Container Packing with Real World Multi Drop Constraints

Guide: Prof.Narayanaswamy
CS19B079 - KR Hariharan

Indian Institute of Technology, Madras

CS4900 - Undergraduate Research Credits

Jan-May 2022

Contents

1 Introduction

To minimize the cost of shipping, it is important to make sure that vehicles used to transport goods are filled as efficiently as possible. If more goods can be transported in each trip, lesser trips will be required, reducing the cost of shipping per item.

Thus, given the dimensions of the transport vehicle, and of each individual item to be transported, we would like to pack the most number of items possible in the vehicle. When creating a packer for real world implementations, we must also respect real world feasibility/constraints, about which we will discuss in the following section.

This problem is essentially a 3 dimensional bin packing with constraints, which is in turn, an extension of the 1 dimensional bin packing problem. As the 1 dimensional, or Knapsack problem, is NP-Hard, we can see that the Container Packing problem is also NP-Hard. For this NP-Hard problem, we would like to build a heuristic algorithm that gives us a satisfactorily efficient solution in a reasonable time.

2 Real World Constraints

As this is a problem that is being approached with a real-world application in mind, we need to consider real-world constraints while building an algorithm.

Stability The algorithm must ensure that any loading strategy that it provides makes sure that each package is loaded in a safe and stable manner. This adds constraints such as:

- Packages must be loaded in a stable manner to prevent movement/falling during transport. For example, for a box to be loaded at a particular position, its base must be fully supported in that position.
- Some items being transported may be fragile and hence may have to be loaded in specific orientations only.
- There may be constraints such that no other consignments may be packed on top of certain fragile consignments. Consignments often also have a limit on the weight which can be packed on top of them.

Unloading Costs - B2B and B2C When trying to optimize B2B shipping, like from a factory to the warehouse, or from warehouse to store, required constraints primarily focus only on stability. However, when creating a loading strategy for a home delivery service, we must also factor in unloading costs.

In B2B shipping, it is safe to assume that there is only 1, or at most a few unloading points. Hence, the goods can be loaded and unloaded into the vehicle in almost any arbitrary order, or with some grouping.

On the other hand, in the case of Home Delivery of (consumer) goods, each item will have a different unloading point. Based on the order in which each destination is reached, the goods will have a particular unloading order.

As it is unfeasible to unload and reload many items (that are yet to be delivered) at each unloading point, the loading strategy must be developed such that, when unloading an item, it is easily accessible within the vehicle.

3 Previous Work

The 1 dimensional knapsack problem is a well studied NP-Hard Optimization problem. Many approximation algorithms have been proposed for the same, with a variety of approaches, such as greedy approximation and dynamic programming.

Approximation algorithms and heuristics have been put forward for the multi dimensional knapsack problem, using a variety of different approaches, such as genetic algorithms, tabu search and simmulated annealing.

For example, one such approximation algorithm, a Multiple Choice Bin Packing Harmonic Algorithm, was proposed by Eklavya Sharma. In this algorithm, the first d-1 Dimensions of rectangles are harmonically rounded up and "flattened" to 1 dimension, so as to create 2 dimensional rectangles. These rectangles are then packed into "shelves", in such a way that rectangles with similar heights are placed in the same shelves. The shelves are then packed into flattened Bins, and this 2 dimensional packing is then "inflated" to give the corresponding d Dimensional packing.

While such approximation algorithms give a good approximation guarantee, they are not designed to factor in real world constraints. The items/rectangles are packed in regardless of vertical support, and there is little scope for factoring in unloading cost (within a single bin). In order to tackle these practical challenges, many heuristics have been proposed.

For the multi drop case, one of the first heuristics were put forward by Bischoff and Ratcliff. In this heuristic, boxes are packed one by one, based on their drop order. The box to be packed in each step, and its orientation and location are determined by few arbitrary criterion prioritizing maximum stacking and minimum protrusion.

Building on this incremental solution, few other "optimizer and packer" solutions have been put forward. These approaches have two parts:

- a "packer", which packs items/rectangles in a deterministic and stable manner, given an ordered list of items. Little to no optimization is done in this step.
- an "optimizer", which uses the packer as a subroutine to identify an optimal packing by varying the order/position/orientation of items/rectangles being packed.

An example of this is the accelerated tree search method proposed by Christensen and Rousøe. In each step, the packing branches out into a number of child nodes, based on the number of different items/orientations/positions being experimented with. The volume optimization of each of these partial packings are estimated using the packer for the remaining items, and an arbitrary number of partial packings with the highest predicted volume optimizations are retained.

Another example is the simulated annealing approach put forward by Ceschia and Schaerf. Simulated Annealing is used to determine an efficient ordering of packing, based on the "score" of each packing, determined by a number of factors, such as unloading cost and volume optimization.

4 Knowledge Representation

Primary Entities: The proposed problem has two primary entities - containers, and consignments.

- **Consignments**: The packages that are to be delivered.
  - Dimensions - length (l), breadth (b), and width (h)
  - Weight
  - Pickup location
  - Drop off location
  - Stackable - Boolean value which specifies if other consignments may or may not be stacked on top of this consignment.
  - Orientations - The possible orientations in which a consignment can be packed. For example, a fragile consignment may be allowed to be packed only in specific orientations (packages specifying “This way up”).
  - Location - Location where the consignment is packed (x, y, z)
  - Orientation - The orientation in which the consignment is packed (l1, b1, h1)
- **Containers**: The containers into which consignments are packed.
  - ID - Used to identify container type
  - Internal Dimensions - packable length (L), breadth (B), and width (W)

Predicates: Boolean operators that are useful to define actions related to the problem:

- dest(c, loc) - true if drop off point of consignment c is loc
- packed(c) - true if consignment c has been packed into the vehicle
- contains(C, c) - true if container C contains consignment c.
- packable(C, [x1,y1,z1]) - true if coordinate (x1, y1) in the container has been packed upto a height of z1
- packedPoint(C, [x1, y1, z1]) - true if an consignments exists at coordinate (x1, y1, z1) in the container.
- position(C, c, [x,y,z], [l1, b1, h1]) - true if consignment c is placed in container C at the cuboidal space starting from (x,y,z) with orientation (l1, b1, h1).

Action: Operations associated with objects defined above that are required to solve the given problem statement. The preconditions must be met for an action to be performed. Performing the action will lead to an effect (on the predicate values).

Load (consignment c, container C, int x, int y, int z, int l1, int b1, int h1)

- Preconditions:
  - $\neg$packed(c)
  - (l1, b1, h1) $\in$ c.Orientations
  - z+h1 $\leq$ C.H
  - $\forall$ (x1, y1) : x $\leq$ x1 $\leq$ (x+l1) $\wedge$ y $\leq$ y1 $\leq$ (y+b1) . packable(C, [x1, y1, z])
  - $\forall$ (x1, y1) : x1 $>$ (x+l1)$\wedge$ y $\leq$ y1 $\leq$ (y+b1) . $\neg$ packedPoint(C, [x1, y1, z+h1])
- Effect:
  - packed(c)
  - contains(C, c)
  - c.Location $\leftarrow$ (x, y, z)
  - c.Orientation $\leftarrow$ (l1, b1, h1)
  - $\forall$ (x1, y1) : x $\leq$ x1 $\leq$ (x+l1) $\wedge$ y $\leq$ y1 $\leq$ (y+b1) . $\neg$packable(C, [x1, y1, z])
  - if c.Stackable, then
    - $\forall$ (x1, y1) : x $\leq$ x1 $\leq$ (x+l1) $\wedge$ y $\leq$ y1 $\leq$ (y+b1) . packable(C, [x1, y1, z+h1])

    else
    - $\forall$ (x1, y1) : x $\leq$ x1 $\leq$ (x+l1) $\wedge$ y $\leq$ y1 $\leq$ (y+b1) . packable(C, [x1, y1, C.H])
  - $\forall$ (x1, y1) : x $\leq$ x1 $\leq$ (x+l1) $\wedge$ y $\leq$ y1 $\leq$ (y+b1) $\wedge$ z $\leq$ z1 $\leq$ (z+h1). packedPoint(C, [x1, y1, z1])

Unload (consignment c, container C)

- Preconditions:
  - packed(c)
  - contains(C, c)
  - $\forall$ (x1, y1) : x1 $>$ (c.Location.x+l1) $\wedge$ C.Location.y $\leq$ y1 $\leq$ (C.Location.y+b1) . $\neg$packedPoint(C, [x1, y1, z+h1])
  - $\forall$ (x1, y1, z1) : c.Location.x $\leq$ x1 $\leq$ (c.Location.x+C.Orientation.l1) $\wedge$ C.Location.y $\leq$ y1 $\leq$ (C.Location.y+C.Orientation.b1) $\wedge$ z1 $>$ (c.Location.z+C.Orientation.h1). $\neg$packedPoint(C, [x1, y1, z1])
- Effect:
  - $\neg$packed(c)
  - $\neg$contains(C, c)
  - $\forall$ (x1, y1) : c.Location.x $\leq$ x1 $\leq$ (c.Location.x+c.Orientation.l1) $\wedge$ c.Location.y $\leq$ y1 $\leq$ (c.Location.y+c.Orientation.b1) . $\neg$packable(C, [x1, y1, z+h1])

- $\forall$ (x1, y1) : c.Location.x $\leq$ x1 $\leq$ (c.Location.x+c.Orientation.l1) $\wedge$
  c.Location.y $\leq$ y1 $\leq$ (c.Location.y+c.Orientation.b1) . $\neg$packable(C, [x1, y1, C.H])
- $\forall$ (x1, y1) : c.Location.x $\leq$ x1 $\leq$ (C.Location.x+c.Orientation.l1) $\wedge$
  c.Location.y $\leq$ y1 $\leq$ (c.Location.y+c.Orientation.b1) . packable(C, [x1, y1, c.Location.z])
- $\forall$ (x1, y1, z1) : c.Location.x $\leq$ x1 $\leq$ (c.Location.x+c.Orientation.l1) $\wedge$
  c.Location.y $\leq$ y1 $\leq$ (c.Location.y+c.Orientation.b1) $\wedge$
  c.Location.z $\leq$ z1 $\leq$ (c.Location.z+c.Orientation.h1). $\neg$packedPoint(C, [x1, y1, z1]
- c.Location $\leftarrow$ null
- c.Orientation $\leftarrow$ null

5 Approach and Algorithm

The heuristic built in this project is based on the heuristics mentioned above. The idea is to simplify and combine the different optimization strategies. In this heuristic, simulated annealing is used to find an optimal packing order, and accelerated tree search is used to find optimal orientation/position for each item.

To start, we require a packer algorithm. Given an empty or partially packed container, and a ordered list of Items, this packer must pack as many of the items as possible, while respecting the order to ensure zero unloading cost.

Algorithm 1 Packer
Time complexity: $O(n^2 * d^2)$
n - number of consignments
d - largest consignment dimension

Require:

- Ordered list of Items to be packed (following order of drop)
- Container that may be empty or partially packed

Ensure: As much of Items as possible is packed in reverse order (item to be dropped first is packed last)

```
for all item in reverse(Items) do
    for all orientation in orientations do
        if orientation is valid for item then
            if (item, orientation) fits in Container then
                Container.pack(item, orientation)
                break
            end if
        end if
    end for
end for
```

To start the packing, an Initializer is run, which is used to determine the order of the packing. To ensure no unloading cost, it is important that in the list of items, the items being dropped off together are also kept together in the list order.

To start off, the items of each drop-off point are sorted in descending order of volume, followed by largest dimensions. This allows for larger packages and packages with abnormal dimensions to be packed first, followed by smaller packages (which can be packed on top of these packages).

Then, for the items pertaining to each drop off point (starting with the last drop off point), simulated annealing is performed to obtain an optimal intra-destination ordering. The overall volume optimization obtained using packer algorithm on the intermediate list orders is used as the score for simulated annealing.

Algorithm 2 Initializer

Time complexity: $O(nlogn + t * s * k * n^2 * d^2)$

t - number of temps

s - number of swaps per temp

k - number of drop points

n - number of consignments

d - largest consignment dimension

Require: Ordered list of drop Destinations, each with an unordered list of items (destItems) to be packed

Ensure: Generate an efficient order of packing items, without unloading costs.

```
for all dest in Destinations do
    sort(dest.destItems)          ▷ Sorting is based on volume and largest dimension
end for
for all dest in reverse(Destinations) do
    simulatedAnnealing(dest.destItems)
end for
```

Once the order of items is obtained, accelerated tree search is used to pack the items.

At each step, an item is packed in all possible orientations, in all available partial packings. Then, the packer is used to predict the volume optimization these partial packings will lead to. The t "most promising" partial packings are retained at each step.

Algorithm 3 Optimized packer
Time complexity: $O(3 * t * n^3 * d^2)$
t - tree width (arbitrary)
n - number of consignments
d - largest consignment dimension

Require:

- Ordered list of Items to be packed (following order of drop)
- maximum tree width
- Empty Container
- empty list (Possibilities) of pair(predicted volume optimization, partially packed containers)

Ensure: As much of Items as possible is packed in reverse order (item to be dropped first is packed last)

```
n ← Items.size()
Possibilities.insert(packer(Items, Container), Container)
for all item in reverse(Items) do
    for i ← 1, Possibilities.size() do
        Container ← Possibilites[i].second
        Possibilities.insert(packer(Items[1, item.pos()-1], Container), Container)
        for all orientation in orientations do
            if orientation is valid for item then
                if (item, orientation) fits in Container then
                    tempC ← Container
                    tempC.pack(item, orientation)
                    Possibilities.insert(packer(Items[1, item.pos()-1], tempC), tempC)
                end if
            end if
        end for
        Possibilites.remove(i)
    end for
    if Possibilities.size() > treeWidth then
        sort(Possibilites, descending)
        Possibilities.resize(treeWidth)
    end if
end for
```

6 Testing and Conclusions

Testing was done on dataset derived from thpack container datasets of the OR library. For each container, its dimensions an unordered set of consignments to be packed are given in the datatset. These consignments are randomized and grouped to simulate multi drop condition. The consignments are also randomly marked as fragile/unstackable with a probability of 10%.

Thpack 1 Dataset - Packer

Container No.s	Average Volume Optimization (%)	Min Volume Optimization (%)	Max Volume Optimization (%)	Standard Deviation	Average Time Taken (s)
1-10	56.4526	45.1669	71.3089	7.7225	0.0214
11-20	54.1530	48.9626	60.0416	3.4987	0.0180
21-30	56.5442	49.2115	67.5381	5.4262	0.0191
31-40	58.8034	48.5779	67.6819	5.3410	0.0194
41-50	53.7535	35.1441	66.9361	10.0942	0.0214
51-60	56.4266	46.2821	60.9765	4.4097	0.0225
61-70	55.6917	48.2784	62.7760	4.5760	0.0256
71-80	54.7408	48.7211	60.3202	3.7839	0.0215
81-90	53.8177	44.3306	64.2941	6.2558	0.0184
91-100	56.9455	48.0895	72.6355	7.4764	0.0194
total	55.7329	35.1441	72.6355	6.3647	0.0207

Using just the packer algorithm gives an average volume optimization of approximately 55.7%. A significant loss of volume usage can be attributed to stacking constraints, as the average exceeded 60% when there were no such constraints.

Thpack 1 Dataset - Optimized Packer of Tree width = 2 - No Initializer

Container No.s	Average Volume Optimization (%)	Min Volume Optimization (%)	Max Volume Optimization (%)	Standard Deviation	Average Time Taken (s)
1-10	70.8470	60.1672	81.7816	6.3698	21.8635
11-20	70.0836	62.7956	78.3125	5.0926	21.2312
21-30	68.7921	63.5473	76.0839	3.9670	25.5826
31-40	70.9818	67.4030	76.4442	3.1202	27.0479
41-50	69.5811	62.4292	82.1147	6.1908	19.9039
51-60	69.1315	65.2474	73.9239	2.4393	32.2281
61-70	68.3227	59.9089	79.8776	6.2538	38.7621
71-80	67.8966	63.7226	74.0580	2.5796	25.5101
81-90	67.6044	59.4635	76.4209	4.8313	21.1479
91-100	69.2630	59.2838	78.9939	5.6670	22.9451
total	69.2504	59.2838	82.1147	4.9946	25.6222

The optimized packer does not involve reordering the consignment list. Hence, using the optimized packer without the Initializer simulates the extreme case, where every consignment has a different drop off point (such

as in B2C delivery) and there is very little scope for modifying the loading and unloading order.

Even using the optimized packer with a small decision tree width leads to a significant increase in volume optimization, with an average of 69.25%. However, this is associated with a large time penalty, as the optimized packer essentially runs the packer a few hundred times.

Thpack 1 Dataset - Initializer and Optimized Packer of Tree width = 2

Container No.s	Average Volume Optimization (%)	Min Volume Optimization (%)	Max Volume Optimization (%)	Standard Deviation	Average Time Taken (s)
1-10	69.2232	58.5902	77.5758	6.8778	28.4428
11-20	69.8118	60.1644	78.1005	5.2639	29.1854
21-30	70.1590	61.3494	79.4730	4.5516	36.5989
31-40	74.2223	68.2082	81.9068	4.2272	37.9983
41-50	69.3255	54.2850	79.6473	7.2198	27.3539
51-60	70.7446	64.8987	75.0321	2.9865	48.3626
61-70	69.5451	64.5408	81.1013	4.8540	55.4783
71-80	70.2866	63.0999	73.4118	2.8893	36.7394
81-90	72.1313	66.6490	77.1415	3.4110	31.3373
91-100	68.5404	60.6266	79.1365	4.8004	33.7959
total	70.3990	54.2850	81.9068	5.1585	36.5293

When the Initializer is introduced, there is a marginal increase of 1.15% in the volume optimization. However, this is associated with a noticable increase in time.

Thpack 1 Dataset - Optimized Packer of Tree width = 5 - No Initializer

Container No.s	Average Volume Optimization (%)	Min Volume Optimization (%)	Max Volume Optimization (%)	Standard Deviation	Average Time Taken (s)
1-10	71.3227	60.2801	81.4506	6.3827	49.7766
11-20	72.8077	67.0614	78.9188	4.4755	51.6827
21-30	70.9203	67.4537	79.2485	3.7603	59.8553
31-40	73.6122	68.2082	83.0158	4.7485	64.3832
41-50	70.6663	59.1210	81.8143	6.6582	47.1557
51-60	71.4983	63.9931	78.2956	4.3378	75.2717
61-70	70.1844	58.3027	78.9988	6.1864	86.6536
71-80	68.3904	63.9439	74.9533	3.4744	55.0332
81-90	70.0542	61.3800	77.1546	5.8284	49.9190
91-100	71.3010	64.8864	79.3494	3.6661	55.1274
total	71.0757	58.3027	83.0158	5.2662	59.4858

When the decision tree width is increased to 5, there is once again marginal increase of 1.82% in volume optimization. This is associated with a time of packing that is a little over doubled.

Thpack 1 Dataset - Initializer and Optimized Packer of Tree width = 5

Container No.s	Average Volume Optimization (%)	Min Volume Optimization (%)	Max Volume Optimization (%)	Standard Deviation	Average Time Taken (s)
1-10	71.7999	58.3949	82.3838	6.6516	60.3424
11-20	71.5006	61.4556	82.1074	6.5686	61.7267
21-30	70.1279	63.2311	79.2123	4.7851	69.6731
31-40	75.0543	69.8181	80.4226	4.3781	78.9376
41-50	69.3277	54.2472	81.1697	6.7173	54.6508
51-60	71.7336	61.9173	79.3035	4.8144	92.3083
61-70	70.4200	59.9970	80.4202	5.5747	104.8325
71-80	71.4717	61.2877	78.5057	4.9943	65.3703
81-90	69.3832	58.7975	75.2059	5.5414	60.4284
91-100	72.2468	66.6049	80.7562	4.0404	67.8263
total	71.3066	54.2472	82.3838	5.7098	71.6096

Adding the Initializer to the Optimized Packer of decision tree width 5 only results a very small average increase.

7 Further Work

Initializer and Packer Currently, the intra-destination consignment order is initialized to descending order of size/dimensions. This can be made more robust, such as using the Bischoff-Ratcliff model's tiebreakers, without much of an increase in algorithm time.

Similarly, the current packer, which uses first orientation fit, can be tested and compared with other approaches, such as first-fit, best-fit, max-height, min-height and weight-balanced.

Simulated Annealing Currently, the simulated annealing implementation randomly chooses intra-destination consignments to swap. Rather, a more targeted selection of consignments to swap could lead to better results. For example, ensuring at least on of the two consignments being swapped is non-stackable.

Weight Balancing As another real world consideration, we would like to ensure that a container is packed in such a way that it is reasonably balanced, at least breadth-wise. This would spread out the stress on the vehicle/trailer, and reduce any topple risk.

In addition to this, consignments themselves often have a limit to the amount of weight that can be packed on top of them without damaging the consignment. These restrictions must also be respected by a packer with real world implementations.

8 Bibliography

- Harmonic Algorithms for Packing d-Dimensional Cuboids into Bins - Eklavya Sharma
- Issues in the development of approaches to container loading - E.E.Bischoff and M.S.W.Ratcliff
- Container loading with multi-drop constraints - Søren Gram Christensen and David Magid Rousøe
- Local search for a multi-drop multi-container loading problem - Sara Ceschia and Andrea Schaerf

9 Acknowledgements

I would like to thank Prof.N.S.Narayanaswamy for this opportunity and his time and guidance throughout the project, the Department of Computer Science and Engineering for offering this research opportunity, and Prof.Shweta Agrawal for coordinating the UGRC process.

I would like to thank the members of the Logistics Lab for their help: Anuj, Deekshit and my team members - Soham, Ishaan, Parth, Varun, and Shree Vishnu. I would also like to thank Madhav, Ankit, and Hakesh for their support.