Bin packing problem with order constraints.

Petter Olsen Lundanes

Master of Science in Computer Science
Submission date: June 2014
Supervisor: Magnus Lie Hetland, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

NTNU

Norwegian University of Science and Technology

Bin packing with order constraints

TDT4900 - Master thesis

Petter Lundanes

June 2014

Advisor: Magnus Lie Hetland

Abstract

This thesis presents an algorithm to solve a variant of the bin packing problem with additional constraints on the order of items. The performance of this algorithm is tested, both for optimal solutions and approximations given by early termination, and is found to be limited for optimal solutions, but fairly efficient for decent approximations.

Denne oppgaven presenterer en algoritme for å løse en variant av bin packing problem med ekstra begrensninger på rekkefølgen av elementene. Ytelses blir testet, både for optimale løsninger og tilnærminger gitt av tidlig avsluttning, og blir funnet til å være begrenset for optimale løsninger, men forholdsvis god for tilnærminger.

Contents

List of Figures

1 Introduction

1.1 Overview

This paper presents and tests an algorithm to solve a variant of the bin packing problem with constraints on the order of items.
It starts with presenting some background knowledge required for the remainder of the paper and subsequently presents a more formal problem definition and motivation for this problem (section 1.3). Section 2 presents a new algorithm, based on branch and bound, to solve this problem. The performance and characteristics of this algorithm are tested and presented in section 3 and discussed in section 4. Possibilities for further work to improve this algorithm are mentioned in the discussion and summed up in section 6.
The appendix includes the implementation used for performance testing, more detailed results from the testing and various additions.

1.2 Background

This paper assumes that the reader has a general understanding of algorithms, including Big-O notation and the basics of complexity classes (this is used in the paper, but is not critical to the general understanding of the algorithm or discussion), and pseudo-code. For literature on this topic see Introduction to Algorithms [4] or similar.
Additionally it relies and builds heavily on the bin packing problem and the branch and bound approach for combinatorial algorithms, both of which are explained in brevity below.

1.2.1 Bin packing problem

The bin packing problem is a classical algorithmic problem where items with a given size are to be divided into a minimum number of uniformly sized bins. This is most commonly presented with constraints in only one dimension (for example weight or length), but 2-D (area), 3-D (volume) and more abstract k-D problems exist. Even the 1-dimensional problem is strongly NP-hard [8], but many decent approximation algorithms exist [6] [5] and several fairly efficient exact algorithms [12] [14].

1.2.2 Branch and bound

The branch and bound approach works by iterating through the search space (all valid, but not necessarily optimal, solutions), but discarding large sets of solutions based on upper and lower bounds of the optimal solution.
This works by incrementally building a plausible solution one step at a time. Starting with an empty node this then considers each valid expansion towards a solution, creating a new set of nodes, then this recursively continues with each of these nodes in the same way, building a tree. But at each node a lower bound and upper bound is compared. If the current lower bound (how this is computed varies greatly from algorithm to algorithm) is greater than or equal to the global upper bound (typically the best solution found so far, but any technique will work correctly) the node can be safely discarded without recursing on its children, which can potentially yield large performance gains. This can be done because any complete solution generated in this branch must be worse (or equal to) the best currently known solution and therefor cannot improve upon it.
For a more extensive discussion on this subject, see literature such as Algorithmics for Hard Problems [9] or similar.

1.3 Problem

1.3.1 Problem definition

Given a set of N *items*, $A = (A_1, A_2 \dots A_{N-1})$, where element k has a weight of $W_k \leq$ **BinSize**, and set of order constraints, $C = ((A_l \rightarrow A_m), \dots)$, with each constraint $(A_a \rightarrow A_b)$ specifying that item a must come before item b. We want to fit these items into an ordered list of *bins* with a maximum capacity of **BinSize**, an upper limit of the sum of all items in each bin, without violating the given order constraints. A feasible solution is any combination of all items placed in any number of bins that does not violate any order constraint or size constraint on the bins. An optimal solution is any feasible solution which uses the lowest possible number of bins[1].

1.3.2 NP-hardness

Without the order constraints, this problem is identical to the original bin packing problem (BPP). This makes reducing BPP to our problem trivial and can be done simply by having an empty set of order constraints. Since BPP is NP-hard [8] and trivially reduceable to our problem, our problem must be NP-hard.

1.3.3 Motivation

The reason for this study is that it appeared as the underlying problem for a rehearsal scheduling problem studied in the authors previous depth study [11]. This study found it to be too hard for the approach presented and concluded that a dedicated algorithm was the most

[1] A problem instance can, and often will, have multiple optimal solutions

plausible approach to achieve sufficent performance.

1.4 Related work

The bin packing problem, and many variants thereof, have been studied extensively for decades and several good algorithms exist for both exact optimal solutions [12] [14] and approximations [7] [13]. The common variants are with different spacial constraints, the classical problem is 1-dimensional, but 2- and 3-dimensional variants have practical applications. Even more abstract k-dimensional problems have been studied, but no previous work has been found for constraints on the order of items. This is plausible as the applications for bin packing generally is minimizing the containers needed for each bulk of items, where the order is irrelevant.

2 Algorithm

2.1 Overview

This section presents an algorithm that solves the problem. The algorithm is not based off anything more specific than the general basic branch and bound technique, but some aspects are inspired from the MTP [12] and BISON [14] algorithms which solve the traditional bin packing problem fairly efficiently. This section will incrementally build to the complete algorithm starting from a basic branch and bound approach, which works but is very slow. Following this we will make more complex additions and changes to vastly improve performance. Complete pseudo code can be found in section 2.7 and an implementation, written in C, for performance and correctness testing is included in appendix D.

2.2 Basic branch and bound

This branch and bound based algorithm builds possible solutions by adding one item for each node and keeps track of only available space in the current bin. When an item is added it is either put in the current bin and the available space reduced, or, if it doesn't fit in the current bin, a new bin is created and the current cost is increased. When the last item is added, the path down the tree represents a possible solution, if the currently best solution has a higher cost than this solution it is replaced by the new solution, otherwise nothing is done (thus discarding the new, inferior, solution).

2.2.1 Branching

This initial algorithm uses a conceptually very simple technique for branching; try every valid edge. This is improved upon in later sec-

tions, but is kept simple here because it still requires noticeable book-keeping (and running time) to keep track of which items are valid.
To keep track of valid items, we use an array with statuses for each item. Each status is then either UNAVAILABLE, AVAILABLE or USED. Initially all items are either unavailable or available, depending on whether they depend on any other item. Then, when an item is added, it is marked USED and each edge from it (an edge from a to b represents that b is dependant on a) is considered: if the connected item is UNAVAILABLE and has no edges going to it from any item that is not marked USED, the connected item is marked AVAILABLE. Or, less formally; every item dependent on the added item is checked on whether it depends on any items not already added, and if so, is marked available for use in subsequent branchings.

2.2.2 Cutting

When entering a new node we immediately make a decision whether to continue branching or stopping in this node (cutting a piece of the search tree). This is done by comparing a lower bound estimate of the remaining items to the best solution discovered so far. We compute the lower bound with a very simple technique, simply removing the integrality requirement of the items and calculating the total size of all remaining items and dividing by the bin size and adding the number of bins already used.

$$LowerBound = BinsUsed + \frac{\sum S_i - RemainderInBin}{BinSize} \tag{1}$$

This estimate has a worst case of $0.5 *$ OPT, which is not very good, but will usually perform very well and can be computed in constant time in each node (the summation can be stored and modified in each step, removing the need for iteration). If the algorithm is applied to problems where the items are large (relative to bin size) it could be beneficial to use more precise (and expensive) estimates.

2.2.3 Proof of correctness

It is fairly easy to show that this algorithm produces an optimal solution. Branch and bound in general is trivial, as it tries every solution not proved not to be an improvement (cutting). This moves the correctness issue to whether the branching and cutting is correct.
The branching keeps a set of branch choices which are items that are not dependent on other non-used items, ergo it contains all (and no other than) valid branch possiblities and since the algorithm explores every branch, every solution must thus be tested.
Branch and bound requires the lower bound estimate to be a strict lower bound for the cutting to be correct. This is trivial as the estimate is filling all bins completely and thus cannot be improved.

2.2.4 Pseudo code

Algorithm 1 Basic branch and bound

```
procedure BASIC BRANCH AND BOUND($graph$)
    $N \leftarrow$ length of $graph.items$
    $BestSolutionValue \leftarrow$ Infinity
    $BestSolution \leftarrow [-1] * N$
    $CurrentSolution \leftarrow [-1] * N$
    $S \leftarrow [\text{AVAILABLE}] * N$
    $TotalRemainder \leftarrow 0$
    for $i \leftarrow 0 \rightarrow N$ do
        $TotalRemainder \leftarrow TotalRemainder + graph.items[i].weight$
        for $j \leftarrow 0 \rightarrow N$ do
            if $graph.edges[i, j]$ then
                $S[j] \leftarrow \text{UNAVAILABLE}$
    RECURSE(0, 1, BinSize)

procedure RECURSE($step$, $cost$, $remainderWeight$, $totalRemainder$)
    // Cut, if possible
    $lowerBound = cost + \text{CEILING}((TotalRemainder - remainderWeight)/\textbf{BinSize})$
    if $BestSolutionValue \leq lowerBound$ then
        return
    // In leaf node, update best solution if an improvement is found
    if $step = N$ then
        if $cost < BestSolutionValue$ then
            $BestSolutionValue \leftarrow cost$
            $BestSolution \leftarrow CurrentSolution$
        return
```

```
25:     for $i \leftarrow 0 \rightarrow N$ do
26:         if $S[i]$ = AVAILABLE then
27:             $S[i]$ = USED
28:             $CurrentSolution[step] \leftarrow i$
29:             // Update status of nodes
30:             for $j \leftarrow 0 \rightarrow N$ do
31:                 if $graph.edges[i, j]$ and $S[j]$ = UNAVAILABLE then
32:                     $S[j]$ = AVAILABLE
33:                     for $k \leftarrow 0 \rightarrow N$ do
34:                         if $graph.edges[k, j]$ and $S[k] \neq$ USED then
35:                             $S[j]$ = UNAVAILABLE
36:             // Recurse
37:             $weight \leftarrow graph.items[i].weight$
38:             if $remainderWeight - weight \geq 0$ then
39:                 Recurse($step + 1$, $cost$, $remainderWeight - weight$, $totalRemainder - weight$)
40:             else
41:                 Recurse($step+1$, $cost+1$, BinSize$-weight$, $totalRemainder-weight$)
42:             // Remove added nodes
43:             for $j \leftarrow 0 \rightarrow N$ do
44:                 if $graph.edges[i, j]$ then
45:                     $S[j]$ = UNAVAILABLE
```

2.3 Edge system

Our first improvement is a new system to keep track of available items. As we can see (line 30 to 35 in algorithm 1) updating the available items has a worst case of $\Theta(|V|^2)$ and all problems will have an average of at least $\omega(|V|)$.
To improve this we make two changes. First, we introduce an array

with N integers, which keeps track of the number of constraints from non-USED items for each item. This changes the updating procedure to just going through every edge from the chosen node, and decrementing the counter of the receiving item. This also removes the need for index-based lookups for the receiving end of an edge, which leads us to the second change: Changing the neighbor matrix to an array of N variable sized lists.
The updating step is shown in algorithm 2. Some other changes are necessary as well (mainly the removal step and some initialization), but these are trivial based on the update step and not shown here (see section 2.7 for complete pseudo code).
This clearly has a worst case of $O(|V|)$, and will for many problems

Algorithm 2 Update step with the improved edge system

```
for $j \leftarrow 0 \rightarrow graph.edges[i].length$ do
    $idx \leftarrow graph.edges[i][j]$
    $dependencies[idx]$ −= 1
    if $dependencies[idx] = 0$ then
        $S[idx]$ = AVAILABLE
```

even be constant, and is a very significant performance gain.

2.4 Found fit

Our next change is a small, but effective, change to the branching strategy. It also translates very well to the next major change in the way we branch.
The change is based on a simple observation; if any item fit in the current bin, branching with a bigger item to a new bin cannot produce a better results than including it. For our algorithm this means if any

item small enough to fit in the current bin has been found, we can safely discard all items that do not fit.
Proof of correctness: Looking at an arbitrary node, we have a partitioning of the items into two sets; the used, U, and unused, F, items. U requires C_U bins and F requires C_F, the total cost of solution is then $C_U + C_F$. When branching we move an item from F to U'. If any item is put in the current bin, $C_{U'}$ does not change and the remaining F' is a strict subset from the original F. Since $C_{F'} \leq C_F$ (this is trivial as the worst case is the missing items in F' leaves unfilled space in the solution for F), this branch is clearly as good as, or better than, creating a new bin.

2.5 Sorting - Biggest fit branching

Since our currently best found solution is critical for cutting the search space, it is very important to quickly find a good approximation and always try the branches most likely to improve this first.
For the original bin packing problem, the *best fit decreasing* [7] and *first fit* [6] approximation algorithms are simple, efficient and give very good estimates, but our additional constraints make them unsuitable here. Instead, we make a derivation inspired by the two, which also is better suited for branching. The idea is to greedily try to fit the biggest item into the current bin until no items can fit, then create a new bin and start again. The worst case for this is bound (likely not a tight bound, but a proof for this is beyond the scope of this paper) by $2 * OPT$, but will (especially with item sizes small relative to the bin size) perform very well. The fact that the biggest items are tried first is also perfect for our lower bound calculation on the remaining items since it generally performs better with smaller items.
To try every possible solution (which is needed for the branch and bound method approach to be correct) we simply try the next biggest item when backtracking. This also fits very well with the observation

we made in section 2.4, that if any item fit, it is unnecessary to branch with items that don't fit. With our sorted list, this can be done very effciently, simply go through the list until an item that fits is found and start our main loop from there. If no item is found, simply start from the beginning.
To try the items in decreasing order, we obviously need a sorted list of the available items. Implementation-wise, we have two general choices: Either sort the set of available items in each node, or maintain a sorted list at all times. To decide our approach, let us first discuss which operations that will be used. Each iteration in the main loop (line 25 to 45 in algorithm 1) tries one item, this item is removed from the set before the recursive call (because it is now used) and added back again after the recursive call. The exact opposite (adding before the recursive call, and removing afterwards) is necessary for every item that becomes available, this will on average happen once per iteration (any item can obviously only be made available once per possible solution). This set of available items will usually be quite small, but as this will be done extremely frequently performance is paramount. Sorting can be done linearly (given a few very reasonable assumptions) [10], but is still quite expensive and will still require additional bookkeeping for adding and removing the nodes into the unsorted set. Maintaining a sorted list can be done in several ways, the method we use here uses a double linked list with naive linear insertion. Removing and adding the item being tried can be done in constant time (because the changes done to the list in between are all undone). Each item made available (on average 1, as discussed) can be inserted in linear time (with respect to the number of items in the list) and removed in constant time (with an addition discussed below). Thus maintaining this list will on average only require iterating through half the list in each iteration and is a logical choice even though the worst case performance is asymptotically worse than sorting in each step.

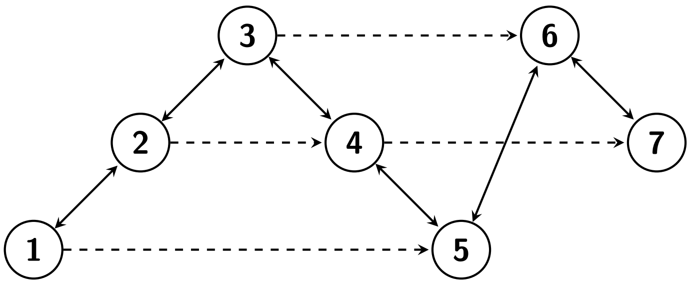

Fig. 1: Example of triply linked list. The normal arrows shows the standard doubly linked list and the dashed horizontal lines show our added third link.

2.5.1 Triply linked list

Above we claimed to be able to remove each item in constant time from the sorted list, but this is not entirely trivial as the naive approach requires iterating through the list to find the items. Since on average there will only be one item, this is very wasteful. To improve this we introduce a third link to our normal doubly linked list, linking to the next item on this 'level' (as items are always symmetrically[2] added and removed on different depths in the recursive calls, they are nicely grouped into what is essentially a stack). With the trivial addition of another "starting" pointer for the first item added on each level, this allows for very efficient removal of all the items by simply traversing the list given by this third link. Figure 1 gives an example of how such a list could look.

[2] Note that items can be (and usually are) added on one level and removed on another lower level, but as the lower removal is reversed before returning to the upper level and it is invisible for all practical purposes.

2.6 Removing intra-bin redundancies

Our current branching strategy creates a significant amount of branches that are for our purposes identical, and thus redundant. These can be grouped into two:
The first being internal order of items inside a bin. Say the set of items contains A, B and other items, we could then branch by trying A first, then adding B in the same bin and continuing the same procedure until the set is empty and we have a possible solution. When we then backtrack back until A is removed, we try the next item which could be B. Continuing we add A into the same bin and continue until we have found the same solution (except the order of A and B) as before. Because the order of items inside the bin does not affect the solution at all, this is very wasteful.
The second redundancy is the order of bins. Imagine two bins filled with distinct sets of items. If there are no constraints between these bins (or the constraints have been fulfilled), the order of these two bins does not affect the cost or validity of the solutions in any way, but all permutations can be computed in a situation similar to the previous example.
The problem is greatly alleviated by our cutting. Since the extra branching cannot compute a better solution than we already have, a good lower bound estimate (which we usually have) will help discard these fairly quickly, but there is still a very significant waste. The latter issue is hard to remove in an efficient manner, but we can improve upon the first.
Let us start by ignoring our order constraints for simplicity. Since we branch with items in decreasing order we can simply force the order of the items in each bin to be identical to that of the sorted list[3], giving each distinct bin only one allowed order and thus completely removing

[3] Note that this is an identical order, not just decreasing, meaning that two items of the same size will have an decided order.

the redundancies. We can implement this efficiently by simply passing the next possible (ignoring size) item in the recursive call, and this gives a starting point for iterating for the first (if any) fitting item. Thus, if any bigger item could be put in the current bin, we know that this already has been tried. If no item can fit, it is simply ignored when starting at the beginning of the list, which is exactly what we want, as a new bin will be created.
There is however a problem with this elegant improvement, it can incorrectly discard items that are added to the list by fulfilling its constraint in the same bin. Say A must be before B, but B is bigger than A, when adding A to a bin (and thus B to the sorted list) the next item to be tried must be smaller than A since all bigger seemingly have been tried, but as B was not available earlier it has not been tried and is thus a valid (and necessary) branching choice that is incorrectly ignored. We work around this by instead of just passing the next smaller item, we pass the either the next smaller item, or (if any exist) the biggest newly added item that is bigger than the next smaller item. This is effectively just relaxing the new internal ordering constraint when the issue occurs. This relaxation will allow some redundancies to remain because items between the newly added item and the next biggest item will now be considered again, but this is not a big issue. A technique with a boolean array for each bin indicating which items have been tried was tested, but even with tricks for no allocation in the recursive calls, the performance loss in each node far outweighed the gain by removing the last redundant branches.

2.7 Complete pseudo code

The following is the complete pseudo code of the algorithm. The auxiliary methods used can found in appendix A.

2.7.1 Pseudo code

Algorithm 3 Complete algorithm

```
procedure PACK BINS($graph$)
    $N \leftarrow$ length of $graph.items$
    $BestSolutionValue \leftarrow$ Infinity
    $BestSolution \leftarrow [-1] * N$
    $CurrentSolution \leftarrow [-1] * N$
    $TotalRemainder \leftarrow 0$
    for $i \leftarrow 0 \rightarrow N$ do
        $TotalRemainder \leftarrow TotalRemainder + graph.items[i]$.Weight
    $GlobalLowerBound \leftarrow$ CEILING($TotalRemainder$/BinSize)
    $ListHead \leftarrow$ New list item
    $ListHead.weight \leftarrow$ Infinity
    for $i \leftarrow 0 \rightarrow N$ do
        for $j \leftarrow 0 \rightarrow graph.edges[i].length$ do
            $graph.edgeCounters[graph.edges[i][j]]$ += 1
    for $i \leftarrow 0 \rightarrow N$ do
        if $graph.edgeCounters[i] = 0$ then
            INSERTSORTED($ListHead$, $graph.items$[i].$Weight$, $TotalRemainder$, i)
    RECURSE(0, 1, BinSize, $nil$)
```

```
procedure RECURSE($step$, $cost$, $remainderWeight$, $totalRemainder$, $nextItem$)
    // Cut, if possible
    $lowerBound = cost + \text{CEILING}((totalRemainder - remainderWeight)/\textbf{BinSize})$
    if $BestSolutionValue \leq lowerBound$ then
        return
    // In leaf node, update best solution if an improvement is found
    if $step = N$ then
        if $cost < BestSolutionValue$ then
            $BestSolutionValue \leftarrow cost$
            $BestSolution \leftarrow CurrentSolution$
        return
    // Find the first item to try. See sections 2.5 and 2.6 for rationale.
    $iterator \leftarrow nextItem$
    while $iterator \neq nil$ do
        if $iterator.weight < remainderWeight$ then
            break
        $iterator \leftarrow iterator.next$
    if $iterator = nil$ then
        $iterator \leftarrow ListHead.next$
    // Main loop
    while $iterator \neq nil$ do
        $i \leftarrow iterator.itemIndex$
        DETACHITEM(iterator)
        $CurrentSolution[step] \leftarrow i$
        // Add newly available items to list
        $newItemsHead, newItemsTail, newItemsBiggest \leftarrow nil$
        for $j \leftarrow 0 \rightarrow graph.edges[i].length$ do
            $idx \leftarrow graph.edges[i][j]$
            $dependencies[idx] \mathrel{-}= 1$
```

```
48:             if $dependencies[idx] = 0$ then
49:                 $newItem \leftarrow$ InsertSorted($ListHead$, $graph.items[idx].weight$, $idx$)
50:                 if $newItemsHead = nil$ then
51:                     $newItemsHead$, $newItemsTail$, $newItemsBiggest \leftarrow newItem$
52:                 else
53:                     // Set the third link in our linked list, see section 2.5.1
54:                     $newItemsTail.NextAtLevel \leftarrow newItem$
55:                     $newItemsTail \leftarrow newItem$
56:                     if $newItemsBiggest.weight < newItem.weight$ then
57:                         $newItemsBiggest \leftarrow newItem$
58:         // Recurse
59:         $next \leftarrow iterator.next$
60:         if $newItemsBiggest = nil$ or $next = nil$ or $next.weigth > newItemsBiggest$ then
61:             $nextItem \leftarrow next$
62:         else
63:             $nextItem \leftarrow newItemsBiggest$
64:         $weigth \leftarrow iterator.weight$
65:         $totRemainder \leftarrow totalRemainder - weight$
66:         if $remainderWeight \geq weight$ then
67:             Recurse($step$+1, $cost$, $remainderWeight - weight$, $totRemainder$, $nextItem$)
68:         else
69:             Recurse($step$+1, $cost$+1, BinSize$-weight$, $totRemainder$, $nextItem$)
70:         // Increment edge counters
71:         for $j \leftarrow 0 \rightarrow graph.edges[i].length$ do
72:             $idx \leftarrow graph.edges[i][j]$
73:             $dependencies[idx]$ += 1
```

```
        // Remove added nodes from list
        while $newItemsHead \neq nil$ do
            $temp \leftarrow newItemsHead.NextAtLevel$
            RemoveItem(newItemsHead)
            $newItemsHead \leftarrow temp$
        // Make tried item available again
        AttachItem($iterator$)
        $iterator \leftarrow iterator.next$
```

2.8 Early termination for approximate solution

As increasing problem sizes rapidly become infeasible to solve for an optimal solution with NP-hard problems, approximations become important for practical usage.
The algorithm we just presented can trivially be modified to be an approximation algorithm, by its nature it always[4] has a feasible solution that is best currently found. The algorithm can thus terminate early (before it has completed the search) and this solution will be an approximation that is guaranteed to be feasible, but not optimal[5].
This solution can naturally only improve the longer the algorithm runs, which raises the question of when to terminate. The natural options are either when a given time has elapsed, when a solution is within a certain error margin, or a combination of both. What is preferred will vary with problem size and the resources available to the user. An error margin can easily be found by computing a lower bound, by the same technique we use in every node, for the entire problem and comparing this to our solution. However, there are issues with this approach. The rather simplistic lower bound will never improve,

[4] Technically only after the first path down the search tree has been completed, but this happens very quickly.

[5] The solution can be optimal, but is not guaranteed to be.

while our solution will converge towards optimality. This means that the longer the algorithm runs (to achieve a more precise solution) the less accurate our estimate of distance to the optimal solution will be.

3 Results

3.1 Benchmarking setup

To measure how well this algorithm perform, we run a suite of problem instances and record relevant data. For the optimal solution we record how long the algorithm takes to finish. To evaluate its usefullness as an approximation algorithm we store the global lower bound, how long it takes to give a solution that is within 5% and 1% error margins, and the best solution it finds before terminating. Each problem is run for a maximum of 2 minutes. The information presented here is a subset of all the measurements, for complete tables see Appendix C.2. Time is measured with the clock function in C [1] with a resolution of 10ms. Specifications for the test computer can be found in Appendix B.
Note that the error margin limits are calculated with CEIL($LowerBound * Margin$), this makes a significant difference for smaller problems, but becomes more or less negligible rather quickly.
The problem instances we use can be grouped into two families. The first is generated the same way as was done in the study. A problem of size N has N arrays of N items (giving a total of N^2 items, thus the complete size is quadratic to the size presented), where each item in the arrays has to come after the previous item in the array. Sizes vary from 6.25% to 25% of the bin size. The only parameter we control here is N. For details about these problems and the practical problem it represents, see the depth study [11] that preceded this paper. These problems are useful because they both represent a practical problem

and gives us the ability to compare the performance of our algorithm to the solution given in the depth study.
The second family of problems is randomly generated with a number of given parameters. It takes a number of items, which is self-explanatory. A percentage of constraint "density", given k, every item will be dependent on $k\%$ of the previous items (any item i will have required to come after $\frac{k}{100} * i$ items chosen randomly from all items with an index lower than i). Additionally, it takes upper and lower limits for item sizes, given in percent of the bin size, all item sizes are uniformly distributed within this range. These problems are essentially a worst case, as they will have a high number of possible branches relative to the number of constraints, but the flexibility of generation is useful for testing specific effects.

3.2 Comparison with depth study implementation

Here we compare our algorithm to the approach presented in the depth study [11] that inspired this paper.
The approach taken there is forming a binary integer programming (BIP) model and solving it with a third party solver. The solver used here is Gurobi [2], which was the vastly superior solver tested in the study. For details about the BIP approach, see the depth study paper [11].
The problem instances used here are from the depth study family, and are identical between the two solvers. Timing the BIP solution was done with the linux time-command [3], which introduces a slight imprecision, but as we see this difference is negligible relative to the performance difference.
Figure 2 shows the two compared visually, complete recorded data is in appendix C.2.

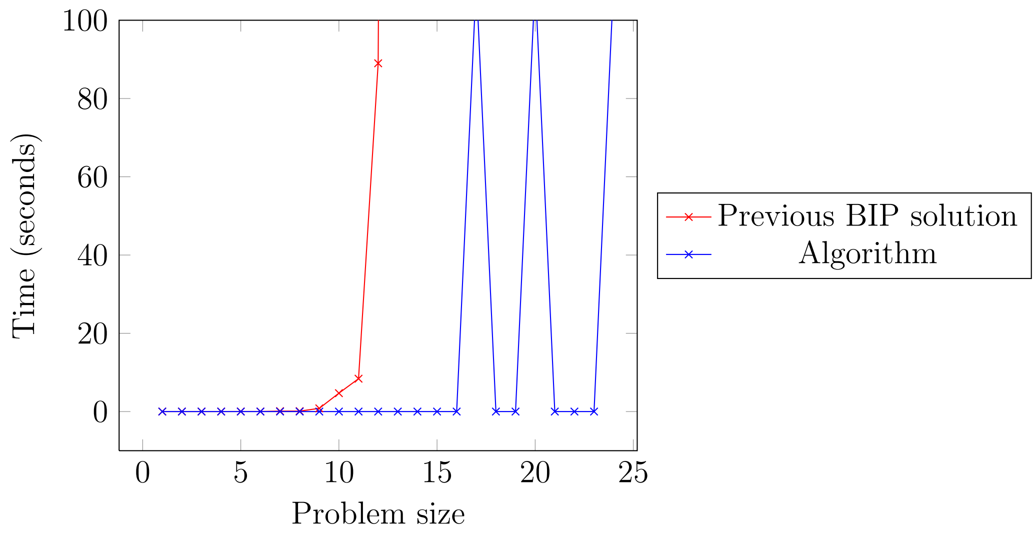

Fig. 2: Comparison with the BIP approach.

3.3 Time for 1% error margin

Using the same problem family, but with much larger instances, we measure how long the algorithm takes to come an solution that is at most 1% more than the global lower bound, which thus is within 1%, but likely significantly closer, to the optimal solution. The results are presented graphically in figure 3 and complete data is given in appendix C.1

Note again the number of items is quadratic to the size presented (thus the largest problem instance tested, of size 180, has 32400 items).

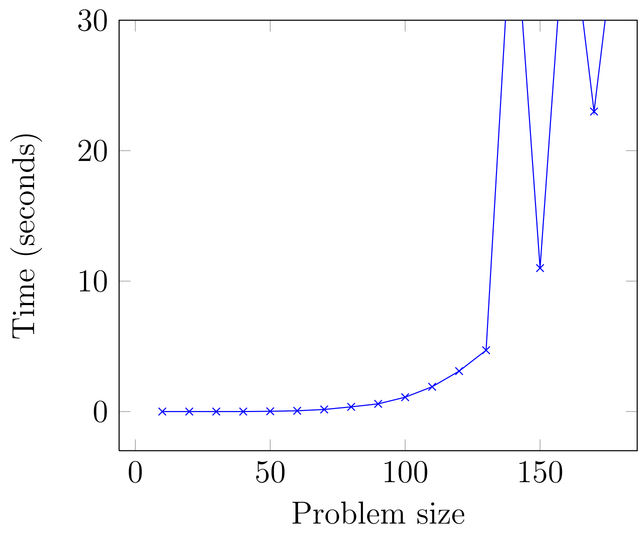

Fig. 3: Time used for approximation within an 1% error margin.

3.4 Progression of best solution over time

Here we use a fairly large problem instance (size 170, with 28900 items) of the depth study problem family and see how the algorithm's best currently found solution improves over time. The results are presented in figure 4.

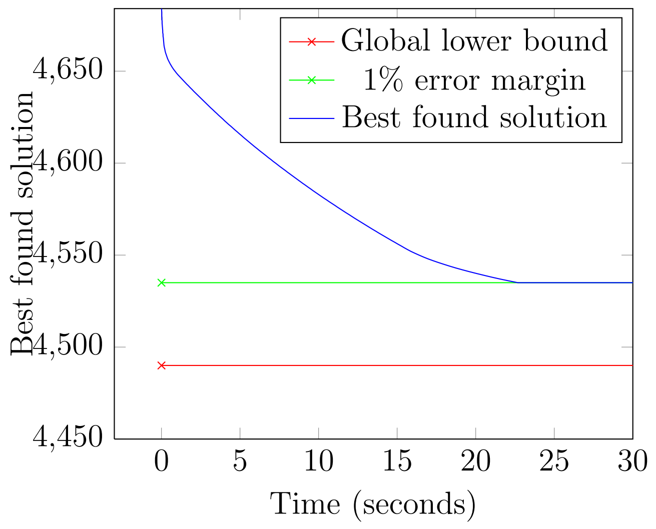

Fig. 4: Improvement of best currently found solution over time.

3.5 Effects of constraints on estimate accuracy

To see how the number of order constraints affect our algorithm we run similar problems, with different amounts of constraints and compare the best found solution with lower bound. The problems used are from the randomized family, generated with 1000 items with sizes between 10% and 50% of the bin size.
The difference is given as the percentage difference between the lower bound and best solution and is shown in figure 5.

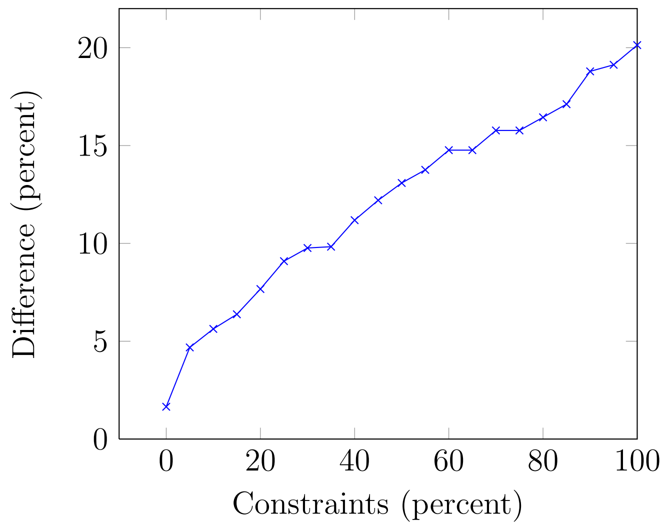

Fig. 5: Effects of constraints on guaranteed approximation accuracy.

3.6 Effects of item size on estimate accuracy

We measure the effects of item sizes with running randomized problems with varying sizes, and all other parameters constant. Note that this does not really give an identically sized problem because item size greatly affects number of bins, but is likely more representative than decreasing number of items to maintain a constant number of bins.
We run problems with 1000 items, 10% order constraints and increasing item sizes. For a run with k average size, sizes are uniformly distributed between $k - 20$ and $k + 20$ (in percent of bin size).
Results can be seen in figure 6.

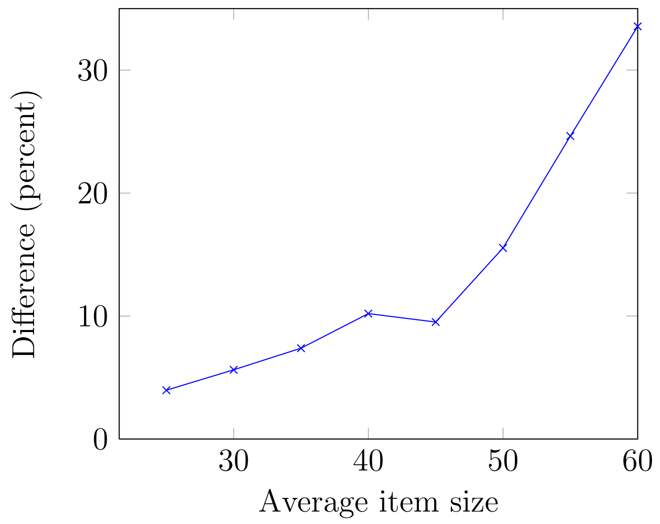

Fig. 6: Effects of item sizes on guaranteed approximation accuracy.

4 Discussion

4.1 Optimal solution performance

It is difficult to evaluate the algorithms performance for finding a guaranteed optimal solution because there is very little to compare it to. The only other solution for similar problems are from the depth study which was made with different goals, but can (given enough time) solve all problems this algorithm can. As section 3.2 shows, our algorithm very clearly outperforms the previous solution. The latter begins to require infeasible amounts of time at just over 100 items, while our algorithm performs exceptionally with all instances to over 200 items and for some instances over with over 500 items. While this is a very significant improvement, they are still fairly small instances, and other problem families can run into performance problems at even smaller

instances.
But it is important to remember that the problem is NP-hard, and thus no polynomial time algorithm exist for conventional computers, unless $P = NP$ [9]. And with non-polynomial running times, it is very hard to avoid performance problems even at rather small problem instances.
A curious thing to note is that almost every test run either completed very fast (less than 10 milliseconds) or did not complete before aborted. The tests presented here were aborted after 2 minutes, but other tests were run where the problems were just slightly larger than problems that ran fine, and these ran for several hours without completing (in fact, no better solution was found in hours, than those found in the first few seconds). Additionally, of the tests that did complete, essentially all of them found an solution with the same cost as the lower bound, and thus completing immediately[6].
This suggests that it is common, but not ubiquitous, that the algorithm actually finds the optimal solution very quickly, but unless this is equal to the lower bound, it will spend impractical amounts of time exploring the search space to guarantee that this solution is indeed optimal. It is, however, very hard to verify this hypothesis without a more efficient algorithm to find the optimal solution so these can be compared.

[6] If the best found solution is equal to the global lower bound, all local lower bounds must be either equal or higher, thus causing the cutting step to stop exploring.

4.2 Approximation performance

Again, this is difficult to evaluate without other solutions to compare it to, and usability depends heavily on what error margins are acceptable. But as we saw in sections 3.3 and 3.4, we are able to find solutions guaranteed to be within a few percent of the optimal solution in just a few seconds for problem instances with well over 10000 items and even the very first feasible solution found, which is found very quickly, is often within 5-10% of the optimal solution.
Additionally, our error margins are calculated based on our lower bound for an optimal solution, not the optimal solution. Mathematically speaking, *Lower bound* $\leq$ *Optimal* $\leq$ *Approximation*, thus some portion of the difference between our solution and the lower bound is the difference between the lower bound and the optimal solution. This means that our approximations are likely closer to the optimal solution than we can guaraantee, but how significant (and prevalent) this difference is, is very hard to measure without having the optimal solution, which is computationally infeasible to find with this algorithm[7].
Despite the limitations in quantifying which factor contributes more to the error, we can make some observations. Firstly, the lower bound algorithm, while good, is extremely simple and ignores important constraints, and is never improved upon during subsequent execution. While our estimate is often good from the beginning, and constantly improves during execution and in fact converges towards the optimal solution. Considering this, it seems unlikely that the lower bound is

[7] One corner case that can be (and was) measured, is problems from the randomized family, with 100% constraints. These will only have 1 feasible solution, which is found quickly and must be optimal. These suggest (see figure 6) that much of the difference is indeed from the lower bound, not our estimate. But as this is essentially a best case for our algorithm, and a worst case for the lower bound algorithm, it cannot be used to draw any general conclusions.

closer to the optimal solution than our estimate.
Additionally, if we consider figure 4 (which is follows a near universal pattern for that problem family), we see that the estimate converges towards a solution at, or slightly below, the 1% line. This strongly supports our previous observation, but is still insufficient to guarantee anything, as the algorithm cannot guarantee (despite it being likely) that it will converge uniformly.

4.3 Effects of problem characteristics

In sections 3.5 and 3.6 we explore the effects of item sizes and order constraint density on the guaranteed accuracy of the approximation. Note that this guaranteed accuracy is just the difference between the approximation and the lower bound, and thus the difference with respect to the optimal solution must be equal or, more likely, less.
Both item sizes and constraint density affect this accuracy very similarly, with a rather dramatic decrease in accuracy. If we just consider our algorithm, these are quite unexpected results. Intuitively, increased constraint density should improve our estimate, as it greatly limits the search space. And item sizes should not make much difference because even though it in effect slightly increases the problem size, it will also (on average) increase empty space in bins which means our lower bound improves faster and thus more of search space can be ignored. However, if we consider the lower bound, which is used as a reference for the accuracy, it makes more sense. The lower bound is computed by dropping the main constraints of the problem, integrality and order, which makes the problem very simple. This corresponds very well to our measurements, less order constraints naturally leads to less difference when these are dropped and smaller item sizes makes

the integrality constraint less important[8]. This also strongly supports the hypothesis above, suggesting that much of the error margin is due to an inaccurate lower bound, not an inaccurate approximation.

4.4 Improving the lower bound

As we have seen above, improving lower bounds could yield significant gains, both increasing how much search space we can discard during exploration and give tighter guarantees for how far our estimate is from the optimal solution. But it can be useful to view these seperatly, our global lower bound (which we use to guarateen an error margin for our best solution) is run only once, while local lower bounds (which are used to cut the search space) are run extremely frequently. The local lower bound is thus so performance sensitive that just iterating through that remaining (non-used) items is likely to be too expensive. This imposes a strict limitation on what algorithms could be used and it is unlikely that we could significantly improve upon the technique currently used. But for our global lower bound, we are much less restricted. In fact, the time used by virtually any polynomial time algorithm will be negligible compared to the rest of our algorithm.
Our current lower bound is found by relaxing the general problem by removing the integrality constraint and the order constraints, and then solving it exactly, which happens to be very easy. If we try the same approach, but relaxing our problem less, we come to two options, removing just one of these constraints instead of both. If we only remove the integrality constraint, we will get the exact same cost, but it will be slightly more expensive to compute[9]. This is obviously useless.

[8] Smaller item sizes will (on average) give less wasted space in bins, which is closer to the non-integrality case, where all (except the last) bins must be full.

[9] It will be more expensive to compute as the constraints have to be considered, but the cost cannot be higher because the lack of an integrality constraint will still

If we only remove order constraints, we are left with the original bin packing problem, and while this is still a very hard problem, it has also been extensively studied. For some problem instances it could even be feasible to solve this exactly with a dedicated algorithm such as MTP [12] or Bison [14], but more likely, we would still want a lower bound algorithm such as those proposed by Martello and Toth [13]. This would improve (or equal, in smaller problem instances) our global lower bound[10] for *all* problem instances, but the difference will naturally vary greatly depending on problem characteristics. It could be a major improvement for problems with large item sizes, but will likely be less useful when order constraints is the cause of error.

5 Conclusion

We have created a working algorithm based on the branch and bound approach that can be used either to find an optimal solution or to find a feasible approximation by terminating early. Its performance is, in general, lacking for guaranteed optimal solutions, but could be sufficient for certain types of problems and smaller instances. The algorithm performs significantly better when optimal solutions are not necessary and can produce good approximations quickly for fairly large problem sizes, despite limitations with the global lower bound used to calculate error margins, and could be useful for practical applications.

cause all (except possibly the last) bins to be filled completly.

[10] It could also help local lower bounds (in constant time) since we could use the highest lower bound of the global and currently computed local as a local lower bound, but this would likely be a rather insignificant improvement.

6 Further Work

As discussed in section 4.4, improving our global lower bound could be very beneficial when used as an approximation algorithm. Developing new algorithms dedicated to this problem could be challenging, but even using existing lower bound algorithms for the original bin packing problem could make a significant difference for many problem instances.
Additionally, some preprocessing could be used. For example it would not be very hard to remove redundant order constraints, given constraints $A \rightarrow C$, $A \rightarrow B$, and $B \rightarrow C$, the first could be ignored because of the transitive property of the dependencies. For certain problem types it could also be possible to handle certain special cases, for example if the items of a problem could be grouped into sets A and B, where all items in A must come before every item in B, it would be possible and extremely beneficial to treat this as two almost[11] independent problems.
It would also be interesting to compare the approximations found with this algorithm to optimal solutions. These optimal solutions are currently infeasible compute, but this can change if more efficient exact algorithms are developed.

[11] The remaining used space in the last bin of the first subproblem would affect the second subproblem. This must be taken into account, but does not make the problem significantly harder.

7 References

[1] Gnu libc time.h. http://www.gnu.org/software/libc/manual/html_node/CPU-Time.html [Online; accessed 26.05.2014].

[2] Gurobi home page. http://www.gurobi.com/ [Online; accessed 13.11.2013].

[3] 'time' man page. http://manpages.ubuntu.com/manpages/precise/man1/time.1.html [Online; accessed 13.11.2013], 2012.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, 2009.

[5] W. Fernandez de la Vega and G. S. Lueker. Bin packing can be solved within $1 + \epsilon$ in linear time. *Combinatorica*, 1(4):349–355, 12 1981.

[6] György Dósa. *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, chapter 1, pages 1–11. Springer, 4 2007.

[7] D. S. Johnsen et al. Worst-case performance bounds for simple one-dimensional packing algorithms. http://math.ucsd.edu/~ronspubs/74_04_one_dimensional_packing.pdf [Online; accessed 31.10.2013], 1974.

[8] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1 1979.

[9] Juraj Hromkovic. *Algorithmics for Hard Problems.* Springer, 2004.

[10] Donald E. Knuth. *The Art of Computer Programming: Volume 3 Sorting and search 2. Edition.* Addison-Wesley, 1998.

[11] Petter Lundanes. Theater rehearsal scheduling, 2013.

[12] S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations*. J. Wiley & Sons, 12 1990.

[13] S. Martello and P. Toth. Lower bounds and reduction procedures for the bin packing problem. *Discrete Applied Mathematics*, 28:59–80, 1990.

[14] A. Scholl and C. Jurgens R. Klien. Bison: A fast hybrid procedure for exactly solving the one-dimensional bin packing problem. *Computers & Operations Research*, 24(7):627–645, 1997.

A Pseudo code for auxiliary methods

Algorithm 4 InsertSorted

```
procedure InsertSorted($listHead$, $item$)
    // Find insertion point
    $iterator \leftarrow listHead$
    while $iterator.next \neq nil$ and $iterator.next.weight > item.weight$ do
        $iterator \leftarrow iterator.next$
    // Insert
    $last \leftarrow iterator.next$
    $iterator.next \leftarrow item$
    $item.previous \leftarrow iterator$
    $item.next \leftarrow last$
    $last.previous \leftarrow item$
```

Algorithm 5 DetachItem

```
procedure DetachItem($item$)
    $before \leftarrow item.previous$
    $after \leftarrow item.next$
    if $before \neq nil$ then
        $before.next \leftarrow after$
    if $after \neq nil$ then
        $after.previous \leftarrow before$
```

Algorithm 6 AttachItem

```
procedure AttachItem($item$)
    $before \leftarrow item.previous$
    $after \leftarrow item.next$
    if $before \neq nil$ then
        $before.next \leftarrow item$
    if $after \neq nil$ then
        $after.previous \leftarrow item$
```

B Test computer specifications

- Dell Vostro 1220. 2010
- Intel Core 2 Duo P8700 2.53GHz (2-core)
- 8GB DDR2 RAM
- Ubuntu 12.04 LTS 64bit (Linux kernel 3.2)
- GCC 4.6.3
- Gurobi Optimizer 5.6

C Complete measurements

These tables show the complete measurements for the tests where graphs presented only give a subset of information gathered.
The columns show the size of problem (number of items is quadratic to this size), the global lower bound (LB), the first solution found within the 5% error margin (5% v) and how long it took (5% t) and similar for the 1% error margin. Addtionally, it shows the best solution found before timeout (Best) and how long it took to find it (TB), the optimal solution if it was found (Opt) and when the algorithm completed (Done). The comparison with the depth study also includes the time gurobi used to complete (Gurobi).
Times are given in milliseconds, except gurobi timing which is given in seconds. -1 indicates an not available (timeout, or not found because of timeout).
Note that first solution within the 5% error margin is often the first solution found and can be significantly better than 5% off.

C.1 Time for 1% error margin

Size	LB	5% v	5% t	1% v	1% t	Best	Opt	TB	Done
10	14	15	0	15	0	14	14	0	0
20	61	64	0	62	0	62	-1	0	-1
30	136	142	0	138	0	137	-1	0	-1
40	244	255	0	247	0	246	-1	10	-1
50	386	403	0	390	20	390	-1	20	-1
60	552	576	0	558	60	557	-1	70	-1
70	766	799	0	774	160	774	-1	160	-1
80	985	1028	0	995	360	995	-1	360	-1
90	1258	1312	0	1271	590	1271	-1	590	-1
100	1547	1614	0	1563	1140	1562	-1	1210	-1
110	1874	1956	0	1893	1900	1893	-1	1900	-1
120	2239	2336	0	2262	3110	2261	-1	3250	-1
130	2617	2730	0	2644	4720	2643	-1	4910	-1
140	3063	3195	0	-1	-1	3096	-1	6740	-1
150	3488	3638	0	3523	11230	3523	-1	11230	-1
160	3985	4158	0	-1	-1	4027	-1	15740	-1
170	4490	4684	10	4535	22740	4535	-1	22740	-1
180	5035	5254	0	-1	-1	5087	-1	30400	-1

C.2 Depth Study comparison

Size	LB	5% v	5% t	1% v	1% t	Best	Opt	TB	Done	Gurobi
1	1	1	0	1	0	1	1	0	0	0.01
4	2	2	0	2	0	2	2	0	0	0.01
5	4	4	0	4	0	4	4	0	0	0.02
6	6	6	0	6	0	6	6	0	0	0.02
7	8	8	0	8	0	8	8	0	0	0.11
8	10	10	0	10	0	10	10	0	0	0.11
9	12	12	0	12	0	12	12	0	0	0.82
10	14	15	0	15	0	14	14	0	0	4.7
11	17	18	0	18	0	17	17	0	0	8.4
12	21	22	0	22	0	21	21	0	0	89
13	26	27	0	27	0	26	26	0	0	-1
14	31	32	0	32	0	31	31	0	0	-1
15	34	36	0	35	0	34	34	0	0	
16	38	40	0	39	0	38	38	0	0	
17	42	44	0	43	0	43	-1	0	-1	
18	48	50	0	49	0	48	48	0	0	
19	54	56	0	55	0	54	54	0	0	
20	61	64	0	62	0	62	-1	0	-1	
21	69	72	0	70	0	69	69	0	0	
22	74	77	0	75	0	74	74	0	0	
23	80	83	0	81	0	80	80	0	0	
24	86	90	0	87	0	87	-1	0	-1	
25	93	97	0	94	0	94	-1	0	-1	
30	136	142	0	138	0	137	-1	0	-1	
35	192	200	0	194	0	194	-1	0	-1	
40	244	255	0	247	0	246	-1	10	-1	
45	308	322	0	312	10	311	-1	20	-1	
50	386	403	0	390	20	390	-1	20	-1	

D Implementation

Complete sample implementation written in C. The exact implementation used for benchmarking.
Compiled with 'gcc -O2 -std=c99 -Wall -Wextra -g main.c -o solver'.

```
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
#include<string.h>
#include<limits.h>
#include<time.h>

#define UNUSED(x) (void)x;
#define PRINT_DEBUG 0

#define BENCHING 1

// 0 off, 1 cost only, 2 cost and combination
#define PRINT_COMBINATION 1

const int MaxWeight = 8*60;

typedef struct Node
{
        char* Label;
        int Weight;
} Node;

typedef struct ListItem ListItem;

struct ListItem {
        ListItem* Previous;
        ListItem* Next;
        ListItem* NextAtDepth;
        int Weight;
        int Index;
        int Depth;
};
ListItem* InsertAfterp(ListItem* item, ListItem* newItem)
{

        ListItem* last = item->Next;
        newItem->Previous = item;
        newItem->Next = last;
        item->Next = newItem;
        if (last != NULL)
                last->Previous = newItem;
        return newItem;
}
ListItem* InsertAfter(ListItem* item, int weight, int index
    , int depth)
{
        ListItem* newItem = malloc(sizeof(ListItem));
        newItem->Index = index;
        newItem->Weight = weight;
        newItem->Depth = depth;
        newItem->NextAtDepth = NULL;
        return InsertAfterp(item, newItem);
}

ListItem* InsertSorted(ListItem* start, int weight, int
    index, int depth)
{
        ListItem* it = start;
        while (it->Next != NULL && it->Next->Weight >
            weight)
        {
                it = it->Next;
        }
        return InsertAfter(it, weight, index, depth);
}
ListItem* InsertSortedp(ListItem* start, ListItem* item)
{
        ListItem* it = start;
        int weight = item->Weight;
        while (it->Next != NULL && it->Next->Weight >
            weight)
        {
                it = it->Next;
        }
        return InsertAfterp(it, item);
}

void RemoveItem(ListItem* item)
{
        ListItem* before = item->Previous;
        ListItem* after = item->Next;
        if (before != NULL)
                before->Next = after;
        if (after != NULL)
                after->Previous = before;
        item->Next = NULL;
        item->Previous = NULL;
        free(item);
}
void DetachItem(ListItem* item)
{
        ListItem* before = item->Previous;
        ListItem* after = item->Next;
        if (before != NULL)
                before->Next = after;
        if (after != NULL)
                after->Previous = before;
}
void AttachItem(ListItem* item)
{
        ListItem* before = item->Previous;
        ListItem* after = item->Next;
        if (before != NULL)
                before->Next = item;
        if (after != NULL)
                after->Previous = item;
```

```
}


void DeinitNode(Node* node)
{
        if (node->Label != NULL)
                free(node->Label);
}

typedef struct Graph
{
        Node* Nodes;
        int Count;

        int* EdgeCounters;
        int* EdgeCounts;
        int** Edges;
} Graph;

Graph* InitGraph(int v)
{
        Graph* g = malloc(sizeof(Graph));
        g->Nodes = malloc(sizeof(Node)*v);
        g->Count = v;
        g->Edges = malloc(sizeof(int*)*v);
        g->EdgeCounters = malloc(sizeof(int)*v);
        g->EdgeCounts = malloc(sizeof(int)*v);
        for (int i = 0; i < v; i++)
        {
                g->Edges[i] = NULL;
                g->Nodes[i].Label = NULL;
                g->EdgeCounters[i] = 0;

        }
        return g;
}
void GenerateEdgeData(Graph* g)
{
        int c = g->Count;

        for (int i = 0; i < c;i++)
        {
                for (int j = 0; j < g->EdgeCounts[i]; j++)
                {
                        g->EdgeCounters[g->Edges[i][j]]++;
                }
        }
}
void DeinitGraph(Graph* graph)
{
        int v = graph->Count;
        for (int i = 0; i < v; i++)
        {
                DeinitNode(&graph->Nodes[i]);
                if (graph->Edges[i] != NULL)
                        free(graph->Edges[i]);
        }
        free(graph->Edges);
        free(graph->EdgeCounters);
        free(graph->EdgeCounts);
        free(graph->Nodes);
        free(graph);
}

void BranchAndCut(Graph* graph);


int random(int lower, int upper)
{
        // This technique is instead of the usual modulus,
           to maintain uniformity.
        float r = ((float)rand())/((float)RAND_MAX);
        int diff = upper-lower;
        int r_int = (int)(r*(float)diff);
        return r_int+lower;
}
```

```
void uniqueConstraints(int* array, int maxValue, int count)
{
        int vals[maxValue];
        for (int i = 0; i < maxValue; i++)
                vals[i] = i;

        for (int i = 0; i < maxValue;i++)
        {
                int idx = random(i,maxValue);
                int tmp = vals[i];
                vals[i] = vals[idx];
                vals[idx] = tmp;
        }
        for (int i = 0; i < count;i++)
        {
                array[i] = vals[i];
        }
}


Graph* GenerateRandomizedProblem(int seed, int items, int
    constraints, float sizeLower, float sizeUpper)
{
        int sizeLow = (int)(sizeLower*(float)MaxWeight);
        int sizeUp = (int)(sizeUpper*(float)MaxWeight);
        srand(seed);

        Graph* g = InitGraph(items);
        for (int i = 0; i < items; i++)
        {
                int length = random(sizeLow, sizeUp);
                g->Nodes[i].Weight = length;
                int cons = (constraints*i)/100;
                g->Edges[i] = malloc(sizeof(int)*cons);
                g->EdgeCounts[i] = cons;
                if (cons > 0)
```

```
                        uniqueConstraints(g->Edges[i], i,
                            cons);
        }
        GenerateEdgeData(g);
        return g;

}

Graph* GenerateDepthStudyProblemInstance(int scenes, int
    reps)
{
        Graph* g = InitGraph(scenes*reps);
        for (int i = 0; i < scenes; i++)
        {
                int length = ((i % 7)+2)*15;
                for (int j = 0; j < reps;j++)
                {
                        int idx = i*reps+j;
                        g->Nodes[idx].Weight = length;


                        if (j != reps-1)
                        {
                                g->Edges[idx] = malloc(
                                    sizeof(int)*1);
                                g->EdgeCounts[idx] = 1;
                                g->Edges[idx][0] = idx+1;
                        }
                        else
                        {
                                g->EdgeCounts[idx] = 0;
                                g->Edges[idx] = malloc(
                                    sizeof(int)*0);
                        }
                }
        }
        GenerateEdgeData(g);
```

```
#if PRINT_DEBUG
        int c = g->Count;
        for (int i = 0; i < c; i++)
        {
                printf("Edges from node %i:", i);
                for (int j = 0; j < g->EdgeCounts[i]; j++)
                {
                        printf(" %i", g->Edges[i][j]);
                }
                printf("\n");
        }

        for (int i = 0; i < c; i++)
        {
                printf("EdgeCounter[%i]: %i\n", i, g->
                    EdgeCounters[i]);
        }
#endif
        return g;
}

clock_t start;
int timePassed(void)
{
        clock_t stop = clock();
        return (int)(stop-start)/(CLOCKS_PER_SEC/1000);
}

int main(int argv, char** argc)
{
        #if BENCHING//print immediatly, to avoid lack of
            output when terminating with timeout-tool.
        setbuf(stdout, NULL);
        #endif
        if (argv < 2)
        {
                printf("invalid input.\n");
                return -1;
```

```
    }


    int type = atoi(argc[1]);
    Graph* graph;
    if (type == 1)
    {
            if (argv < 3)
            {
                    printf("invalid input for type\n");
                    return -1;
            }
            int size = atoi(argc[2]);
            graph = GenerateDepthStudyProblemInstance(
                size, size);
    }
    else if (type == 2)
    {
            if (argv < 7)
            {
                    printf("invalid input for type. 
                        required: seed items 
                        constraints sizeLower sizeUpper
                        \n");
                    return -1;
            }
            int seed = atoi(argc[2]);
            int items = atoi(argc[3]);
            int constraints = atoi(argc[4]);
            float sizeLower = ((float)atoi(argc[5]))
                /100.0f;
            float sizeUpper = ((float)atoi(argc[6]))
                /100.0f;
            graph = GenerateRandomizedProblem(seed,
                items, constraints, sizeLower,
                sizeUpper);
    }
    else
```

```
        {
                printf("Unsupported type\n");
                return -1;
        }
        start = clock();
        BranchAndCut(graph);
        int ms = timePassed();
        printf("done - %i\n", ms);

        DeinitGraph(graph);

        return 0;
}

int* _res;
int _resHead;
int* _bestimate;
int _bestimateVal;
ListItem* _listStart;
Graph* _graph;

void Recurse(int step, int currentCost, int remainder, int
    totalRemainder, ListItem* nextInQueue);

void BranchAndCut(Graph* graph)
{
        int c = graph->Count;
        _res = malloc(sizeof(int)*c);
        _resHead = 0;
        _bestimate = malloc(sizeof(int)*c);
        _bestimateVal = c+1;
        _graph = graph;

        int totalWeight = 0;
        for (int i = 0; i < c;i++)
        {
                totalWeight += graph->Nodes[i].Weight;
        }
        printf("Lower␣bound:␣%i\n", (totalWeight+MaxWeight
            -1)/MaxWeight);

        ListItem* listStart = malloc(sizeof(ListItem));
        listStart->Previous = NULL;
        listStart->Next = NULL;
        listStart->NextAtDepth = NULL;
        listStart->Index = -1;
        listStart->Weight = INT_MAX;
        listStart->Depth = -1;
        _listStart = listStart;


        for (int j = 0; j < c; j++)
        {
                if (graph->EdgeCounters[j] == 0)
                        InsertSorted(listStart, graph->
                            Nodes[j].Weight, j, 0);
        }

        Recurse(0, 1, MaxWeight, totalWeight, NULL);
        #if 0
        printf("Best␣(%i):", _bestimateVal);
        for (int i = 0; i < c; i++)
                printf("␣%i", _bestimate[i]);
        printf("\n");
        #endif

        free(_res);
        free(_bestimate);
        _graph = NULL;
}

void Recurse(int step, int currentCost, int remainder, int
    totalRemainder, ListItem* nextInQueue)
{
        if (currentCost >= _bestimateVal || _bestimateVal
            <= (currentCost+(totalRemainder-remainder+
            MaxWeight-1)/MaxWeight))
                return;
        int len = _graph->Count;
        if (step == len)
        {
                //printf("(%.2f,%i)", (float)timePassed()
                    /1000.0f, currentCost);
                #if PRINT_COMBINATION >= 1
                printf("Combination  %i  - %i :",
                    currentCost, timePassed());
                #if PRINT_COMBINATION >= 2
                for (int i = 0; i < len; i++)
                        printf(" %i", _res[i]);
                #endif
                printf("\n");
                #endif

                if (currentCost < _bestimateVal)
                {
                        _bestimateVal = currentCost;
                        for (int i = 0; i < len; i++)
                                _bestimate[i] = _res[i];
                }
                return;
        }

        ListItem* it = nextInQueue;
        if (it == NULL)
                it = _listStart->Next;
        while (it != NULL)
        {
                if (it->Weight < remainder)
                        break;
                it = it->Next;
        }
        if (it == NULL)
                it = _listStart->Next;
            while (it != NULL)
            {
                    int i = it->Index;

                    DetachItem(it);
                    _res[_resHead++] = i;

                    ListItem* firstAtDepth = NULL;
                    ListItem* lastAtDepth = NULL;
                    ListItem* biggestAtDepth = NULL;

                    //Decrement edge counters and update
                        potential nodes
                    for (int j = 0; j < _graph->EdgeCounts[i];
                        j++)
                    {
                            int idx = _graph->Edges[i][j];
                            _graph->EdgeCounters[idx]--;
                            if (_graph->EdgeCounters[idx] == 0)
                            {
                                    ListItem* item =
                                        InsertSorted(_listStart
                                        , _graph->Nodes[idx].
                                        Weight, idx, step+1);
                                    if (firstAtDepth == NULL)
                                    {
                                            firstAtDepth = item
                                                ;
                                            lastAtDepth = item;
                                            biggestAtDepth =
                                                item;
                                    }
                                    else
                                    {
                                            lastAtDepth->
                                                NextAtDepth =
                                                item;
                                            lastAtDepth = item;
                            }
                            if (biggestAtDepth->Weight
                                < item->Weight)
                                            biggestAtDepth
                                               = item
                                              ;
                      }
                }


                //Recurse
                int weight = it->Weight;
                ListItem* nxt = biggestAtDepth == NULL ||
                    it->Next == NULL || it->Next->Weight >
                    biggestAtDepth->Weight ? it->Next :
                    biggestAtDepth;
                if (remainder - weight < 0)
                      Recurse(step + 1, currentCost + 1,
                          MaxWeight - weight,
                          totalRemainder - weight, nxt);
                else
                      Recurse(step + 1, currentCost,
                          remainder - weight,
                          totalRemainder - weight, nxt);

                //Increment edge counters
                for (int j = 0; j < _graph->EdgeCounts[i];
                    j++)
                {
                      _graph->EdgeCounters[_graph->Edges[
                          i][j]]++;
                }

                //Remove added potential nodes
                ListItem* it2 = firstAtDepth;
                while (it2 != NULL)
                {
                      ListItem* tmp = it2->NextAtDepth;
```

```
                                RemoveItem(it2);
                                it2 = tmp;
                        }

                        _resHead--;
                        AttachItem(it);
                        it = it->Next;
                }
}
```

E Benchmarking script

```
cont=1

#file="bench_depth.txt"
#step=1
#limit=100

#file="bench_depth_approx.txt"
#step=10
#cont=10
#limit=180

#file="bench_randomized_constraints.txt"
#cont=0
#step=5
#limit=100

file="bench_randomized_sizes.txt"
cont=25
step=5
limit=60

#file="bench_test.txt"
#step=1
#cont=1
```

```
#limit=1

echo "cont,lb,off5_v,off5_t,off1_v,off1_t,best_v,opt_v,
   best_t,done_t" > $file

while [ $cont -le $limit ]; do

        #res=$(timeout 120 ./solver 1 180) #testing

        #res=$(timeout 120 ./solver 1 $cont) #bench_depth.
            txt
        #res=$(timeout 120 ./solver 1 $cont) #
            bench_depth_approx.txt

        #
                              type seed items constraints
            sizeLower sizeUpper
        #res=$(timeout 30 ./solver 2 0 150 10 20 30)
        #res=$(timeout 120 ./solver 2 0 1000 $cont 10 50) #
            bench_randomized_constraints
        res=$(timeout 120 ./solver 2 0 1000 10 $(($cont-20)
            ) $(($cont+20))) #bench_randomized_sizes


        lb=-1

        off5_t=-1
        off5_v=-1
        off5_l=-1
        off1_t=-1
        off1_v=-1
        off1_l=-1

        best_t=-1
        best_v=-1

        done_t=-1
```

```
opt_v=-1

while read -r line; do
        #echo "ln:␣$line"
        if [[ $line == "Lower␣bound"* ]]; then
                lb=`echo $line | cut -d'␣' -f 3`
                off5_l=$((($lb * 105+99)/100))
                off1_l=$((($lb * 101+99)/100))
                #echo "off5_l:␣$off5_l"
                #echo "off1_l:␣$off1_l"
                #echo "lower␣bound:␣$lb"
        fi
        if [[ $line == "Combination"* ]]; then
                cmb=`echo $line | cut -d'_' -f 2`
                time=`echo $line | cut -d'␣' -f 4`
                if [[ $off5_t -eq -1 ]] && [[ $cmb
                    -le $off5_l ]]; then
                        off5_t=$time
                        off5_v=$cmb
                        #echo "5off␣($cmb)␣-␣$time"
                fi
                if [[ $off1_t -eq -1 ]] && [[ $cmb
                    -le $off1_l ]]; then
                        off1_t=$time
                        off1_v=$cmb
                        echo "1off␣($cmb)␣-␣$time"
                fi
                best_v=$cmb
                best_t=$time
                #echo "mtc:␣$cmb␣-␣$time"
        fi
        if [[ $line  == "done"* ]]; then
                done_t=`echo $line | cut -d'␣' -f
                    3`
                opt_v=$best_v
                echo "Done:␣$done_t"
        fi
done <<< "$res"
```

```
        echo "Done␣with␣iteration:␣$cont.␣$best_v/$lb␣−␣
            $done_t"
        echo "$cont,$lb,$off5_v,$off5_t,$off1_v,$off1_t,
            $best_v,$opt_v,$best_t,$done_t" >> $file

        cont=$(($cont+$step))
done

#echo "Best␣found␣($best_v)␣−␣$best_t"
```