NYILATKOZAT

Név: LEITEREG MIKLÓS
ELTE Természettudományi Kar, szak: ALKALMAZOTT MATEMATIKUS MSC.
NEPTUN azonosító: EQT8YB
Szakdolgozat címe: PACKING AND SUPPORT PLANNING ALGORITHMS IN 3D PRINTING

A **szakdolgozat** szerzőjeként fegyelmi felelősségem tudatában kijelentem, hogy a dolgozatom önálló szellemi alkotásom, abban a hivatkozások és idézések standard szabályait következetesen alkalmaztam, mások által írt részeket a megfelelő idézés nélkül nem használtam fel.

Budapest, 2020.12.29

Leitereg Miklós

a hallgató aláírása

Packing and support planning algorithms in 3D printing

Thesis submitted for the degree of
MSc in Applied Mathematics

Leitereg Miklós

Supervisor:

Jüttner Alpár

Operations Research Department

Eötvös Loránd University
Faculty of Science
2020

Acknowledgement

I would like to thank my supervisor, Alpár Jüttner for the helpful guidence, and the valualbe time spent on our consultations.

Contents

1 Introduction

Additive manufacturing, also known as 3D printing is an increasingly important industry with widespread applications. In this thesis we consider optimization problems arising from this field. Namely the positioning of the objects in the printing space is an important nontrivial problem. Our aim here will be to fill available space as efficiently as possible, or conversely pack the objects in as small a space as possible. This is important because the cost of this kind of production is not only dependent on the objects that we print, but also the number of printing rounds needed. In addition it is important to mention that many 3D printing techniques are slow, and the time taken can depend on the number of rounds, so reducing it is crucial.
The other task we need to solve is providing support structures for the objects already placed in 3D-space. This is typical task that needs to be solved, inherent to most additive manufacturing processes.

In this thesis we will approach the problem from the perspective of metal 3D printing. In this process, in each step metal powder is spread in a thin layer above the already present materials, then the next 2D slice of the desired object is fused onto the already existing material by selectively melting the powder with a high powered laser.
The heat generated by the laser creates large stresses in the material which means that strong support structures are needed.
The process is slow compared to other additive manufacturing techniques, so the speed gained by printing multiple objects at once is important. This is our motivation for trying to optimally pack objects in the printing space.
Similar to plastic printing, the orientation of the objects influence the quality of the product. But throughout this thesis, we not only consider vertical axis fixed, but most of the time think of the orientation as completely fixed. So our packing algorithms will only translate the objects. This is done because the printing process is not independent of direction, which can impact the outcome. Also this allows us to design more efficient algorithms. We will detail the potential to allow rotation in the sections when possible.

In this thesis, we will outline multiple algorithms for solving both the packing and the support planning problems. Some of these algorithms are implemented, and evaluation is provided on the performance, and the potential strengths and weaknesses.
In the first chapter we present different approaches to the packing problem. We introduces some basic concepts of packing, such as boxpacking. Presenting multiple types of algorithms allows us to adapt to the practical requirements of different types of objects and printing environments.
In the second chapter we present a novel way of algorithmically generating support structures. We also show different methods to solve the optimization problem arising from this new approach.
In the final chapter we present the running times and the quality of the solutions provided by the algorithms.

2 The Packing Problem

2.1 Introduction of the Packing Problem

In the packing problem we are given objects, and the dimensions of the printing space. The objects are given as meshes. A Mesh is a set of triangles making up a closed surface. Together with each triangle is given a normal vector of the triangle, the normal vector is understood to be pointing to the inside of the object. While we are discussing the packing problem, in the input only the relative position of the vertices matter, translation in space doesn't alter the mesh. In the output however, the absolute position of the mesh in coordinate system is what determines the placement. This is how we provide the placement. This means we can think of placement as a translation for each object. Based on this we can define two optimization problems.

Maximum filling problem
In this optimization problem, we optimize over all subset-placement pairs, where no two objects in the subsets can intersect each other given their respective placements. The following is the formal definition of the problem.
Let M be the set of meshes, B the printing volume

$$\max_{S \subseteq M, P} \sum_{o \in S} V(o)$$

Where $V(o)$ is usually the total volume if the chosen object
$P \ni p_s : \mathbb{R}^3 \to \mathbb{R}^3$ translation or translation and rotation, for each member of S
Satisfying: $\nexists s_1, s_2 \in S : p_{s_1}(s_1) \cap p_{s_2}(s_2) \neq \emptyset$ and $\forall s \in S : p_s(s) \subseteq B$

This problem is relevant in cases, where it is unknown what objects may be required in the future, or when not all objects need to be printed immediately. The most common scenario where this is the case, is when the objects come in throughout time, in an "online" environment. In order to be able to handle different use cases, we need to allow V, the objective function to not always be the volume, but an arbitrary reward. If we can solve this more general version, we can simulate scenarios, where objects might need to be printed earlier, or objects that have more value.

Minimum number of runs problem
In this variant the input is the same as before, but all objects have to be placed in one of multiple copies of the printing space, and the objective is to minimize the number of copies used. Formally:

$$\min_{M=\bigcup^{k} S_i, P_i} \quad k$$

Where S_i is the i-th printing run, and P_i is a valid placement for the objects in S_i

It is also worth mentioning that being able to solve the maximum filling problem with arbitrary V value function, allows us to use it as a subroutine in the column generation LP-solving algorithm, to solve the minimum number of printing runs problem

2.2 Solutions for the packing problem

Boxpacking
Packing problems are generally hard. For example even the one dimensional version (the knapsack problem) is NP-hard. The complexity of mesh objects increase the difficulty of the packing problem, so for the following sections, we replace the meshes with their bounding boxes. If we can pack these boxes so that they don't intersect, it means that placing corresponding original meshes in the corresponding positions will also be a valid placement. The bounding box is the smallest axis aligned rectangle containing every vertex.
This means that we have to solve the well known boxpacking problem. The two main types of algorithms are heuristic algorithms, and algorithms that try to give an exact solution the the optimization problem. We later in this chapter present an example for each. For solving this problem Sweep presents methods reduces the problem to one dimensional bin packing, by slicing the bin along two axes [8]. T. G. Crainic, G. Perboli and R. Tadei define extremal points on which to place the boxes, and derive heuristics based on an extremal point rule used to solve the 3D binpacking problem [9].

When packing boxes in a rectangular space its very rare that the optimal solutions has any boxes non-parallel to the sides of the outer rectangle. This means that we can ignore every rotation other than 90°. If rotation is allowed, we can try to find the orientation of the meshes, which minimize the size of their bounding box. Allowing

rotation of 90° is often accounted for by having each box twice in the input, once in each possible orientation, and allowing only one of each to be placed.

The IP algorithm for boxpacking

One of the possible approaches to solve the packing problem is to leverage the efficiency provided IP solvers. The solution of LP and IP problems is a central area of operations research, and thus there is much to be gained by exploiting the time and energy spent on developing and perfecting these solvers.

In our case the packing problem is NP-complete, we cannot hope for a fast general algorithm. We can give an IP formulation of the problem, and our hope is that the solver, using expert crafted heuristics, will be able to provide a good solution in most use cases.

In the following we present the IP formulation

IP formulation

In general the smaller the problem, the more chance we have that it is solvable in a reasonable time, the size of the problem is usually measured in the number of constraints and in the number of variables. To allow us to have small problem size, in this formulation we compute the bounding box, for each mesh. We replace every mesh with its bounding box.

The following is the IP formulation.

Let sx_i, sy_i, sz_i be the sidelenghts of the i-th box, and c_i the size.

x_i, y_i, z_i are going to represent the corner of the i-th box with smallest coordinates.

Two corners of the bounding rectangle are $(0, 0, 0)$ and (ax, ay, az).

Parameters:

$\forall i : sx_i, sy_i, sz_i, c_i$

ax, ay, az

Variables:

$\forall i : x_i \in [0, ax - sx_i],\ y_i \in [0, ay - sy_i],\ z_i \in [0, az - sz_i]$

$\forall i < j : left_{ij}, right_{ij}, up_{ij}, down_{ij}, front_{ij}, back_{ij} \in \{0, 1\}$

$\forall i : used_i \in \{0, 1\}$

The optimization problem:

$$
\begin{aligned}
max \quad & c * used \\
s.t \quad & x_i + sx_i \leq x_j + right_{ij} * ax, && \forall i < j \\
& x_j + sx_j \leq x_i + left_{ij} * ax, && \forall i < j \\
& y_i + sy_i \leq y_j + up_{ij} * ay, && \forall i < j \\
& y_j + sy_j \leq y_i + down_{ij} * ay, && \forall i < j \\
& z_i + sz_i \leq z_j + front_{ij} * az, && \forall i < j \\
& z_j + sz_j \leq z_i + back_{ij} * az, && \forall i < j \\
& right_{ij} + left_{ij} + up_{ij} + down_{ij} + front_{ij} + back_{ij} \leq 7 - used_i - used_j && \forall i < j
\end{aligned}
$$

The *used* decision variables represent whether each object is used, the $right, left, up$, etc. decision variables represent which direction each pair of object is from the other, $right_{ij} = 0$ meaning the j-th object is right from the i-th 1 meaning it doesn't have to be. When either $used_i$ or $used_j$ is 0 than the last equation trivially holds, otherwise it forces one direction variable to be 0.

We can see that this formulation has $4n + 6\binom{n}{2}$ variables, and $7\binom{n}{2}$ constraints.

This formulation solves the *maximum filling problem*, but we can give a similar formulation for the *minimum number of runs* proble.

Local search using IP

When trying to solve IP complete problems it is common to use well known heuristic approaches like local search. In this section we propose a local search algorithm based on the IP formulation. In this local search framework the graph nodes are the solutions, and the edges i.e. the closeness is defined by requiring the two solutions to have similar objects being placed, and similar relative positions of objects. This can be expressed as constraints in the IP description.

We will use two types of constraint.

Used distance: This determines how different can two neighbouring solutions be in terms of objects used.

$$\|used_{new} - used_{old}\|_1 \leq used\_distance$$

This can be expressed as a single linear constraint.

Direction distance: This is a constraint on changes of the $right, left, up...$ vectors.

$$
\begin{aligned}
\|right_{new} - right_{old}\|_1 + \|left_{new} - left_{old}\|_1 + \\
\|up_{new} - up_{old}|_1 + \|down_{new} - down_{old}\|_1 + \\
\|front_{new} - used_{old}\|_1 + \|used_{new} - used_{old}\|_1 \leq direction\_distance
\end{aligned}
$$

Direction distance is only understood to include the objects that are used. This is done because if we take a used and a non-used object pair, all of corresponding direction variables are 1. So placing an object means we have to update at least one direction variable for each object already used.
Although adding these constraints increases the size of the problem, we can still expect speedup when using a good IP solver, because the solution space is greatly reduced.

2.3 Coverings with boxes

The main drawback of replacing the objects with bounding boxes is, that some shapes are not suited well for this, and waste a lot of space. Keeping in mind that we are using IP formulations, the best way to try to remedy this problem is to cover the mesh with not one, but multiple (2-3-4) rectangles. We have to try to keep the number of rectangles low in order to not increase the size of the IP too much. We can then easily modify the IP to handle this by requiring the rectangles to be in a fixed relative position if they belong to the same object. If we have 2 rectangles covering a mesh, and their corners are (x_1, y_1, z_1) and (x_2, y_2, z_2), and and in the covering these are offset by the $(1, 2, 3)$ vector, then we can add to the IP

$$x_1 + 1 = x_2, y_1 + 2 = y_2, z_1 + 3 = z_2$$

Or we can even find all occurrences of x_2, y_2, z_2, and replace them accordingly.
Finding these coverings is non trivial, in the corresponding literature we could find the following results.
Bespamyatnikh and Segal [1] propose an algorithm for finding two axis parallel boxes covering a set of points. Their algorithm runs in $O(n \log n + n^{d-1})$ time in d-dimensions, and minimizes the area of the larger box.
In [2] Saha and Das give a similar algorithm in the plane, where the boxes orientations are arbitrary but parallel to each other. This algorithm runs in $O(n^3)$ time.
In our case the meshes can contain large numbers of points so the second algorithm might be too slow. The first algorithm works in $O(n^2)$ time in 3D, so it might be useful, when we restrict ourselves to find exactly two boxes. It is also not clear whether minimizing the volume of the larger box or minimizing the area of the union of the boxes is more beneficial. For this reason we propose heuristic algorithms to find suitable coverings.

Branch and cut algorithm

We propose an algorithm for efficiently enumerating all possible sets boxes. The number of possibilities can be very large, so we begin by overlaying a rectangular grid over the mesh and replacing the mesh by the union of cells that intersect the mesh. This union contains the mesh inside it. This is done so that by choosing the fineness of the grid we can tune the complexity of the task.

B-C algorithm

INPUT: Set of grid cells to be covered, n: the number of boxes to be used
OUTPUT: n boxes covering the cells
Branch-and-Cut$(n, cells, volume\_so\_far)$
  if $n = 0$ **then**
    if $HigherBound > volume\_so\_far$ **then**
      $HigherBound = volume\_so\_far$
      save solution
      return
    end if
  else
    $min\_volume = volume\_so\_far + V(cells)$
    if $min\_area \geq HigherBound$ **then**
      CUT
    end if
    for all box **in** $boxes\_to\_try$ **do**
      $cells' := cells \setminus box$
      Branch-and-Cut $(n-1, cells', volume\_so\_far + V(box))$
      if $min\_area \geq HigherBound$ **then**
        CUT
      end if
    end for
  end if
end

This type of algorithm is reliant on good hand crafted heuristics, because otherwise we have to check a high number of possible sets of boxes. Here we have two heuristic functions in the algorithm. First is $boxes_to_try$. By only trying boxes that are likely to be part of a good covering we reduce the running time greatly, and by ordering the boxes from "best" to "worst" based on some heuristic, we increase the number

of cuts we have to do by lowering the $HigherBound$. Second is to calculate the min_volume lower bound by using a better estimate. For example using the algorithm from [1] as a subroutine when $n = 2$.

Adaptation of K-means

Our next proposed algorithm is based on the fact that we can formulate the covering problem as finding a partition or clustering so that when covering each cluster with a box, the total area of the boxes is small. We will adapt the basic idea of the K-means clustering. The framework of the K-means algorithm is the following.

Clustering algorithm
Initialize the clusters
while the clusters are changing **do**
(1) Recalculate the centers
(2) Assign every point to a center
end while
Return the clusters

We only need to define (1) and (2).

In our algorithm the centers will not only be a point in space, but a point-direction pair, where the direction represents the direction of the diagonal of the box, the box centered at the point. This allows us to have elongated or flat boxes adjusting to the shape we are trying to cover. With this we can define the two steps

(1): Having a partitioning of the points we find the axis parallel bounding box of every partition, and save its center and the direction of its diagonal.

(2): Having the center diagonal pairs we calculate for every point p and center+direction (c, d) pair, what is the smallest volume box centered at c having diagonal direction d that contains p We then assign every point, to the center, where this volume is the smallest.

The main advantage of this algorithm is that it is very fast. We can even improve the quality of the solution by generating random initial clusters, running the algorithm for each, and choosing the best of the outputs. We hope to get a solution where the covering boxes align well with the shape of the input.

If rotation of the objects is allowed, we can modify step (1) by finding the optimal orientation which minimises the area of the union of bounding boxes, where the boxes have that orientation. This can be done by calculating the convex hull of each partition, trying all orientations parallel to one of the faces of the convex hull, and choosing the best. If only rotation around a vertical axis is allowed, then we have to project the points to the horizontal plane, and do the same.

2.4 A heuristic algorithm for boxpacking

So far we have only seen packing algorithms based on the IP formulation, but often the best algorithms for NP-complete problems are based on intuition, and heuristic approaches exploiting the nature and characteristics of the problem. Meta heuristic techniques are, methods that can be used with different underlying heuristics and can solve different problems. We well present a Tabu Search algorithm for boxpacking decribed by A. Lodi, S. Martello & D. Vigo in [3].
In this section we solve the *Minimum number of runs problem*
Given a set of objects, and a rectangle called bin, what is the smallest number of bins that can hold all objects inside them, so that the objects don't intersect.
Same as in the IP formulation, the objects are replaced with their bounding boxes, .
First we define a simple algorithm, which can solve this problem in a very short run time. This algorithm is based on the intuition that the total area of objects touching each other and the walls of the bins are large when the objects are packed tightly. We will refer to this as the sub-heuristic.

Sub-Heuristic

```
Sort the boxes in decreasing order by volume.
for B in boxes do
    Generate admissible positions P for B.
    if P = ∅ then
        We have to use a new empty bin.
        Put B in the bottom-left corner of a new bin.
    else
        for p in P do
            Calculate the sum of all areas where B in position p is touching
            already placed objects and walls.
        end for
        Place B in position P
    end if
end for
return the number of bins used
```

We define a position p admissible for box B, if B placed at position p doesn't intersect other boxes, and is inside the bin. And B has a wall or objects touching the bottom, left, and the back sides of it. This is done so that the bins start to fill up from one corner.

This algorithm in itself can provide a solution to the problem, but we will use it to provide a Tabu Search meta heuristic algorithm.

Tabu search is a variant of local search. We define the usual local search graph, where edges are moves that transform one solution into an other. We will allow moves that don't improve the quality of the solution. But we use a Tabu list in order to avoid going in circles on the local search graph. The Tabu list forbids some non-improving moves based on previous moves made.

Let $A(S)$ denote a call of the Sub-heuristic algorithm, and its return value: the number of bins used. Moves in the local search procedure will consist of removing objects one-by-one from a target bin, by repacking all objects from k other bins and exactly one object from the target bin. This means that k defines the size of the neighbourhood of a node. We will modify the value of k throughout the procedure, as smaller k lets us make progress faster, and larger k allows us to not get stuck in a local optimum. We set a maximum value for k: $k_{max} = 2$ or 3 in practice. We'll chose the target bin as the least full bin by volume, breaking ties by the number of objects in the bins. This action is called Search in the algorithm and it updates the value of k.

The initialization step consists of placing every object in a separate bin. The algorithm runs until a predetermined time limit is not reached.

Tabu search Algorithm

Tabusearch()

pack each object in a separate bin
while time limit is not reached **do**
determine target bin t
while $diversify$=false **do**
Search$(t, k, diversify, movemade)$
if $movemade$ **then**
determine new target bin
end if
end while
Diversify()
end while
end
Search$(t, k, diversify, movemade)$
$penalty^* = \infty$
for each object o in target
for each k-tuple K of other bins
$S := o \cup K$
case
$A(S) < k$ or ($A(S) = k$ and o is the last object):
remove o from target bin, pack $K \cup o$ according to $A(S)$
$k := max(1, k - 1)$
$movemade = true$
$A(S) = k$ and o isn't the last object:
remove o from target, pack $K \cup p$ according to $A(S)$
$movemade = true$
$A(S) = k + 1$:
compute $penalty$
if $penalty$ is not tabu **then**
$penalty^* = min(penalty, penalty^*)$
end if
end case
end for

end for
if $penalty^* \neq \infty$ **then**
make move corresponding to $penalty^*$
$movemade = true$
else
if $k < k_{max}$ **then** $k = k + 1$ **else** $diversify = true$
end if
end

We need to define how to compute the penalty, and the diversification action. As we can see, penalty is given exactly if the move increases the number of bins used. In this case we require, that the objects in the target bin t without the target object o, and the objects in the least full of the new bins can be packed into a single bin. Notice that this would easily be true, if the new packing had a bin with only o in it, which can be achieved by leaving the other bins K as they were. So we can see that this requirement is natural. If this requirement is not met, then the *penalty* is infinity, otherwise the penalty is the filling function of the least full new bin. Where the filling function is defined as the ratio of total volume of objects in the bin divided by the volume of the bin.
Diversification needs to happen when we cannot make a move. This means that we cannot remove any object from the target bin. In this case we choose the next empties bin. This is governed by variable d. At the d-th diversification we set the target to be the bin with $d + 1$-th smallest filling function. We also have a preset limit d_{max}, after d_{max} diversifications, we make a *strong diversification* action. This means we order the bins in increasing order by filling function, and we pack all the objects from the first half of the bins into separate new bins, each bin having a single object. This is done in order to get out of a region of the local search graph the algorithm has explored thoroughly, but from which the algorithm can't move out.

2.5 Mesh simplification, and collision based on nearly convex decomposition

In the following two sections we propose ways of using the meshes more directly. So far we have used bounding boxes, which can be too coarse to allow the objects to fit very close together. If there is only small separation required, and different shapes of the objects fit together well, then bounding volumes which cover the objects well, are needed to obtain a space efficient packing. Or as we will see later, we can even

use the meshes directly.
One potential bounding volume can be built by partitioning the mesh into nearly convex pieces, and taking the union of the convex hulls of these pieces. Nearly convex here means, that the convex hull of the piece is close to the piece itself.
In this section we present an algorithm for finding such decomposition, and a mesh simplification algorithm which can be used in conjunction with the convex decomposition, and also can be used to make dealing with large, i.e. very fine, meshes easier.

Nearly convex decomposition

It is intuitively clear that if we want to solve the packing problem, we should at least be able to detect if two objects collide. We will see later how this can be used to build a packing algorithm.
A common way to provide collision detection or distance measurement, is based on convex decomposition. This is because, deciding if two convex polyhedra intersect is an LP problem, and can be solved quickly. The other advantage of convex decomposition is that we can "inflate" the objects easily, thereby forcing them to have certain separation between them. This can be useful for us to help support structures fit between objects, and to make handling the objects after printing easier. This can be done in the convex case by scaling up the object from the center. Scaling up concave objects would not have the property of containing the original.
One such algorithm is given by Mamou and Ghorbel in [7]. Their algorithm segments the mesh into partitions, based on a bottom-up approach, where they start with every face in a different partition, and unite partitions based a concavity measure, until it exceeds an arbitrary bound. There is no universally used concavity measure, here it is defined as the longest distance between any point of the object and the convex hull.

To define the algorithm we first define the dual graph of the mesh. This has faces as the nodes, and edges between every pair of faces sharing an edge. We will perform edge contractions on this graph.
After some edge contractions are performed, we denote by $A(v)$ the ancestors of v. Formally:
Initially $\forall v : A(v) = v$
After performing edge contraction uv, and denoting the resulting node by v'
$A(v') = A(u) \cup A(v)$
We denote by $S(u, v)$ the union of the faces corresponding to $A(v)$ and $A(V)$. $S(u, v)$ represents the object we get by taking union of the objects corresponding to u and v.

Now we need to define a cost function to decide which edge to contract. The two goals we have is to 1: have compact clusters, and 2: have low concavity measure. Having compact clusters, where the faces in the cluster form a shape close to a disk, is important at the beginning because when the clusters only contain a small number of faces the concavity measure is small, and elongated clusters would make for clusters that are harder to unite.
The cost function has two terms corresponding to the two goals above.

$$E(v,w) = \frac{C(S(v,w))}{D} + \alpha E_{shape}(v,w)$$

where

$$E_{shape}(v,w) = \frac{\rho^2(S(v,w))}{4\pi\sigma(S(v,w))}$$

$\rho(S(v,w)$ and $\sigma(S(v,w))$ are respectively the perimeter $(S(v,w))$. $E_{shape}(v,w)$ is the aspect ratio, it is smaller, the close the object is to a disk, it is equal to 1 if the shape is a disk.

D is a normalization factor equal to the length of the diagonal of the bounding box of $S(v,w)$.
Parameter α is chosen in a way that, E_{shape} has no influence on the last few contractions. The shape penalty should only have an influence when the concavity measure is very small.

$$\alpha = \frac{\epsilon}{10D}$$

where ϵ is the bound we set on the concavity measure.

The concavity measure is defined as

$$C(S) = \max_{x \in S} \|x - P(x)\|$$

where $P(x)$ denotes the projection to the convex hull.

In each iteration the algorithm chooses the edge with the smallest penalty, performs the associated contraction, and recalculates the penalties as necessary. This is repeated until the concavity measure C is below an ϵ threshold.

The two algorithms in this section can be combined well, by first finding a bounding mesh, and then calculating the convex decomposition of it. This is needed because otherwise the convex decomposition algorithm would be too slow.

The resulting object can be used to quickly and efficiently check collisions and distance.

Mesh simplification

Meshes have a wide range in terms of how much detail they contain. Very fine meshes can have huge numbers of triangles, that in our application has drawbacks, as it is not needed for the meshes to be placed, but working with large objects naturally slows down most algorithms. Large file sizes can be a result of exporting from a CAD program, where the user can set the smoothness of the triangulation. If we don't have the original CAD files, we have to use mesh simplification to control the number of triangles.

Most mesh simplification algorithms use a heuristic edge contraction method, with no guarantee about the resulting mesh. In this section we present an algorithm by A. Gaschler, Q. Fischer, and A. Knoll [6], which can produce a mesh that contains original. This is crucial in our application as we need this property to guarantee that the objects don't intersect.

The algorithm iteratively contracts edges, but in a way that the new mesh contains the original inside it. It does this by computing a penalty. For given edge e replaced by vertex v the penalty is.

$$E(e, v) = \sum_{p \in P(e)} d^2(p, v) = v^T Q(e) v$$

Where $P(e)$ denotes the faces adjacent to edge e. This approximates the Hausdorff distance of the new and the old mesh. $E(e, v^*)$ is defined by finding v^* minimizing the above expression while having the property, that replacing e with v^* results in the new mesh containing the old.

Parameter ϵ controls the maximum distance.

Mesh Simplification Algorithm

for every edge **do**
    Compute $Q(e)$
    Minimize $E(e, v^*)$
end for
while $E(e, v^*) < \epsilon$ **do**
    Pop best edge e
    Remove edge e and adjacent triangles
    Insert new vertex v^* and triangles
    Recalculate costs of changed edges
end while

This algorithm can be used if needed in conjunction with the following collision detection algorithm, as smaller size leads to smaller data structures, which can lead to quicker operations.

2.6 Collision detection for packing meshes

In this section we describe an algorithm which uses the mesh directly, and doesn't replace it with boxes, or other bounding volumes. This is useful when objects have shapes that fit well with each other, and most approximation would not have this property.
Compared to the LP based collision detection of convex polyhedra, this approach can be often much quicker. We will use a hierarchical structure which provides fast queries, and often requires very small number of operations, compared to the roughly linear running time of the LP algorithms.

When using simple descriptions of an object we can use more sophisticated and more efficient algorithms, because we have more available operations we can do on them, in a feasible time. This simple description could be a function like signed distance field (SDF), or some simple mathematical object like constructive solid geometry (CSG). Converting the mesh to one of these is possible but often unpractical, because the size of the model would become too large.
Using meshes doesn't allow us to implement very efficient operations. In this section we present a way to test if two meshes collide. Collision detection is one of the simplest operations, which still allows us to pack objects. This method of collision detection is described in [4] by S. Gottschalk, M. C. Liny, and D. Manocha.
In addition to the contents of [4], we show how to use the basic idea of the article, to translate and rotate meshes without intersection.
Our idea is that if we can detect when two objects collide, then we can place objects in space with no two objects colliding with each other, and we can maintain this property. We can then try to move objects in small steps so that they fit in a smaller volume of space.

Collision detection on meshes
Here we describe the collision detection method, based on a hierarchical data structure. We will have two meshes as input. The goal is to quickly decide if the two objects are intersecting or not. We will build a data structure for each mesh, to

allow us to later quickly query collisions multiple times, with the meshes moving in space between queries. Building these data structures can be more computationally intensive as we only need to do it once for every object.
This method is based on simple ideas. First observation is that if we calculate the bounding box of the two objects then we can quickly decide, if the two boxes are intersecting, and if they aren't, then we also know that the two objects are disjoint as well. Secondly we remark that checking every pair of triangles from the two meshes would take two long. So we would like to use the idea of bounding boxes to reduce the number of triangles necessary to check, to guarantee the two meshes are disjoint. The structure we use is a binary tree, where each node of the tree has a set of triangles, and the bounding box of those triangles assigned to it. The root has the set of all triangles assigned, the leafs have singles triangles assigned, so that every triangle of the mesh is assigned to exactly one node. For every non-leaf node of the tree, the two sets of triangles assigned to the two child nodes form a 2-partition of the set assigned to the parent.

$$S = \{\text{set of triangles}\}, T = (V, A) \ \text{(tree)}$$
$$r \in V \ \text{(root)}, f : V \to 2^S \ \text{(assignment)}$$
$$f(r) = S,$$
$$\forall s \in S \exists! v \in V : f(v) = \{s\}, deg(v) = 0$$
$$\forall (vx) \in A, (vy) \in A : f(v) = f(x) \cup f(y)$$

We use oriented bounding boxes (OBB) as the bounding volumes, this means the rectangles can have any orientation. It is worth mentioning that, there are other possible choices. These include axis aligned bounding boxes, ellipsoids, spheres. We choose oriented bounding boxes because they are easy to compute, easy to check collision between them, and can still cover different shapes well in practice. Our hope is that in the tree hierarchy after a couple of steps from the root, each node will have small area of continuous surface assigned. These can often be well covered with a flat rectangle, with shortest axis of the rectangle perpendicular to the surface.

Building the OBB Tree

It remains to describe how to build the tree. for this we only need to define and compute an approximate "orientation" of a set of triangles. We do this by thinking of the set of points of the union of the triangles as a uniform distribution over the surface defined by the triangles. We then calculate the mean and covariance matrix of this distribution. Two of the three eigenvectors of this matrix is the axes

of minimum and maximum variance, so often they align well with an elongated or a flat object. We will use these eigenvectors as the axes of the bounding box by finding the extremal points along these directions. If the vertices of the i-th triangle are the points p^i, q^i, and r^i, the mean and the covariance matrix can be expressed as:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} m^i \frac{p^i + q^i + r^i}{3}$$

$$C_{jk} = \frac{1}{24n}\sum_{i=1}^{n} m^i[(\bar{p}_j^i + \bar{q}_j^i + \bar{r}_j^i)(\bar{p}_k^i + \bar{q}_k^i + \bar{r}_k^i) + \bar{p}_j^i\bar{p}_k^i + \bar{q}_j^i\bar{q}_k^i + \bar{r}_j^i\bar{r}_k^i]$$

where $\bar{p}^i = p^i - \mu$, $\bar{q}^i = q^i - \mu$, and $\bar{r}^i = r^i - \mu$, and m^i is the area of the i-th triangle. Now we only need a method to construct the tree structure. We choose a top-down approach. Beginning with the bounding box at the root of the tree, we cut it in two pieces along its longest axis. We choose the mean of the vertices as the division point. We partition the triangles in two groups based on which of the two parts they are in (choosing one if they intersect both pieces of the box). Then we use the above method to calculate the new bounding boxes of the two sets. At every depth, we do this recursively until the leaf nodes have only one triangle assigned to them.
About checking collision of two boxes and the special cases when building the tree we refer the reader to the original paper [4].

Collision Detection with OBBs

Given two meshes, and their OBBTrees we can detect collision in the following way.

Algorithm for collision detection

Intersect(A, B)

if $BoundingBox(A) \cap BoundingBox(B) = \emptyset$ **then**

return false

else

return

$Intersect(r\text{-}child(A), r\text{-}child(B) \vee Intersect(r\text{-}child(A), l\text{-}child(B) \vee$

$Intersect(l\text{-}child(A), r\text{-}child(B) \vee Intersect(l\text{-}child(A), l\text{-}child(B)$

end if

end

Where $l\text{-}child(A)$ is the left child of the node A. When A or B is a leaf node we can check for intersection easily. Since we start by checking for intersection of the bounding boxes, often we don't have to check many nodes of the trees before terminating.

Notice that if the two meshes as surfaces are not intersecting but one contains the other as 3D objects, this function will return false. This case can be avoided by starting out with non-intersecting meshes, and maintaining this property. Or we can use a similar recursive algorithm for checking if a point p is inside the mesh. It is simple to generate a point q that is outside the mesh, for example one of the corners of the bounding box (if its not a point of the mesh). We then can count the number of triangles intersecting the line segment pq, if the count odd p is in, if it's even p is out. We can find all triangles intersecting the line segment by recursively checking if the bounding boxes intersect the line segment.

Directional distance measurement using OBBTree

Since we want to use the above approach to move objects, it would be beneficial to have an algorithm to decide how much can we continuously move an object by, so that it still doesn't intersect a given object. In this section we will show how to solve the following problem. Given mesh A, and mesh B both placed in 3D space, and direction v: what is the most distance we can move A in direction v without hitting B.

Algorithm for Measuring Distance

Global upper bound variable UB given in the input
or initialized to ∞
The direction v is also given in the Input, and used
by the $dist()$ function.
$Distance(A, B)$

if A,B are leaf nodes **then** $UB = min(UB, dist(A, B))$, return $dist(A, B)$
if $dist(BB(A), BB(B)) \geq UB$ **then**
return ∞
else
return $min($
$Distance(r\text{-}child(A), r\text{-}child(B), Distance(r\text{-}child(A), l\text{-}child(B),$
$Distance(l\text{-}child(A), r\text{-}child(B), Distance(l\text{-}child(A), l\text{-}child(B))$
end if
end

Where the $dist$ function, given a direction v, can compute how far we can push a rectangle or triangle in the direction given, before hitting another rectangle or triangle. It returns ∞ if the to objects don't hit each other. We use a global variable UB for updating an upper bound on the distance. We hope to quickly cut of parts

of the search tree where the two objects go past each other when moving in the v direction. But also when we find triangle pairs which do meet when pushed, we can then cut of any parts which are farther than the best pair of triangles we have found so far. For this latter concept to work best, we have to try to find parts of the OBBTrees which are closer as early as possible. To do this we modify the algorithm:

Modified Algorithm

Distance(A, B)

if A,B are leaf nodes **then** $UB = min(UB, dist(A, B))$, return d

$pairs :=$

$\{(r\text{-}child(A), r\text{-}child(B))$, $(r\text{-}child(A), r\text{-}child(B))$

$(r\text{-}child(A), r\text{-}child(B))$, $(r\text{-}child(A), r\text{-}child(B))\}$

for (a,b) **in** pairs:

calculate $dist(BB(a), BB(b))$

delete the pair if $dist \geq UB$

end for

order $pairs$ by $dist$

make the recursive calls in this order.

return the min

end

In this version we check to see which pairs of bounding boxes are closer, and go down the search tree accordingly. This means it is much more likely that we find a good upper bound early, and therefore increase the likelihood that we can cut off branches.

The Upper bound UB can also be set in the input. This is important because we use this algorithm for checking collisions for one object with all others, so if we know a maximum distance for one other object, we don't need to check objects farther than that, and can use that distance as the upper bound.

Given an axis and an object, we can compute the maximum continuous rotation around the axis, where the rotating object doesn't intersect any other object. Note that computing this rotation for pairs of boxes, and triangles can be done in constant time. This means we can use the above algorithm exactly, by replacing $dist()$ with the appropriate function. Here we see one of the advantages of OBB-s compared to axis aligned boxes, as rotation is allowed.

Bin packing based on collision detection

We propose a simple method for binpacking, based on the above algorithms. We

consider the same problem we discussed earlier, where we are given a rectangle called the bin and a set of objects given as meshes, and we have to pack the bin as efficiently as possible.

The algorithm makes use of a well known meta-heuristic approach called simulated annealing. We start by placing all objects in space, such that no two objects intersect. We will move the objects randomly, but over time we will reject more and more moves that move an object out of the bin. We can make a random move, by picking a random object, a random direction, and calculating the maximum distance d we can move the object without intersection. We then move the object a random amount that is smaller than d. Notice that checking an other object during the above calculation is often just a few operations, because we know they never intersect, based on the bounding boxes. We can also speed up this calculation by ordering the objects whose bounding boxes will intersect the box we are moving, by ordering these by the closeness of the bounding boxes.

Simulated Annealing Algorithm

parameter: mmd max move distance
for i=1 to n **do**
pick a random object o
pick a random direction v
calculate max distance md
$d = rnd[0, min(md, mmd)]$
calculate $\Delta E(move)$
if $exp(\frac{-\Delta E(move)}{T}) > rnd[0, 1]$ **then** make move **else** reject
update T
end for

Where $\Delta E(move) = V(objects)$ if the object leaves the bin, and 0 otherwise.

Calculate max distance means calculating the maximum distance o can be moved in direction v without hitting any other objects.

This version strictly follows the usual structure of the Simulated Annealing algorithm. But we often have to calculate the maximum distance we could move, just to reject the move anyway. This can slow down the algorithm, because calculating collisions is the most computationally expensive part, and when T is small, we reject most moves when the object leaves the bin. The following is a solution to this flaw. We choose a random object. We first calculate $exp(\frac{-V(objects)}{T}) > rnd[0, 1]$ if its $True$ then we calculate the move in the same way as above, and then make it, if it's

$False$, we calculate a move which keeps the chosen object in the bin and make it. This avoids the above problem. While its not the same algorithm, it is equivalent to choosing the same object and direction until we move it.
One additional heuristic idea is to increase the likelihood of choosing a direction that takes the object toward to bin, if the chosen object is outside. This helps pack the objects closer to the bin, and increases, the chance they will be packed. This behaviour can also be controlled by the temperature T. Additionally as T gets low, we should consider rejecting moves that result in the object not being fully inside the bin, as this reduces the space available in the bin, while in this case we can't include the object in the final packing.

The above algorithm ignores rotation, but if rotation is allowed, then we can include the rotation version of maximum distance subroutine, and modify the Simulated Annealing algorithm by picking a random rotation instead of random translation with some probability.

3 Support Planning

3.1 Introduction of support planning

In this chapter we will discuss what is the support planning problem we have to solve. We formalize the problem, and create a setting where we can provide solutions. We then provide algorithms to solve the optimization problems, and finally we show how to generate output that can be used to print the objects and the supports with a 3D printer.

In this discussion we are considering the challenges posed by the metal 3D printing process. Generally when working with metals it is much harder to remove the support structures after printing than with plastics. So we aim to connect all supports to the ground, which is the bottom side of the rectangular printing volume. This allows us to cut all objects and supports of the base at the same time, and we only need to deal with the surfaces that required supporting. A typical way of planning support structures is to just build vertical structures from the places that need support, until we hit an object or the ground. This is practical, if we use different materials for the support and the objects so that the support can later be dissolved. But in our case this is not possible, and this above approach of support planning can lead to supports which cannot be removed practically.

Additionally we have to consider the stresses caused by the high temperature of the laser melting process. This means that we have to provide adequate support at possibly multiple areas of the object. In practice we have to support every face of the mesh, which faces downward, if the angle between the face and the horizontal plane is less than a threshold, for example 45°. If this requirement is not met, the material can warp from the huge temperature difference between the place of melting, and the already cooled parts.

In practice this is usually enough but sometimes we have to also check this threshold for edges. It possible that two valid faces meet at an edge which need support. An example for this can be a knife edge facing downwards, both faces forming the edge is nearly vertical thus not needing support, but the edge does need support otherwise we would have a floating object.

3.2 Problem formulation

Like in the previous chapter, we will work with triangle meshes, but unlike before, now their position in space is fixed. In other words we are given some objects placed in space. We are also given a printing volume, which is a box containing all objects inside it, and a threshold angle, which determines what faces need support.
In order to have a well defined solution space we place the objects in a rectangle grid. The grid has to be aligned with the ground, so that the ground is one of the grid planes. We will call the distance between the planes of the grid one step, and the horizontal planes as layers. The supports will follow the grid in the following way. They start from the ground and each section of support will connect a horizontal grid square with one, one step up and possibly one step sideways. This "at most one step sideways" rule encapsulates the 45° angle threshold. If the threshold is different, we can adjust the step between layers accordingly. This approach also allows increase or decrease in the fineness of the grid to allow better support structures or have better running time.
After we have chosen our grid, we color the grid cubes, that intersect any object black. The rest of the grid cubes are colored white. We also color the bottom sides of all black cubes containing triangles needing support red, but only if the cube below is white. We also color all squares of the ground, not having a black cube on top, green.
We have to connect all red squares to green squares at the bottom, avoiding all black squares.

Defining the support graph

To formalize the support planning process, we define a graph. The nodes of the graph are all the horizontal grind squares. We draw an edge between two squares if they are one layer apart, and the lower square is, or is orthogonally adjacent to, the square exactly below the upper square. And the (potentially oblique) prism connecting the two squares doesn't intersect any black cubes. The nodes inherit the color of the corresponding square.
It must be noted that finding a path from a red node to any green node might be impossible. In this case we connect everything we can, and report the unconnected nodes in the output. It can also happen that a path exists but we cannot find it because the grid is not fine enough.
Connecting the red nodes to the green nodes is a simple pathfinding task, but if we take the union of these paths and use that as a support structure, it is often

inadequate. It can happen that the many paths join, and use the same single edge. This can be insufficient to conduct heat away, may not be stable enough when we are trying to remove the supports after the printing process. This leads us naturally to the idea of having capacity on the edges. Capacity of one would mean no path can join, but if we choose the capacity carefully we can encourage the paths to join, and still maintain the structural requirements. This chapter will mainly focus on the algorithms used to solve this flow planning problem.
The graph we defined above can potentially be large, so we delete any nodes that are unnecessary. These can be found by deleting all nodes that cannot be reached from any green node, and all nodes that cannot be reached from any red node. We also use a Digraph instead of a graph by orienting all edges downward, as it is unnecessary to have draw the edges in both directions.

In our terminology the black cubes with red bottom squares are the ones that contain faces of the object needing support, so far we have laid out a plan to build up support from the ground to the red square. This means we have a finally step, no matter what algorithm we use, which is to connect the end of the support (the red square) to the faces needing support. This is done by just building a simple vertical structure. The exact way of connecting is outside the scope of this discussion, and has to be designed based on the material used, the way the supports are removed after printing, and other engineering considerations. To be able to simply project the faces onto the red square below, we need to have no parts of the object between the face and the bottom of the cube its in. This is achieved by choosing a suitably fine grid.
In the final section of this chapter we show how we can turn the flows produced by the algorithms into meshes ready to be printed.

3.3 Flow algorithms

The main part of our support planning algorithm is calling an already implemented flow algorithm. First we build the graph defined above. Then we add a source node s, and a sink node t. We draw an edge from s to every red node, and from every green node to t. If the maximum flow value is equal to the degree of s then it is possible to connect every red node.
Usually we set the edge capacity to between one and ten, but a fixed edge capacity can mean that there are red nodes that can be connected, but aren't in the maximum flow. It is preferable to exceed the edge capacity compared to having unsupported triangles. We will address this where it is possible.

It can also happen that the **edge** capacity constraint is not violated, but there are **nodes** of the graph where too many paths meet. This can also lead to insufficient structure. To avoid this we can add node capacity constraint. The standard way to do this is to double all nodes and add an edge with the node capacity between them.

Push-relabel algorithm

First presented in [5], the Push-relabel maximum flow algorithm has become a standard way of finding maximum flows. Its simplicity and quick runtime of $O(|V|^3)$ makes it a good candidate for us to use.

The main drawback of using a max-flow algorithm is that different max-flow solutions can have very different structural characteristics. Our main concern is the fact that there is no difference between vertical and oblique (edges that go one step down and one step sideways) supports, from the viewpoint of maximum flow. Slanted support structures can waste material based on what is structure we use, and these supports are less stable.

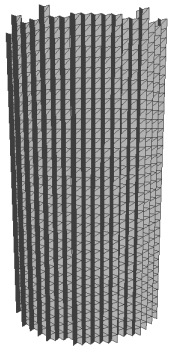
Slightly modifying the Push-relabel algorithm, can help overcome this effect with no significant slowdown of the algorithm. During the Push-relabel algorithm we sometimes have a choice of which edge to choose in the "push" step. The modification is to always use vertical edges if we have a choice of which edge to increase the flow value on, and never choose a vertical edge if we have a choice of which edge to decrease the flow value on.

This small change can produce drastically different results. It is clear that this is only a heuristic, but to do better, we would have to use a min-cost flow algorithm which are usually slower. When the graph is large this might be our best option to use.

The problem of supporting everything that can be supported, while trying to obey the capacity constraints can be addressed similarly easily. We delete all red nodes which don't have a path leading to the ground, these cannot be connected anyway. The we start running the max-flow Push-relabel algorithm, and when we would stop normally, which means we found a min-cut with every edge on maximum capacity, we increase the smallest of these capacities by one. We choose a vertical edge if possible to encourage the supports to be straight.

Min-cost flow algorithm It is clear from the above discussion, that maximizing the flow value is not perfectly aligned with our real goal, as we also prefer if the support stays as vertical as possible. The simplest and cleanest way to achieve this, is to assign a cost to every edge, vertical edges having smaller cost than slanted

Figure 3.1: Supports when no obstruction below



edges. We will then use a min-cost flow algorithm, for example network simplex.
Before calling the network simplex we calculate the maximum flow value with the Push-relabel algorithm, if we want to reach all possible red nodes we use the modification described above. After the flow value v is known, we can call the network simplex algorithm, in the input we have to specify that the supply at the source is v, at the sink its $-v$, and everywhere else is 0.

3.4 Shortest paths algorithm

The main motivation for trying further methods to solve the planning problem, is that the graph can be large if the grid is fine. So here we present an algorithm which can produce reasonably good results in a very short run time.
In addition to the min-cost flow algorithm being slow on large graphs, when possible the structure it generates is just a simple straight down line from the red nodes. This is usually good but if we have a large surface with no obstructions below it leads to an unnecessarily large support structure. Figure 3.1 shows the support for a ball "in the air". To avoid this we want some of the paths to "merge" together. To do this, we discount edges that have multiple paths using them, or in other words we want a cost function where having an edge with flow value of for example 4, is cheaper than sum of the costs 4 edges with flow value 1. This problem cannot be remedied with linear cost functions.
The idea is based on the fact that a flow can be thought of as union of paths. So we calculate shortest paths from every red node, to any green node. Where shortest means shortest with respect to a carefully crafted cost function. We use a cost function to both discourage slanted edges, and edges that have too many paths using them, and to encourage the paths to use the same edges to save material.

Let the number of red nodes be N

Algorithm for collision detection

Input: $g : \mathbb{N} \to \mathbb{R}$ cost based on flow value
Parameters: timelimit, $redraw\_ratio$

for all $e \in E$ set $c_0(e)$ to 1 for vertical edges, 2 otherwise
for all red node n **do**
    find and save shortest path from n to the ground using with respect to c_0
end for
for all $e \in E$ set $f(e)$ to be the number of paths using edge e,
while timelimit not reached **do**
    Select $N * redraw\_ratio$ number of random red nodes
    Update f to not include paths from the selected nodes
    for all $e \in E$ set $c(e) = g(f(e)) * c_0(e)$
    find and save the shortest paths for each selected nodes with respect to c
end while

The key idea of this algorithm is using the cost function g which lets us control the output of we get. $g(n)$ is the cost of increasing the amount of the flow on any edge by one, if the current flow amount is n. We can for example set g according to the following table.

n	0	1	2	3	4	5	6	7
$g(n)$	1	0	0	0	1	2	3	4

With this we can encourage the flow to use the same edges up to 4 times if the edge is already used, but after that we give bigger and bigger penalties to avoid too many paths using the same edge. This is advantageous because sometimes we want the supports to collect and use the same edges because this reduces material used.
We can also further encourage paths to go close to each other by giving a discount to $c(e)$ for edges that have edges with non zero $f(e')$ next to them. This can help in providing more stable support structures.
Giving vertical edges lower cost through c_0 allows us to have straighter supports.
Through the use of $redraw\_ratio$ we can control how many paths to redraw at the same time. Using higher value speeds up the convergence of the algorithm to an optimum, because in a single run of a shortest paths algorithm we can recalculate all shortest paths, bit it can lead to non-convergent behavior where the algorithm oscillates between two or more solutions.

It is easy to see that the graph is a DAG, because every edge is oriented downwards. This means, we can use the linear time DAG shortest paths algorithm. Choosing a small iteration number like 30, and $redraw\_ratio$ of 10%, allows this algorithm run quickly even on large graphs. And since we redraw all paths thrice on average, the algorithm finds a reasonably good solution.

3.5 Support structures

In this section we detail how the flow on the graph can be used to generate physical support structure.

As described in the beginning of the chapter, every node corresponds to a horizontal square and every edge corresponds to the prism connecting two horizontal squares. So the simplest way to generate a support from the flow is to take every edge where the flow is nonzero, and put a prism in the corresponding place. The collection of these prisms with the connecting procedure described earlier can be used to provide a support structure.

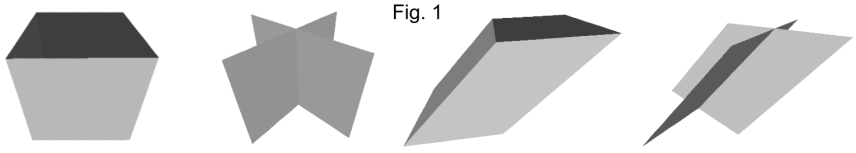
This however is impractical because the support structure would use too much material, and would be hard to remove after printing.

In order to design thinner, easier to remove and print supports, it is helpful to know that the printer can be instructed to interpret the mesh in two different ways. One interpretation is as a solid, the other is as a surface. We will use the latter in the following design.

We can replace each edge with an X shape shown in figure 1. This has multiple advantages, neighbouring supports connect to each other, thereby increasing stability. This structure has much less material, therefore it is cheaper and easier to remove. If the grid is fine enough, it is sufficient to extend the ends of the X shaped supports until they meet the object, to provide support to the faces.

One drawback is that the metal dust used during printing cannot be removed easily from between the supports, so to avoid this some layers we don't connect the neighbouring X's. Figure 2 shows the case where the X's touch at every third layer.

Fig. 1



Vertical and slanted support prism and support X

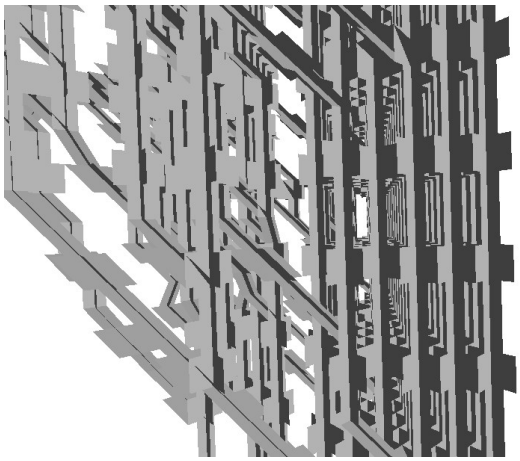
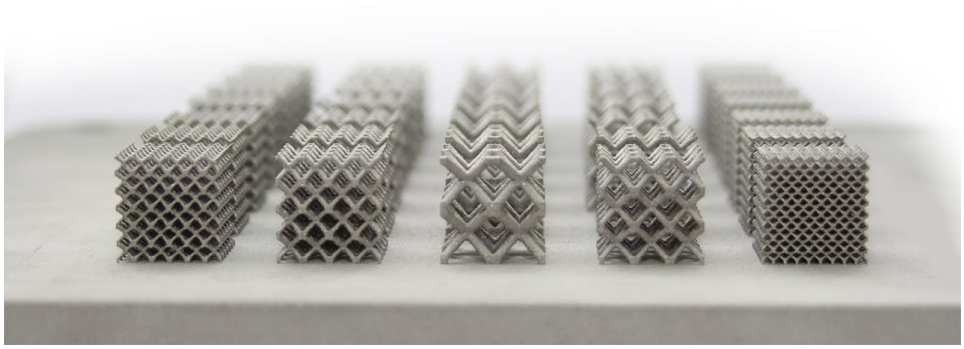


Fig. 2

The main advantage of using this graph to plan the support is that we can use any structure designed by an engineer. Like the lattice structures shown below. We can replace the prisms with that structure and its appropriate affine transforms.



4 Computational Experiments

One of our aims was to provide a framework, which allows a user to input some meshes, and get a packing with support structures, which is ready to be printed. In order to achieve this we implemented both packing and support planning algorithms. Having a variety of different algorithms, and parameters is beneficial, both because it allows us to compare them, and also because we can choose which ones to use if more information is available about the input we are expecting, and the requirements we have to meet.
In this chapter we present the runtimes, and performance of the algorithms we implemented.

Packing algorithms
We implemented the IP algorithm, and its local search variant, and the Tabu search algorithm. We present their performance on the *minimum number of runs* problem as Tabu search is built for this problem. We also present a comparison of two IP based methods. For this we use the maximum filling problem as it makes it easier to see small differences. For both of these we use a set of boxes with multiples of the same box imitating a real input. We use CPLEX optimizer for solving the IP problem. It must be noted that there can be huge differences between the performance of different optimizers.
The following table shows the time taken to find the optimal solution for increasing numbers of boxes. The last column shows a difficult, large input, where we write the best solution each algorithm produced within a 500 second time limit.

# Boxes	15	30	45	60	75	120
IP	0.06	0.7	2.76	6.99	33.1	29 runs
Tabu	0.04	0.06	0.08	0.14*	3.7*	27 runs

*We wouldn't have known, that we can stop with the optimal solution value, had we not solved the instance beforehand with the IP solver.
We can see, that the heuristic algorithm is faster, and the IP slows down with bigger inputs. One advantage of the IP is that we know, if the solution is optimal, and we get an upper bound on the gap between the current solution and the optimum.

For testing the IP algorithms we used a 120 second time limit. This is done because finding the optimum is a very hard problem, and can take a long time. Usually a well crafted IP solver can find good solutions early during its running, and improves these solutions gradually.
The local search algorithm has parameters governing the size of the neighbourhood, these are:
$used\_d, dir\_d$. We experimented with different setups, trying to keep the size of the neighbourhood big enough that it allows for improvement, but small enough so that the search space is not too large. We measure the runtime and the filling function.

	$used\_d$	dir_d	# Boxes=20	25	30	35
IP	X	X	76%, 10s	80%, 30s	84%, 120s	83%, 120s
Local search	2	30	76%, 11s	69%, 2.25s	83%, 94s	78%, 58s
Local search	5	20	76%, 8s	79%, 76s	83%, 90s	81%, 90s
Local search	8	10	76%, 5s	79%, 58s	83%, 61s	81%, 90s

It is clear that using the local search variant is not aimed at finding the optimal solution, but it can help finding a reasonably good solution when the input is big.

support algorithms

We also implemented all of the support planning algorithms detailed above. We present their runtimes on five different inputs. The inputs where selected to cover a range of sizes, and are generated from stl files as described in the previous section. Object B is just two copies of object A next to each other. Object C is two balls, one is above the other so that the supports have to go around the lower ball. Input E represents a packing where smaller objects fill the printing volume.

The size of the graphs are as follows:

	A	B	C	D	E
nodes	8196	27670	62708	1110380	6424582
edges	35674	126566	296235	5348590	31381816

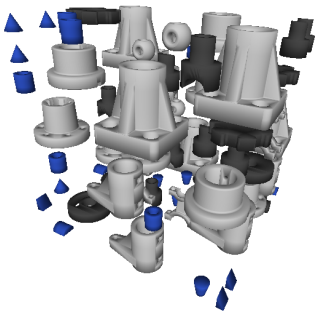
The runtimes in seconds:

	A	B	C	D	E
Push-relabel	0.002	0.007	0.027	0.664	6.75
Min-cost flow	0.017	0.089	0.133	12.1	186
Shortest paths	0.092	0.325	0.948	20.73	142

Figure 4.1: The input stl files

(a) Input A

(b) Input D



(c) Input E

From this we can conclude that the Push-relabel algorithm is the fastest, and that the Shortest paths algorithm is a viable replacement for the Min-cost flow algorithm when the input is large. The speed of the Push-relabel algorithm is offset by the fact, that out of the three it gives us the least control over the output.
It is important to mention that although the shortest paths algorithm gives us the most control, setting all the parameters is crucial, and can often be done only, if we have sample inputs to test on.

Bibliography

[1] Bespamyatnikh, Sergei & Segal, Michael. (1999). Covering a Set of Points By Two Axis-Parallel Boxes, Information Processing Letters 75 (2000) 95–100.

[2] C. Saha & S. Das, "Covering a Set of Points in a Plane Using Two Parallel Rectangles," 2007 International Conference on Computing: Theory and Applications (ICCTA'07), Kolkata, 2007, pp. 214-218, doi: 10.1109/ICCTA.2007.45.

[3] A. Lodi, S. Martello & D. Vigo "Heuristic and Metaheuristic Approaches for a Class of Two-Dimensional Bin Packing Problems"

[4] S.Gottschalk, M. C. Liny, and D. Manocha "OBBTree: A Hierarchical Structure for Rapid Interference Detection"

[5] A. V. Goldberg, ; R. E. Tarjan. "A new approach to the maximum flow problem". Proceedings of the eighteenth annual ACM symposium on Theory of computing

[6] A. Gaschler, Q. Fischer, and A. Knoll "The Bounding Mesh Algorithm"

[7] Khaled Mamou and Faouzi Ghorbel. "A simple and effcient approach for 3D mesh approximate convex decomposition."

[8] Seth Sweep "Three Dimensional Bin-Packing Issues and Solutions"

[9] Teodor Gabriel Crainic, Guido Perboli, and Roberto Tadei "Extreme Point-Based Heuristics for Three-Dimensional Bin Packing"