Chapter 1

FrontPage - Missing

Chapter 2

Acknowledgements

Chapter 3

Abstract

Contents

List of Figures

List of Tables

Chapter 4

Introduction

Chapter 5

Literature review

Chapter 6

Problem description and mathematical formulation

Chapter 7

Solution algorithms

7.1 Beam Search

Beam Search (BS) is an heuristic graph search algorithm designed for systems with limited memory where expanding every possible node is unfeasible. The idea behind BS is to conduct a iterative truncated breadth-first search where, at each iteration, expanded nodes are ranked based on an heuristic and only the best ones are further explored. To perform BS one must define the node structure, an expansion function to generate new nodes from an existing one, an evaluation function to compare nodes between eachother and a function to determine if a node is a solution to the problem.

Let s_i be a node in the graph of possible solutions of the 3DBPP, s_i can be seen as an instance of the problem where a sequence of placements has taken place. An expansion of a node s_i generates a new node s_j where a placement has occured for a given set of items. Since evaluating possible expansions can be computationally easier than computing new node data structures, a *Commit* function is defined which applies a pre-computed expansion by updating the supporting data structures in its node.

Given S_{init} the set of initial nodes to start from and k the number of best nodes to expand at each iteration, the described procedure is rappresented by algorithm 1.

Algorithm 1: Beam search

input : S_{init}, k
output: S_{best}
$S \leftarrow S_{init}$
$S_{final} \leftarrow \emptyset$
repeat
  $S_{new} \leftarrow Expand(S)$ (Algorithm 2)
  $S_{final} \leftarrow S_{final} \cup \{\forall s_i \in S_{new} : IsFinal(s_i)\}$
  $S_{new} \leftarrow S_{new} \setminus S_{final}$
  $S_{new} \leftarrow Sort(S_{new})$
  $S \leftarrow \{\forall Commit(s_i) \in S_{new} : i \in \mathbb{Z}^+ \wedge i \leq k\}$
until $S \neq \emptyset$
$S_{final} \leftarrow Sort(S_{final})$
return $s_0 \in S_{final}$

The *Expand* function computes new nodes which rappresent possible placements that can be made starting from a given packing. Each node contains a number of supporting data structures that are updated across iterations by the *Commit* function. Let S be the set of nodes that need to be expanded, each node s is rappresented by a structure which contains

- *bins*: the set of open bins
- *unpacked*: the set of items that aren't assigned to any bin
- s_b: a substructure which cointains informations about a bin b

Let $GroupByFamily(I)$ be a function which operates on a set of items and outputs a set of tuples $(family, I)$ where $family$ is the family of the set I of items. A new set of nodes can be computed by using an underlying 3DSPP heuristic which evaluates the best move for each family of items for each currently opened bin. The described procedure is detailed in algorithm 2

Algorithm 2: Expand

```
input  : S
output: S_new
forall s ∈ S do
    S_new ← ∅
    I_family ← GroupByFamily(unpacked)
    placed ← false
    forall (family, I) ∈ I_family do
        forall bin ∈ bins do
            placement ← SPBestInsertion(s_b, I) (Algorithm 3)
            if placement ≠ ∅ then
                placed ← true
                S_new ← S_new ∪ Next(s, placement)
            end
        end
    end
    if placed = false then
        S_new ← S_new ∪ OpenNewBin(s)
    end
end
return S_new
```

7.1.1 Scoring States

In order to sort nodes, a scoring function needs to be defined over the nodes. To allow the BS to explore better solutions the scoring function can't be as flat as the objective function defined in the mathematical formulation of the problem.

7.2 Support Plane

Support Plane (SP) is an heuristic introduced in this thesis based on an underlying 2DBPP heuristic which is used to evaluate feasible expansions of a given node in the BS. The proposed heuristic ensures that the constraint of support isn't violated. The idea at the base of SP is to build a solution to the 3DSPP by filling 2D planes called support planes.

Each support plane can be characterized by the triple $S_z = (z, I_{support}, I_{upper})$ where

- z: the height of the plane

– $I_{support}$: the set of the items that can offer support to items placed on the plane

– I_{upper}: the set of items that will be obstacles to potential new items placed on the plane

Let s_b be a data structure containing

– *planes*: the set of triples S_z of support planes to evaluate, ordered in ascending z order

– *aabb*: the AABB Tree of the items placed in the evaluated bin

– (W_b, D_b, H_b): the dimensions of the bin

Let *coords* be the set of possible coordinate changes which allow for the problem to evaluate placements starting from different corners of the bin.

Given a function $IsFeasible(i, bin, I_{support}, I_{upper}, aabb)$ which evaluates if a packing of item i in bin *bin* is feasible, and the function $ComparePacking(p, p')$ which defines a ranking over placements in the same plane, the SP algorithm can be written as algorithm 3.

Algorithm 3: SP Best Insertion

```
input  : s_b, I
output: placement
placement ← ∅
forall S_z ∈ planes do
    I_p ← I \ {∀i ∈ I : z + i.h > H_b}
    forall change ∈ coords do
        I'_upper ← CoordinateChange(change, I_upper)
        I'_p ← CoordinateChange(change, I_p)
        p' ← SPRectanglePackingWithObstacles(W_b, D_b, I'_upper, I'_p)
        p ← CoordinateChange(change, p')
        p ← {∀i ∈ p : IsFeasible(i, bin, I_support, I_upper, aabb)}
        if ComparePacking(placement, p) then
            placement ← p
        end
    end
    if placement ≠ ∅ then
        return placement
    end
end
return placement
```

To evaluate a packing on a plane an heuristic to solve the 2DBPP is used with the introduction of fixed placements which rappresent items on other planes that will be obstacles in the current one.

Given the dimensions of the 2D bin (W_b, D_b), the set of obstacles I_o and the set of items to pack I_p a new placement can be computed following algorithm 4

Algorithm 4: SP Rectangle packing with obstacles

input : W_b, D_b, I_o, I_p
output: P
$P \leftarrow \emptyset$
$2dPacking \leftarrow \emptyset$
foreach $i \in I_o$ **do**
    `//Initialize the 2D bin packing instance with each obstable already placed`
    $2DPlaceRect(2dPacking, i)$
end
repeat
    `//Pack untill full`
    $p \leftarrow 2DPackRect(2dPacking, W_b, D_b, i)$
    $P \leftarrow P \cup \{p\}$
until $p \neq \emptyset$
return P

Once the k best nodes are selected the placements evaluated for each node are applied and the *Commit* function updates every datastructure in S, including the ones used by SP. Given the instance that generated one of the placements selected and p the current set of support planes, z_{min} the minimum z coordinate for which a placement was made in the related bin starting from the current state, I the set of items placed, U the set of items unpacked. Since placements are evaluated in order starting from the lower z possible, if no placement was made in an open support plane with z lower than z_{min}, the plane can be pruned to avoid further evaluations. The algorithm which updates the structures for a given SP instance is rappresent by algorithm 5.

Algorithm 5: SP Apply and Filter

input : s_b, I, z, z_{min}, t
output: s'_b
`//Filter bad planes`
$P' \leftarrow planes \setminus \{\forall S_z \in planes : z \leq z_{min}\}$
`//Apply insertion`
$B \leftarrow placed \cup I$
$U \leftarrow unpacked \setminus I$
$T \leftarrow aabb$
forall $i \in I$ **do**
  $T \leftarrow InsertAABB(i, T)$ `//If balanced` $O(log(n))$
  $generate \leftarrow true$
  forall $S'_z \in P'$ **do**
    `//Based on the distance from the top of the item`
    $dz \leftarrow S'_z.z - i.z_{max}$
    if $0 \leq dz \leq t$ **then**
      $generate \leftarrow false$
      $S'_z.I_{support} \leftarrow S'_z.I_{support} \cup i$
    end
    else if $dz < 0$ **then**
      $S'_z.I_{upper} \leftarrow S'_z.I_{upper} \cup i$
    end
  end
  if *generate* **then**
    $P' \leftarrow P' \cup (i.z_{max}, \{i\}, \emptyset)$
  end
end
return $Update(s_b, P', B, U, T)$

7.2.1 Scoring Insertions

7.3 Max Rects

7.4 Feasibility

7.4.1 AABB Tree

In order to check the feasibility of a given insertion, a way of checking for intersections is needed. Since every box in a solution is axis aligned and

defined by a static bounding box an Axis Aligned Bounding Box Tree (AABB Tree) is constructed and updated throughout the various nodes of the search. AABB Trees are accelleration structures which allow the computation of intersections given a bounding box with a time complexity of $O(\log n)$ where n is the number of items placed.

Chapter 8

Computational experiments

Chapter 9

Conclusions