Team Zaragoza Report

Anthony Cheng, Gavin Shek, Vijeykannen Vijayaratnam, Ng Zheng Onn

We used Python for the implementation and Java for the visualisation for this project.

We used the Python Shapely Library to implement the geometric algorithms such as:

Polygon: The library allows the creation of Polygon objects with boundaries, exteriors and interiors which can be used with the other library functions.

a.area: provides area of polygon a

l.length: provides length of line l

a.within(b): This function returns True if polygon a lies entirely inside the boundary of polygon b

a.difference(b): This function returns the polygon formed by taking a difference b (in terms of set theory)

translate(x, y): translates polygon by x units in x direction and y units in y direction

rotate(d degrees): rotates polygon by d degrees with a specified fixed point (usually a vertex of the polygon)

We developed many different approaches to solving the problem and some were better than others depending on the question. We decided to take the highest score from all our attempts for each question for the final submission.

The algorithm

Stage 1: Pre-processing

We first sort the furniture into a queue. We found 2 ways of doing this one way was to sort in decreasing order of score (area \* unit cost) and the other sort in decreasing order of unit cost. Both ways made sense, placing the highest scoring furniture in the room first would potentially lead to a quick gain in total score for the room. While placing highest unit cost furniture in the room would ensure that the space occupied would provide the highest increase in score. Neither sorting method would consistently beat the other so we ran the algorithm twice using different sorting methods.

Stage 2: Furniture Placement

The second stage of the algorithm greedily places the furniture in the room using the queue from Stage 1. We developed multiple approaches to find appropriate locations for the furniture. Each approach had varying results and computation times.

Scanning Approach

The first method we came up with was to start at an arbitrary point in the room such as the bottom left corner, and then rotating it. The furniture is then translated by a fixed step in the x direction. After it has covered the width it is translated by a fixed step in the y direction. Essentially, at every step along the way it is checked whether the furniture lies within the room and whether it intersects with any other pieces of furniture which have already been placed in the room. If it fits, it is dropped in that location and the algorithm moves on to the next piece of furniture.

The computation time for this method depends largely on the step chosen. If step is fixed to an arbitrary value such as 0.5 then complexity would roughly be:

Where L is the length of the room, W is the width of the room, and G is the number of geometric operations at each step (rotate, translate, within, intersect). The size of G would also increase as more furniture is placed as more intersection checks must be conducted. This is very slow, especially for large rooms.

If the step is chosen to be relative to the size of the room then complexity would roughly just be:

Since the number of transformations now becomes fixed based on the step chosen. The main limiting factor for the speed of the algorithm is the geometric operations to be conducted.

In practice, the relative step was much faster (as expected) but often had poor, even failing, results for large rooms. The fixed step often had better results but took far too long to compute for large rooms.

Snapping Approach

The snapping approach was designed based on the assumption that the best place for any given piece of furniture in any given room is in a corner of the room, as this should reduce the creation of small pockets which cannot be easily occupied by subsequent pieces of furniture. To effectively use this method the room would have to be continually be redefined as once a piece of furniture is added to the room, that corner cannot be used by any other piece of furniture. The solution was to shrink the room, so that the edges of the most recently added piece of furniture become the new wall of the room, as shown below.

The difference function from Shapely helps us achieve this.

Shrinking Algorithm

Replace room with room.difference(furniture)

Our first version of the snapping algorithm would try each vertex of each piece of furniture on each vertex of the room and rotate until it found a fit.

Snapping Algorithm

for each vertex (a, b) of room

for each vertex (i, j) of furniture

translate furniture by (a-i, b-j) #Snap vertex of furniture to vertex of room

for (12 times) #rotate 180 degrees in total

rotate(15 degrees)

if furniture fits within room

place furniture at current location

continue

reject furniture

This method had a complexity of roughly:

Where Vr is the number of vertices in the room, Vf is the number of vertices in the furniture, and G is the number of geometric operations at each step. It is important to note that the G here is much smaller than the G in the Scanning algorithm as there is no need to check for intersections with existing furniture since furniture is added by shrinking the room. However, as more pieces of furniture are added, Vr increases.

Our later version of the snapping algorithm adds a second assumption which is that the vertex of the furniture which has the greatest chance of fitting in the room, is also one of the 2 vertices which form the longest edge of the furniture. This way, only 2 vertices need to be checked for each piece of furniture. We had to write our own function to find the longest edge of a polygon.

Longest edge(furniture)

current\_longest\_edge\_length = 0

current\_longest\_edge = NULL

for each edge i in furniture

if i.length > current\_longest\_edge\_length

current\_longest\_edge = i

current\_longest\_edge\_length = i.length

return current\_longest\_edge

Snapping Algorithm

for each piece of furniture

find longest edge(e)

for each vertex (a, b) of room

for each vertex of e (i, j)

translate furniture by (a-i, b-j) #Snap vertex of furniture to vertex of room

for (12 times) #rotate 180 degrees in total

rotate(15 degrees)

if furniture fits within room

place furniture at current location

run Shrinking Algorithm

continue

reject furniture

This approach roughly has a complexity of:

This significantly reduces computation time as Vf is only used to find the longest edge hence it no longer is multiplied with Vr and G.

In practice, the longest edge approach was about twice as fast as the first approach while also providing similar or better results in most cases.

Testing

The testing of the algorithms was done by calculating the coverage of the room and returning a pass/fail grade depending on whether 30% coverage was achieved. The total score for each room was also calculated for each run of the algorithm.

Input and Output

The input was parsed using a python script. A python script was created to run through all the outputs from the various algorithms and take the best answers for each room.

Workload Split

Anthony – Visualiser Design and Implementation, Amazon Web Services Virtual Machine Wrangler

Gavin – Algorithm Design and Implementation, Lead Programmer

Vijey – Algorithm Design and Implementation, Algorithm Research

Zheng Onn – Visualiser Design and Implementation