i := 1$ to #segments in $f - 1$
        begin Let $l$ and $r$ be the left and the right of box $b$ at place $x$;
                Calculate cumulative volume behind $f$ left of $l$ and $r$ using $vol$ and $f$;
                Empty volume of $x$ := volume behind $x$
                                     $-$ cumulative volume behind $f$ left of $r$
                                     $+$ cumulative volume behind $f$ left of $l$;
        end
end .
```

Algorithm 8: Calculating Empty Volumes.

volume behind the front from left to right. For each candidate place, we can now calculate the empty volume caused by a box at this candidate place, by subtracting the volume behind the packing and the box place from the volume behind the back of the box. These volumes can all be computed in O(1) time using the cumulative volumes. Concluding, we can calculate all empty volumes in time linear in the number of segments and the number of candidate places, which is linear in the number of segments. Pseudo code for the algorithm is given as Algorithm 8.

3.5 The Complexity Measure

When we place boxes of different metrics in a packing, the front of the packing will change with each box placement. More precisely, new faces against which a box can be placed in a later stadium will be created. After several box placements, the front can become fragmented. This might cause subsequent boxes to enclose large empty volumes, and thus reduce the volume utilisation of the resulting packing. So, a fragmented front is something to avoid if possible.

We propose the number of *2D-vertices* in a front as a measure for its fragmentation. A *vertex* in a two dimensional front exists on the intersection between two edges in the front; an *edge* in a front is either a segment, or the side between the end of a segment and the start of its succeeding segment.

Given a front f, a box, and a number of candidate places, we want to know the number of vertices in the new fronts that emerge after placing the box at any of the candidate places in f. It is easy to compute these measures from the number of vertices in f, and the differences in 2D-vertices caused by the new box. These differences can be computed in time linear in the number of segments in f, similar to the calculation of the empty volumes in the previous section. We sort the candidate places on start segment, calculate the cumulative number of vertices, and report for each candidate place the number of new vertices at the edges of the box minus the number of vertices that are behind the box.

3.6 Finger-printing Fronts

We now know how to generate candidate places for a given box in a front, and how to calculate the newly created empty volume and difference in vertices of a box at a given place. Algorithm 1 required one more operation on the front data structure: Given a front, we want to know a finger print of it; the finger print is needed for a cheap comparison between different fronts. In this section we will derive a finger print that can be computed in time linear in the size of the front; furthermore, we will show that we can easily bound the probability that two physically different fronts have the same finger print by n^{-k} for any constant $k > 0$, and $n > 0$.

The basic idea for a finger print that fulfils the requirements stated above is to have an array of k independent hash functions, each computable in linear time and having a probability on collision of n^{-1}. Given a front, we make a sequence of bytes $s = (b_i)_{i \geq 0}$ by traversing the front from left to right, and adding the x- and y-coordinate of the start of each segment to the sequence. We take $b_i = 0$ for $i > 2 \cdot \#\text{segments in the front}$. Note, that this is a bijective function, and that the generated sequence is exactly twice as long as the number of segments. Next, we choose a prime number p larger than n and the upper bound on the keys. Finally, for hash function $h_j(\cdot)$ we generate a sequence of random bytes $(a_{ji})_{i \geq 0}$ in the range of $[0, p\rangle$. Then, the hash functions $h_j(\cdot)$ are given by

$$h_j(s) = \sum_{i \geq 0} a_{ji} b_i \pmod p, \qquad \text{for } 0 \leq j < k, \text{ and } s = (b_i)_{i \geq 0}.$$

That these hash functions indeed satisfy the required bound on the probability of collision is shown by Cormen *et al.* [CLR90, p. 231]. The random bytes have to be generated only once; and only for those i that have $b_i \neq 0$. To avoid unnecessary copying of the random bytes, the size of the table containing them should be increased by a constant factor each time the table is relocated. The hash functions can be evaluated in a single pass over the front, so calculating them takes time linear in the size of the front.

4 Three-Dimensional Packing Data Structure

In this section, we will generalise the two-dimensional data structure presented in the previous section to three dimensions. Three-dimensional bin packing is a lot more complex, as can be seen from the following observations. First of all, in two dimensions, any orientation of a box was allowed; in three dimensions it is natural to have boxes that have to be placed with some side up. Further, in two dimensions a candidate place for a box is good if the box fits in the space; in three dimensions we not only want to know whether the box fits in, but we also require that its bottom is fully supported by other boxes or by the floor of the bin. And last but not least, the front of a two dimensional packing is essentially one dimensional, which allowed for a natural order over the segments in a front from left to right. This was crucial for deriving our linear algorithms. However, the front of a three dimensional packing is essentially two dimensional, and we couldn't come up with any ordering over the cells of a two dimensional front that allows for linear algorithms.

These observations lead us to the following strategy. Instead of looking for linear algorithms, which probably don't even exist, we look for algorithms that operate in $\mathrm{O}(n \log n + k)$ time, where n is the size of the front, and k is proportional to the amount of result given (such as the number of reported candidates). So, in the process of inserting a box, we will sort its border. We will use plane sweep algorithms for inspecting the data structure. For the finger-printing, we will sort edges to make a unique sequence of bytes for a front.

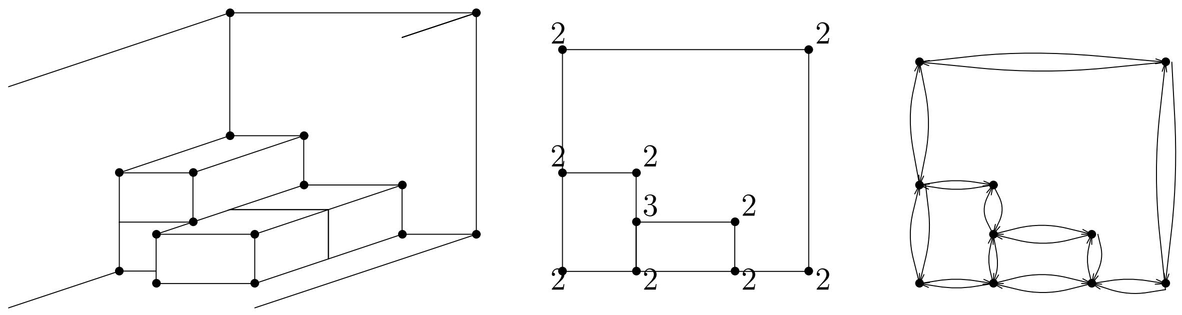
The rest of this section is organised as follows: Section 4.1 presents the data structure; in Section 4.2 we show how to insert a box. Section 4.3 presents a one-pass plane sweep algorithm that reports all candidate places together with their corresponding empty volumes and differences in complexity. Concluding this section, Section 4.4 shows how to finger print a front represented using the data structure.

4.1 The Data Structure

We use a standard edge-representation for connected two-dimensional graphs called *doubly connected edge list* (dcel) to represent the front of a three-dimensional packing. The same structure is described by de Berg *et al.* [BKOS94, Chapter 2.2]. We extend it to contain the data of the already placed boxes similar as described for the two dimensional case.

In the front of a packing, we distinguish vertices, edges and cells. The cells are the two dimensional regions against (and on top of) which boxes can be placed. A cell is bounded by edges, which are straight horizontal or vertical line segments. In each cell we store the depth of the cell, measured from the back of the truck; adjacent cells have different depth (there is one exception, which we will discuss later in this section). Each edge connects two vertices. To eliminate ambiguities, an undirected edge is split into two directed twin half-edges; each half-edge is adjacent only to the cell to its left. As a consequence, half-edges around any cell except the outer cell form a cycle oriented in anti-clockwise order; the half-edges around the outer cell form a cycle oriented in clockwise order. Each half-edge contains a pointer to its incident cell, its twin half-edge, its preceding and succeeding half-edge around its adjacent cell, and to the vertex it originates from. Each cell contains a pointer to one of its incident half-edges, and contains depth and boxes information (just like the segments in Section 3.1); each vertex contains a pointer to one of its incident edges and stores its coordinates. To allow for easy iteration over all edges, cells, and vertices they are kept in separate doubly linked lists. To complete the data structure, we also store a pointer to the outer cell.

Using this data structure, an empty truck can be represented with a dcel that consists of only two cells, four edges (or eight half-edges), and four vertices. The inner cell has depth 0; the outer cell has the same depth as the front of the truck. A more complicated example of how the data structure represents a partially packed truck is given in Figure 9,



(a) (b) (c)

a. A bin with four boxes placed. The vertices in the front of the bin are shown. b. The front of the bin in (a). The numbers show the number of 3D-vertices projected on each vertex in the front. c. The dcel representation of the front in (b).

Figure 9: Three dimensional packing data structure.

which shows (a) a parallel projection of a bin with 4 boxes placed, (b) its front, and (c) the dcel representation of the front. Since we are placing boxes at lower left-back corners in the front, and we require that the cell on top of which a box is placed has a larger depth as the front of the box we place, the front will remain connected as long as the boundary of the outer cell remains connected. For this reason, we allow cells in the front that are adjacent to the outer cell to have the same depth as the outer cell.

Again, we will be judging packings based on their volume utilisation, an upper bound on the volume utilisation that still can be realised, and the complexity of the front. We measure the complexity of a front with the number of *3D-vertices* in the front; a vertex in the front exists on the intersection of three or more faces in the three dimensional front. We include those faces in the front that are parallel to the sides of the truck, just as we did in the two dimensional case.

4.2 Inserting a Box

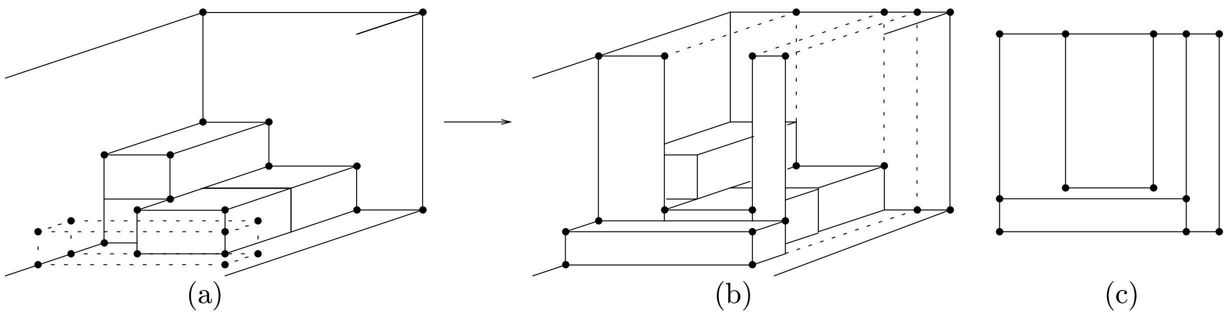
When we insert a box in the front of a packing, we have to modify the front data structure to reflect the new box. This boils down to inserting a rectangle in a dcel, discarding those parts of the dcel that are interior to the inserted rectangle. Denote all cells that intersect a rectangle or are adjacent to its boundary by the *zone* of the rectangle. Suppose, we are given a rectangle R and a cell c that intersects R. First, we locate all edges that are in the zone of R by searching the dcel starting from c. Let E be the set of edges in the zone of R. Next, we clip the dcel against R similar to the Cohen-Sutherland line clipping algorithm [FDF$^+$90, pp. 103–107] as follows: For each edge e in E, we do the following. If both e's adjacent vertices are outside R, e enters R at one side, and leaves it at the opposite side. In this case, we create two new vertices on the boundary of R, shorten e such that it goes from its starting vertex to the boundary of R, and insert a new edge between the opposite vertex on the boundary of R and e's end vertex. If only one of e's end vertices is inside R, we create a new vertex on the intersection between R and e, and shorten e such that it goes from its vertex outside R to the newly created vertex. All cells from which none of its adjacent edges (partially) survive the clipping operation are completely contained in R and are discarded, their sets of boxes are added to the set of boxes of the new cell.

So now, we punched a hole in the dcel where the box is to be inserted. We can now insert the rectangle. We start this with inserting the corners of the rectangle if they are not already present in the dcel. Next, we sort all new vertices on distance along the boundary of R in counter-clockwise order to the lower-left corner of R. We create new edges between all adjacent vertices, and we create and insert a new cell that forms the body of our rectangle. Finally, we traverse the edge of the new cell, and make sure that all pointers in the data structure reflect the new local situation of the cell. This also includes creating new cells for all cells that are split in several parts by R.

After inserting the new cell, we have to discard all edges between cells that are of equal depth, except if one of both cells is the outer cell. We do this by traversing all edges on the boundary of the new cell, and comparing the depth of the cell c at the other side of the current edge with the depth of our new cell. When the depth of c is equal to the depth of our new cell, and c is not the outer cell, we discard all edges on the common boundary. If this operation causes a vertex to lie on a straight line, this vertex is discarded and its incident edges are merged. To conclude this operation, all references to c are replaced by references to the new cell; c is discarded, and its set of boxes is added to that of the new cell.

What is the complexity of this insertion operation? Clearly, the insertion procedure only works in the zone of R. Denote the complexity of the zone of R by k. All operations involved in the insertion procedure except the sorting of new vertices traverse some part of the zone of R and do $O(1)$ local modifications. In total, this can be done in $O(k)$ time. The sorting of the new vertices can take up to $O(k \log k)$ time, so insertion can be done using $O(k \log k)$ time.

Until now we ignored a very complicating constraint, namely, that we want to place the boxes with full support. This means that not only do we have to make sure that a candidate place its width is sufficient for the box we want to place, but also that the surface on top of which it is placed is filled with boxes. However, our choice to treat a front as essentially two dimensional implies that we lose all information about the surface on top of a cell in the front! Consider the case that we place a high box b before a low box. The top of b does not touch the cell c behind it; so we are not allowed to place a box directly against c. However, there still is a possible location in front of c, namely on top of b. To reflect this, we advance the depth of the front before c by placing a dummy box that goes up to the top of the bin and has depth zero on the back-most top edge of b; its empty volume is added to the empty volume of b. In general, after inserting a box we place dummy boxes on top of it for all intervals that the box did not have contact with the cell in the front behind it. An example of the insertion procedure is given in Figure 10.



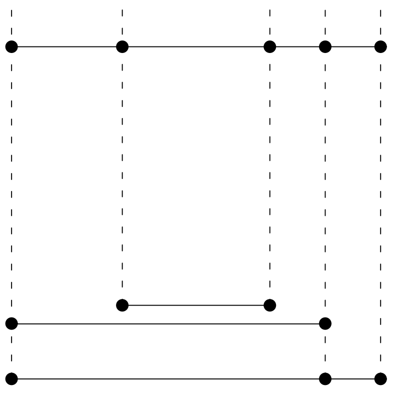
a. The packing in which we want to insert the dashed box. b. The packing that emerges after inserting the box in (a). Note the two dummy boxes that are needed to maintain the full support constraint. The front of the packing is completed with the dashed lines, and the bold vertices are the vertices in the front. c. The projection of the front of (b). Note the concave cell.

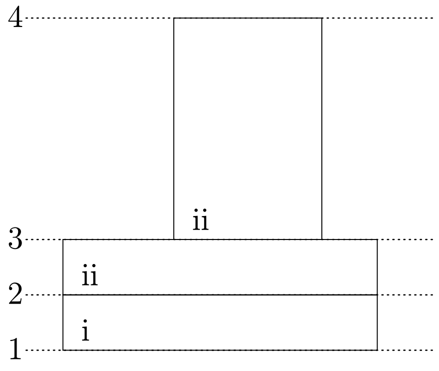
Figure 10: INSERTING A BOX.

4.3 Inspecting the Data Structure

In two dimensions, we presented optimal algorithms for calculating candidate places, their empty volumes, and their differences in complexity. These algorithms made use of the order over the segments in the one dimensional front. Since we do not have such an order in the three dimensional case, we propose to use plane sweep algorithms instead. A plane sweep algorithm sweeps the plane with a imaginary horizontal line from below to above the front of a packing, maintaining a sweep line invariant and the status of the sweep line. The sweep line invariant is that all objects of interest that are completely behind the sweep line are fully processed; the status of the sweep line reflects the objects of interest that intersect the sweep line. All places where the status of the sweep line changes are events of the plane sweep.

Events that still have to be processed are kept in a priority queue; any data structure which allows for an $\mathrm{O}(\log n)$ time (where n denotes the number of events) insertion and extraction of the minimum event is suitable, such as a heap [Wil64]. We will be maintaining two different status structures to maintain the status of our plane sweep; one, that contains the candidates that intersect the sweep line, and one that contains the depth of the cells that intersect the sweep line. The operations that a status structure has to support are location, insertion, deletion, and traversion of the local neighbourhood of an object. Any data structure that supports these operations in $\mathrm{O}(\log n)$ time (n denotes the size of the status structure) can be used, the authors favourite is a skip list [Pug90]. Although insertion and location only operate in expected $\mathrm{O}(\log n)$ time, it has tight code and traversal and deletion code can be made as efficient as code for circular doubly-linked lists. Actually, the cell status will contain the start and end positions of the most recent horizontal edge the sweep-line passed, together with the depth above and below these horizontal edges.


(a)



(b)

a. The cells that enter the edge status for the front in Figure 10c. Each edge causes an edge event. b. The candidate placements (numbered i and ii) that enter the candidate status structure if we insert the same box as in Figure 10a in the front of Figure 10c. The events that concern the candidate status are: 1. the start event of (i); 2. the top event of (i) and the start event of (ii); 3. the top event of (ii), which inserts the dummy on top; and finally 4. the end event of (ii).

Figure 11: EDGE AND CANDIDATE STATUS STRUCTURES.

Suppose we are given a front of a three dimensional packing and the dimensions of the box we want to insert. The events that are relevant for our plane sweep fall in two categories: events that affect the cell status structure, and events that affect the candidate-places status structure. The status of the cell status structure changes with every horizontal edge we pass, so every horizontal edge defines an *edge* event. Every corner in the front that is a lower left-back corner and has enough vertical space for our box is a potential candidate place, so it defines a *candidate place* event. If a candidate place has room enough for the given box, it generates a *candidate start* event, which in its turn is terminated by a *candidate top* event. When processing the top of a candidate, we will insert dummy candidates in all segments that the top of the candidate does not touch the cell behind it. These dummy candidates are terminated by *candidate end* events at the top of the bin. Finally, for each candidate, we want to know how many vertices of the (three dimensional) front fall into its interior. For this purpose, every vertex generates a *vertex* event. If different events have the same y-coordinate, the order of processing is edge events first, than candidate place events, followed by candidate start events, vertex events, candidate top events and candidate end events. In this way, we make sure that all candidate start events are generated properly, and that a vertex is registered for all candidates in which interior it falls. Figure 11 illustrates the edge and candidate status structure.

```
procedure HandleCandidatePlace($S$, $x$, $y$)
(* $S$ is the cell status structure; $(x, y)$ is the coordinate of the candidate place. *)
begin cell := Cell at position $x$ in $S$;
    depth := 0, space := $\infty$;
    while cell starts before the end of the box at height $y$ do
    begin if (depth above cell) > depth then adjust depth;
        if (depth below cell) < space then adjust space;
        cell := Succeeding cell of cell in $S$
    end ;
    if box fits at candidate place
    then Report candidate and generate candidate start event at $(x, y)$
end .
```

Algorithm 12: Handling Candidate Place Events.

Handling candidate place events. To handle a candidate place event, we locate the x-coordinate of the start of the candidate in the cell status. We calculate the maximum depth of all cells behind the candidate, which gives us the depth of a box placed at this location, and the minimum depth of all cells below the candidate, which gives us the space our box has. If this space is large enough for the box we report it as a candidate place. The procedure is summarised in Algorithm 12.

Handling vertex events. Vertex events are even easier to handle as candidate place events: if w_{box} is the width of the box, and x the x-coordinate of the vertex, we just locate $x - w_{box}$ in the candidate status structure, traverse the status until we found all candidates which have x interior, and subtract the number of 3D-vertices projected on the vertex in the front from the differences in 3D-vertices of each candidate. The number of 3D-vertices projected on a vertex in the front can be calculated using the depth of the cells around the vertex. The pseudo code of this procedure is given as Algorithm 13.

Handling edge events. So how do we handle edge events? This is slightly more complicated, and asks for a more structured approach. Suppose we are given a horizontal edge e. We have to report the number of 3D-vertices on all intersections of candidates in the candidate status structure with e (except when e coincides with the top of the candidate, which is processed by the candidate top event), and update the cell status structure. Furthermore, we have to report all empty volumes that involve the part of the cell status structure we modify. To this end, we traverse the edge with a point x from left to right. We keep track of the cell in the cell status structure that contains x (including the right edge of the cell and excluding the left) by stopping at all places where a new cell starts in

```
procedure HandleVertexEvent($S$, $x$, $y$, $v$)
(* $S$ is the candidate status structure; $(x, y)$ is the coordinate
   of the vertex; $v$ its number of 3D-vertices. *)
begin $cand$ := First candidate with start $\geq x - w_{box}$ in $S$;
      while start candidate $\leq x$ do
      begin Subtract $v$ from the complexity measure of $cand$;
            $cand$ := Candidate succeeding $cand$ in $S$
      end
end .
```

Algorithm 13: Handling Vertex Events.

the cell status structure, and maintaining the following invariant:

$$\begin{aligned} I_{edge} \;=\; & \forall \text{cells } c \text{ that start} \leq x : \text{empty volumes involving } c \\ & \quad \text{reported, and } c \text{ shortened, deleted, or split as applicable} && (1) \\ \wedge\; & C = \{\forall \text{candidates that start in the interval } [x - w_{box},\, x]\} && (2) \\ \wedge\; & \forall c \in C : \#\text{3D-vertices at intersections of } c \text{ with } e \text{ reported.} && (3) \end{aligned}$$

As initialisation, we set x on the start of the edge, and construct C using the candidate status structure. Each time we enter a new candidate in C, we report its number of real vertices at its intersections with e. This validates our invariant. Upon termination, clause (1) of I_{edge} implies that we reported all empty volumes below e, and erased the area of the cell status where the cell above by e is to be inserted. Clause (2) of I_{edge} implies that all candidates that intersect e enter C, which in turn implies that upon termination clause (3) ensures that we reported all 3D-vertices. Finally, we just insert the new cell in the cell status structure. Together with I_{edge} this implies that we made all necessary modifications to the edge status structure, which establishes the correctness. Pseudo code for this procedure is given in Algorithm 14.

Handling candidate start events. The next event we consider is the candidate start event. Obviously, we have to insert the new candidate in the candidate status. We also have to report the number of 3D-vertices for all intersections of vertical edges with the bottom of the candidate. As vertical edges only exist on the boundary of cells, we can retrieve all the information we need from the cell status structure. So we traverse the bottom of the new candidate with a point x, stopping at all places where a new cell starts in the cell status structure. For each place we stop, we determine the depths around x and report the number of 3D-vertices. Finally, we generate a candidate top event for the new candidate, and insert it in the priority queue. The procedure is illustrated in Algorithm 15.

```
procedure HandleEdgeEvent($CS$, $DS$, $e$)
(* $CS$ is the candidate status structure, $DS$ is the cell status
   structure (maintaining the *depth* of the front below the
   sweep line); $e$ is the edge that caused the event. *)
begin $x$ := Start of $e$;
        $cell$ := Cell containing $x$ in $DS$;
        $C$ := $\{\forall$candidates $c \in CS$ that start in the interval $[x - w_{box}, x]\}$;
        forall $c \in C$
        do Report number of 3D-vertices at the intersections of $c$ with $e$;
        while $x <$ end of $e$ do
        begin $x$ := min(start of cell succeeding $cell$ in $DS$, end of $e$);
                Add all candidates $c \in CS$, $c \notin C$ that start in the interval
                 $[x - w_{box}, x]$ to $C$ and report the number of 3D-vertices at
                 the intersections of $c$ with $e$;
                forall $c \in C$
                do Report empty volume involving $c$ and $cell$;
                Remove all candidates starting before $x - w_{box}$ from $C$;
                Remove part of $cell$ covered by $e$ from $DS$;
                $cell$ := Next cell in $DS$
        end ;
        Insert cell above $e$ in $DS$
end .
```

Algorithm 14: Handling Edge Events.

Handling candidate top events. The processing of the top of a candidate c involves three different actions: we have to report all empty volumes that haven't been reported by an edge event, we have to insert dummy candidates at all segments that where the top of c does not touch the front of the packing, and we have to report the number of 3D-vertices at the top of c, taking the dummy boxes into account. We again traverse the top of c with a point x, stopping at all occasions where a new cell starts in the cell status structure, and maintain the following invariant:

$$\begin{aligned} I_{top} \;=\;& \forall \text{ 3D-vertices} \leq x \text{ are reported} && (4)\\ \wedge\;& \forall \text{ cells } \mathit{cell} \leq x\text{: empty volume } V \text{ between } \mathit{cell} \text{ and } c \text{ reported} && (5)\\ \wedge\;& \forall \text{ completed segments } s \leq x \text{ with } V > 0\text{: dummy inserted at } s. && (6) \end{aligned}$$

To make the invariant initially true, we set x to the left of the top of c, and report the number of 3D-vertices at x. Upon termination, part (4) of I_{top} implies we reported all 3D-vertices, part (5) implies we reported all empty volumes, and part (6) implies that we inserted all dummy candidates. Finally, removing c from the candidate status structure completes the processing of a candidate top event. The procedure is summarised in

```
procedure HandleCandidateStartEvent($CS$, $DS$, $c$)
(* $CS$ is the candidate status structure, $DS$ is the cell status
   structure, $c$ is the new candidate. *)
begin $x$ := Start of $c$;
      $cell$ := Cell containing $x$ in $DS$;
      Report 3D-vertices of $c$ at $x$;
      while $x <$ end of $c$ do
      begin $cell$ := Cell succeeding $cell$ in $DS$;
            $x$ := min(end of $c$, start of $cell$);
            Report 3D-vertices of $c$ at $x$;
      end ;
      Insert $c$ in $CS$;
      Generate candidate top event for $c$
end .
```

Algorithm 15: HANDLING CANDIDATE START EVENTS.

Algorithm 16.

Handling candidate end events. Candidate end events are slightly easier to handle than candidate top events, since 3D-vertices can only occur at the corners of the event, and

```
procedure HandleCandTopEvent($CS$, $DS$, $c$)
(* $CS$ is the candidate status structure, $DS$ is the cell status
   structure, and $c$ is the candidate which top caused the event. *)
begin $x$ := Start of $c$;
      $cell$ := Cell containing $x$ in $DS$;
      Report 3D-vertices at $x$;
      while $x <$ end of $c$ do
      begin $x$ := min(start of cell succeeding $cell$ in $DS$, end of $c$);
            Report empty volume left of $x$ and 3D-vertices at $x$;
            if $x$ is start of dummy candidate then remember $x$
            else if $x$ is end of dummy candidate then insert dummy candidate;
            $cell$ := Cell succeeding $cell$ in $DS$
      end ;
      Remove $c$ from $CS$
end .
```

Algorithm 16: HANDLING CANDIDATE TOP EVENTS.

```
procedure HandleCandidateEndEvent($CS$, $DS$, $c$)
(* $CS$ is the candidate status structure, $DS$ is the cell status
   structure, and $c$ is the (dummy) candidate that caused the end event. *)
begin $x$ := Start of $c$;
        $cell$ := Cell containing $x$ in $DS$;
        Report 3D-vertices at $x$;
        while $x <$ end of $c$ do
        begin $x$ := min(start of cell succeeding $cell$ in $DS$, end of $c$);
                Report empty volume left of $x$;
                $cell$ := Cell succeeding $cell$ in $DS$
        end ;
        Report 3D-vertices at $x$, and remove $c$ from $CS$
end .
```

Algorithm 17: Handling Candidate End Events.

we do not have to insert dummy boxes. So, we first report the number of 3D-vertices at the top-left corner of the dummy candidate, then traverse the top of the dummy candidate with a point and report all empty volumes we pass, and then report the 3D-vertices at the top-right corner of the dummy candidate. Algorithm 17 shows the pseudo code of the procedure.

We have know presented all ingredients of our plane sweep algorithm, so lets analyse its running time. Denote the size of the front we want to process by n. We will need a bound on the number of dummy candidates generated if we try to fit a box in a front, which we give first. As a candidate place is defined by a unique lower left-back corner in the front, and there is at least one upward vertical edge leaving this corner, we charge the leftmost and rightmost dummy candidate on top of a certain candidate to this unique edge. For all remaining dummy candidates we proceed as follows. For a dummy candidate c, let I_c be the horizontal interval it spans. Let E_c containing all horizontal edges in the front below c whose horizontal span intersects I_c and that have the same depth as the back of the candidate causing c or, if no such edge exists, the same height as the bottom of the candidate causing c. We charge c to the leftmost edge in E_c. We claim that each edge in the front is charged at most two dummy candidates. This can be seen as follows. Focus on any edge e. If e is vertical, then e belongs to a lower left-back corner which defines at most one candidate place for the given box, with at most one leftmost and one rightmost dummy candidate, which are exactly two dummy candidates. So suppose e is horizontal. Dummy candidates charged to e are caused by candidates situated either on top or in front of e. We argue that there can be at most one candidate c on top of e that causes a dummy candidate that it charges to e as follows. If any of the dummy candidates charged to e are caused by c, then these are not the leftmost and rightmost ones of c. So, e is bounded by two boxes in the front, and the back of c touches both. As there can be only one lower left-back corner that causes such a candidate place, c is unique, and so is its dummy charged to e. A similar argument holds for candidates in front of e, which establishes our claim. It follows that the number of generated dummy candidates is $\mathrm{O}(n)$.

Denote the complexity of the front behind and above the i^{th} reported candidate by k_i, and let $k = \sum_i k_i$. All generated events can be associated with some part of the front. This is obvious for edge, candidate place and vertex events. The number of candidate start events is at most the number of candidate place events, and so is the number of candidate top events. So, the time taken by all operations on the priority queue is $\mathrm{O}(n \log n)$.

The running time used for locating cells in the cell status structure is $\mathrm{O}(n \log n)$, since it happens once for each event and takes $\mathrm{O}(\log n)$ time. The running time used for locating candidates in the candidate status structure is $\mathrm{O}(n \log n)$ time, since it is done for each event, takes $\mathrm{O}(\log k)$ time, and k is polynomial in n. The time needed for reporting empty volumes and 3D-vertices for candidate i and the dummy candidates on top of it takes $\mathrm{O}(k_i)$ time. Summing up, the time complexity of the plane sweep procedure is $\mathrm{O}(n \log n + k)$. Unfortunately, k can be quite large, possibly even $\mathrm{O}(n^2)$. This gives us the theoretical bound on the whole plane-sweep of $\mathrm{O}(n^2)$.

4.4 Finger-printing Fronts

The last operation we need before we can use our three dimensional data structure for bin packing is a reliable finger print of our three dimensional front. Recall our approach for tackling this problem: given a front, we used a bijective function to generate a unique sequence of bytes, on which we applied k universal chosen hash functions to give us our finger print.

The only thing we have to do is make a suitable bijective function. Fortunately, this is quite easy. We start by sorting the half-edges in the front in non-decreasing order. Half-edge e is larger than half-edge f if the origin of e is larger than the origin of f, or e and f have the same origin and the destination of e is larger than the destination of f. A vertex (origin or destination) v is larger than a vertex w, if the x-coordinate of v is larger than the x-coordinate of w, or v and w have the same x-coordinate and the y-coordinate of v is larger than that of w. Having sorted the half-edges, we build our sequence $s = (b_i)_{i \geq 0}$ by adding for each half-edge the x- and y-coordinate of the origin, the x- and y-coordinate of the destination, and the depth of its incident cell. Note that we added exactly five bytes to s for each half-edge in the front. We take $b_i = 0$ for $i > 5 \cdot \#\text{half-edges}$.

Next, we choose a prime number p larger than the total number of boxes n we have to place and the upper bound on the keys of the sequence. Finally, for hash function $h_j(\cdot)$ we generate a sequence of random bytes $(a_{ji})_{i \geq 0}$ in the range of $[0, p\rangle$. Then, the hash functions $h_j(\cdot)$ are again given by

$$h_j(s) = \sum_{i \geq 0} a_{ji} b_i \pmod p, \qquad \text{for } 0 \leq j < k, \text{ and } s = (b_i)_{i \geq 0}.$$

The interesting feature of these finger prints is, of course, that the probability that two physically distinct fronts have the same finger print is bounded by n^{-k}. The most expensive step in the calculation of the finger print is sorting the half edges, which makes the time complexity for finger-printing a front containing m half-edges $\mathrm{O}(m \log m)$.

5 Experimental Results

In this section we give an impression of the performance of the bin packing algorithms presented in Section 2. All experiments were run an a 33 MHz Hewlett Packard workstation (Model 715), except for the ones reported in Figure 21 which were executed on a 60 MHz Hewlett Packard workstation (Model 712). In Section 5.1 we evaluate the performance on two-dimensional problems using the data structures of Section 3. In Section 5.2 we evaluate the performance on three-dimensional problems using the data structures of Section 4.

5.1 Results for Two Dimensional Bin Packing

The results presented in this section were obtained by the bin packing algorithm in Section 2, with only one small refinement in the selection process. Instead of selecting lexicographically on all 12 permutations of volume utilisation, upper bound on volume utilisation and complexity, we defined five different equivalence classes within each of the 12 permutations, and used the resulting 30 operators to select. (We describe this refinement for the three dimensional case with more detail in the next Section.)

We generated the box types at random from the domain $([425, 850] \times [280, 700])$. The truck size was kept constant at $(2000, 7000)$. After fixing the box types for a certain input we generated the actual boxes by picking a random (order, type) pair and adding a box of the chosen type to the chosen order, while the total volume of the chosen boxes was smaller than that of the truck. We carried out experiments with four different orders per input, and k distinct box types per order, with k in 3, 4, 5, 8, and 10. For each value of k, we executed the algorithm on hundred different random inputs, taking $c_P = 120$ and $c_s = 2$.

The quality of the solutions returned by the algorithm was quite good, as shown in the left of Figure 19. Surprisingly, the number of box types per order has hardly any influence on the average volume utilisation of the output. This might be caused by the fact that the boxes were chosen relatively small, compared to the size of the truck. Figure 18 gives a typical packed bin calculated by the algorithm.

Furthermore, we carried out experiments with three different box types per order and k orders, with k in 1, 2, 4, 8, and 16. For each value of k, we executed the algorithm on hundred different random inputs, taking $c_P = 120$ and $c_s = 2$. The results are shown in the left of Figure 20. As was to be expected the quality decreases when the number of orders increases, but only slightly for two dimensional bin packing.

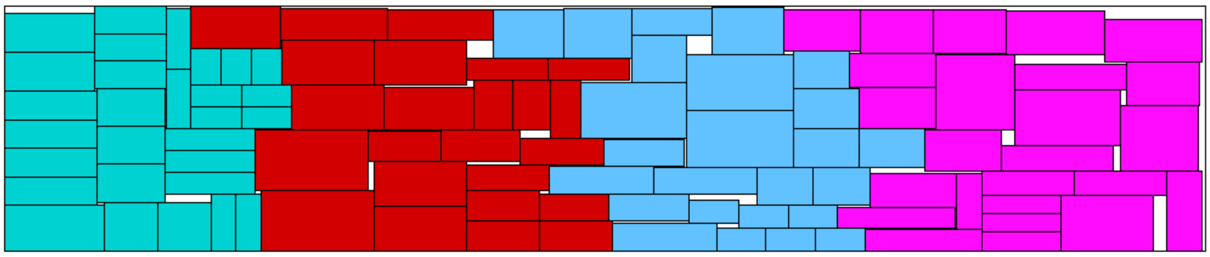


Figure 18: A Bin Packed with Four Orders (Ten Box Types Each).

5.2 Results for Three Dimensional Bin Packing

The experimental results discussed in this section were obtained by the bin packing algorithm in Section 2. However, it turned out that a slight refinement of the selection mechanism improved the results. Instead of selecting lexicographically on all 12 permutations of volume utilisation, upper bound on volume utilisation and complexity, we defined five different equivalence classes within each of the 12 permutations, and used the resulting 30 operators to select (if the population size allows it) and pre-select (if the selection constant allows it). The equivalence classes were defined as follows. For a bin with volume v, the volume in class i was defined to be

$$\text{volume in class } i = \begin{cases} \lfloor 25^{-3} \cdot 8^{-i} \cdot v \rfloor & \text{if } 1 \leq i \leq 4, \\ v & \text{if i} = 5. \end{cases}$$

The upper bound on the volume utilisation was treated the same. The complexity of a bin in class i for a bin with complexity c was defined to be

$$\text{complexity in class } i = \begin{cases} \lfloor 2^{-i} \cdot c \rfloor & \text{if } 1 \leq i \leq 4, \\ c & \text{if i} = 5. \end{cases}$$

#Box Types	2D		3D	
(per Order)	Vol. Ut	Time (s)	Vol. Ut	Time (s)
3	0.939	55	0.731	438
4	0.937	61	0.713	523
5	0.936	67	0.705	606
8	0.934	81	0.664	790
10	0.935	88	0.647	883

Figure 19: Multiple Destinations Bin Packing — Impact of #Box Types.

	2D		3D	
#Orders	Vol. Ut	Time (s)	Vol. Ut	Time (s)
1	0.954	58	0.802	399
2	0.948	54	0.776	457
4	0.939	55	0.731	437
8	0.927	54	0.678	436
16	0.919	53	0.610	390

Figure 20: Multiple Destinations Bin Packing — Impact of #Orders.

Individuals within the same class are then compared lexicographically. Note that class five consists of the lexicographic operators. The intuition behind these definitions is that small differences in one criteria should not make a substantial difference in the selection process, where the notion of small depends on the actual class.

Pallet Loading. We applied the algorithm to input with the same characteristics as Bischoff *et al.* [BJR95]. This input consisted of two different types of problems. The *Type I* problems consisted of small boxes; their dimensions are randomly chosen from the domain $([200, 400] \times [150, 350] \times [100, 300])$. The *Type II* problems consisted boxes of chosen from a wider range: their dimensions are randomly chosen from $([150, 450] \times [100, 400] \times [50, 350])$. The pallet size was kept constant at $(1000, 1200, 1500)$. Orientations of a box in which the vertical dimension exceeds one of the other dimensions by more than a factor of two were not allowed. After fixing the box types for a certain input we generated the actual boxes by picking a random type and adding a box of the chosen type to the input, while the total volume of the chosen boxes was smaller than that of the pallet. For each problem type we carried out experiments with inputs that contained k different box types, where k was one of 3, 5, 8, 10, 12, 15 or 20. For each value of k we executed the algorithm on hundred different random problems. All experiments were carried out with a population size (c_P) of 150, and a selection constant (c_s) of 2. Figure 21 shows the average volume utilisation, and average running time.

If one compares our algorithm with that of Bischoff *et al.*, the first thing to note is that ours is far more complicated, resulting in an algorithm that is slower. The average volume utilisations published by Bischoff *et al.*, obtained on Type I and Type II problems (and rounded up to two digits after the floating point) are also shown in Figure 21. In terms of volume utilisation, our algorithm performed better than the algorithm of Bischoff *et al.* on problems of Type I for $k = 3$ and $k = 5$; equally good for $k = 8$ and $k = 10$, and worse for $k = 12, 15$, and 20. However, on Type II problems our algorithm performed better than the algorithm of Bischoff *et al.* except for the case of $k = 20$, where both algorithms performed approximately equal.

#Box	Type I			Type II		
	This Paper		[BJR95]	This Paper		[BJR95]
Types	Vol. Ut.	Time (s)	Vol. Ut.	Vol. Ut.	Time (s)	Vol. Ut.
3	0.848	510	0.83	0.854	489	0.82
5	0.827	716	0.82	0.833	728	0.80
8	0.815	1071	0.82	0.815	1039	0.78
10	0.796	1374	0.80	0.798	1228	0.78
12	0.777	1680	0.80	0.788	1479	0.76
15	0.760	2060	0.80	0.764	1952	0.76
20	0.739	2944	0.78	0.744	2552	0.75

Figure 21: THE SINGLE BIN ALGORITHM APPLIED TO PALLET LOADING.

Multiple Destination Bin Packing. To generate inputs for multiple destination bin packing, we used a slight variation of the input generation process described above. We generated the box types at random from the domain $([600, 1200] \times [400, 1000] \times [200, 800])$. The truck size was kept constant at $(2000, 7000, 2000)$. After fixing the box types for a certain input we generated the actual boxes by picking a random (order, type) pair and adding a box of the chosen type to the chosen order, while the total volume of the chosen boxes was smaller than that of the truck. An example output can be found in Figure 22.

We carried out experiments with four different orders per input, and k distinct box types per order, with k in 3, 4, 5, 8, and 10. For each value of k, we executed the algorithm on hundred different random inputs, taking $c_P = 120$ and $c_s = 2$. The average volume utilisation and running time are shown in the right of Figure 19. To measure the performance of the algorithm if the number of orders increase, we also carried out experiments with k different orders, taking k in 1, 2, 4, 8, and 16. For each order we generated three box types. The average volume utilisation and running time observed are shown in the right of Figure 20.

If one compares the results of the algorithms proposed for two and three dimensional packing, it is clear that the two dimensional one suffers less from increasing the number of box types than the three dimensional one. This was to be expected, because it might not be possible to stack the larger variance of boxes whereas they can be fitted on a surface. It is also interesting to compare the performance of the two dimensional data structure with that of the three dimensional one. The three dimensional data structure seems to be approximately twelve times as slow as the two dimensional data structure, per box in the output. This factor is caused by the differences in the size of the two and three dimensional search space (contributing at least a factor of three due to the number of possible orientations of a box), and by the differences in complexity of the data structures for two and three dimensional packing.

To gain more insights into the actual dependence of our algorithm on the population

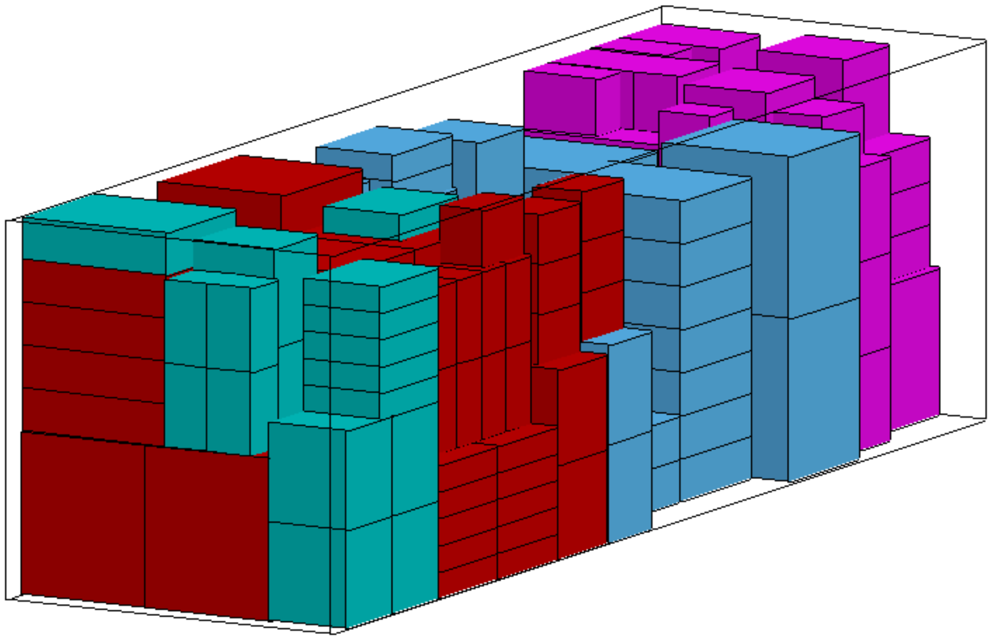


Figure 22: A PACKED BIN CONTAINING FOUR DIFFERENT ORDERS.

size c_P and the selection constant c_s, we carried out several additional experiments with the same set of hundred random inputs, each consisting of four orders, each order having three random chosen box types. The average volume utilisations and running times (in seconds) are shown in Figure 23. Figure 23a shows us that doubling the population size does indeed give us better results, however, the average volume utilisation increases with a small constant and the time required increases proportional to the population size. In Figure 23b, the product of c_s and c_P was kept constant at 720. This is interesting because the amount of memory used by the algorithm is linear in this product. The best results were obtained for the combination in which c_s was 2 and c_P was 360. Finally, Figure 23c shows the dependence of the algorithm on the selection constant c_s. The results indicate that c_s should be chosen as small as possible, namely $c_s = 2$. This might seem counter intuitive, since increasing c_s makes the search more thorough, but it can be explained as follows. The larger c_s, the less rounds of the algorithm are needed for a super individual to take over the population with its offspring. Sometimes, this causes the algorithm to converge towards a certain solution prematurely. Of course, this premature convergence is slowed down by choosing c_s smaller.

$c_s = 2$

c_P	Vol. Ut.	Time
2	0.622	7
12	0.683	40
30	0.695	106
60	0.718	224
120	0.731	438

(a)

$c_P \cdot c_s = 720$

$c_P \cdot c_s$	Vol. Ut.	Time
$30 \cdot 24$	0.680	360
$60 \cdot 12$	0.682	517
$120 \cdot 6$	0.698	743
$240 \cdot 3$	0.726	1096
$360 \cdot 2$	0.757	1642

(b)

$c_P = 120$

c_s	Vol. Ut.	Time
2	0.731	438
3	0.704	518
6	0.698	744
12	0.692	1063
24	0.694	1564

(c)

Figure 23: Dependence on Population Size and Selection Constant.

6 Conclusions and Further Research

We presented a packing algorithm that can be used for multiple-destination bin packing, and showed that our approach results in highly competitive volume utilisations both for three-dimensional pallet loading and for two- and three-dimensional multiple-destination bin packing. Furthermore, we showed that these results can be improved at the cost of higher running times (or faster and probably more expensive hardware).

The algorithm presented in this paper uses an upper bound on the bin utilisation that can be achieved with a packing as criterion to choose between different packings. However, when deriving this upper bound we ignored those spaces that emerge in the front that are too small for the smallest box we might want to place. So, the algorithm might be improved by taking this into account.

Another course of further research could be finding more efficient ways to manipulate and inspect a data structure that represents fronts. This would enable us to achieve better volume utilisations within the same time bounds. For example, an efficient plane sweep algorithm that calculates the differences in 3D-vertices and empty volumes in one pass for all possible box placements in a front would reduce the overhead for inspecting the front drastically. Another example might be slightly change the algorithm such that it projects on the bottom of the bin instead of on the back. This would eliminate the need of dummy candidates, but maybe complicate the calculation of our selection criteria.

Other interesting further research could be refining the pre-selection and selection process, for example with more sophisticated ranking schemes or some kind of stochastic selection. Stochastic selection produces good results for genetic algorithms, maybe it would do the same for our bin packing algorithm.

References

[Bea85] J. E. Beasly. An exact two-dimensional non-guillotine cutting tree search procedure. *Operations Research*, 33:49–64, 1985.

[Ben82] B. E. Bengtsson. Packing rectangular pieces—a heuristic approach. *The Computer Journal*, 25(3):353–357, 1982.

[BJR95] E. E. Bischoff, F. Janetz, and M. S. W. Ratcliff. Loading pallets with non-identical items. *European Journal of Operations Research*, 84:681–692, 1995.

[BKOS94] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. Computational geometry by example. Unpublished manuscript, 1994.

[CLR90] T. H. Cormen, E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, second edition, 1990.

[Col60] G. E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 2(12):655–657, December 1960.

[DF92] H. Dyckhoff and U. Finke. *Cutting and Packing in Production and Distribution.* Physica-Verlag, Berlin, 1992.

[FDF$^+$90] J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes, and R. L. Phillips. *Introduction to Computer Graphics.* Addison-Wesley Publishing Company, second edition, 1990.

[Geo92] J. A. George. A method for solving container packing for a single size of box. *Journal of the Operational Research Society*, 43(4):307–312, 1992.

[GMM90] H. Gehring, K. Menschner, and M. Meyer. A computer-based heuristic for packing pooled shipment containers. *European Journal of Operational Research*, 44:277–288, 1990.

[GR80] J. A. George and D. F. Robinson. A heuristic for packing boxes into a container. *Computers & Operations Research*, 7:147–156, 1980.

[HKE89] C. P. Han, K. Knott, and P. J. Egbelu. A heuristic approach to the three-dimensional cargo-loading problem. *International Journal of Production Research*, 27(5):757–774, 1989.

[HT90] R. W. Haessler and F. B. Talbot. Load planning for shipments of low density products. *European Journal of Operational Research*, 44:289–299, 1990.

[IK75] O. H. Ibarra and C. E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22(4):463–468, 1975.

[NTC94] B. K. A. Ngoi, M. L. Tay, and E. S. Chua. Applying spatial representation techniques to the container packing problem. *International Journal of Production Research*, 32(1):111–123, 1994.

[Por91] M. C. Portmann. An efficient algorithm for container loading. *Methods of Operations Research*, 64:563–572, 1991.

[Pug90] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[Wil64] J. W. J. Williams. Algorithm 232 (heapsort). *Communications of the ACM*, 7:347–348, 1964.