Research Skills and Methodologies
Path Optimization of 3D Printer

Michał Kowalski, 195447
Piotr Lechowicz, 195651

1   Introduction

Nowadays 3D printing is one of the fastest developing technologies. It allows us to convert digital 3D models into solid 3 dimensional objects. It is commonly used in prototyping because it allows to make 3D object without a need to use forms. The main advantages of 3D printing technology are: low cost of the printer, low cost of printing object, variety of materials and variety of technologies. To use this technology, we need to:

1. create a 3D model of our object,

2. "cut" (convert) it into set of very thin layers,

3. choose path for printing tool.

Our path should consist of points from layer currently being printed. Tool have to visit each point exactly once and can move without printing. Task is to minimize "cost" of moving between all succeding points in our path. We can calculate it for each pair in (at least) three ways:

1. distance between points,

2. time needed for move from one to next,

3. energy needed for that move.

We are assuming that our printing tool is a point moving on two perpendicular axes.

Another task was to write an environment, which will deliver layers for algorithms, provide methods commonly used in finding path, and will save and show results in human-friendly way, including graphical representation of found path. It should also be open to addition of new algorithms.

2 Mathematic description of problem

Given:

- printing layer as an array (size n x m) of binary points, where

$$X_{i,j} = \begin{cases} 1 & \text{if printing point} \\ 0 & \text{otherwise} \end{cases} \quad \text{and} i \in n, j \in m \tag{1}$$

- cost for move between two points calculated in one of three possible ways:

  - minimum distance

  $$L_{P_{a,b}, P_{c,d}} = \sqrt{(a-c)^2 + (b-d)^2} \tag{2}$$

  - minimum time (only axis which need to move more)

  $$L_{P_{a,b}, P_{c,d}} = max(|a-c|, |b-d|) \tag{3}$$

  - minimum energy (sum of movements, due to separate engines)

  $$L_{P_{a,b}, P_{c,d}} = |a-c| + |b-d| \tag{4}$$

  where $a, c \in n$ and $b, d \in m$.

Find:

- order of visiting points $V = [X_1, X_2, ..., X_p]$ where p is count of points in layer,

- total cost of path consisting of visiting points $L = \sum_{k=2}^{p} (L_{X_{k-1}, X_k})$,

- time of calculations for each considered algorithm,

such that length of path and time of calculations are minimized.

3 Description of Algorithms

Following algorithm were implemented:

- Left-To-Right algorithm

- Snake algorithm

- Edge Following algorithm

- Greedy algorithm

- Two Opt algorithm

- Greedy Two Opt algorithm

- Harmony Search algorithm

- Greedy Harmony algorithm

- Simulated Annealing algorithm

- Greedy Annealing algorithm

All implementations of algorithms are based on superclass, which makes all necessary communication between environment and algorithm, like:

- setting up an algorithm, e.g. getting a layer to print,

- sending results of algorithm to rest of application,

- measuring time of calculations.

Therefore, algorithms only have to do all calculations in method returning a path, and optionally set up all variables in separate method. It also simplifies adding new algorithms, because all they have to do is implement this method, and they have to be added to list of algorithms.

3.1 Left-To-Right algorithm

This algorithm was created mostly for testing purposes. It had to be as simple as possible, and still find correct route, so it allows to test how environment works with algorithms in general.

Algorithm starts in top-left corner and goes line-by-line from left to right, adding to route all points on its way. And because only point to print are on found route, printer will go straight from last point in one line, to first in next one containing any points. Behaviour of this algorithm is similar to traditional paper printes.
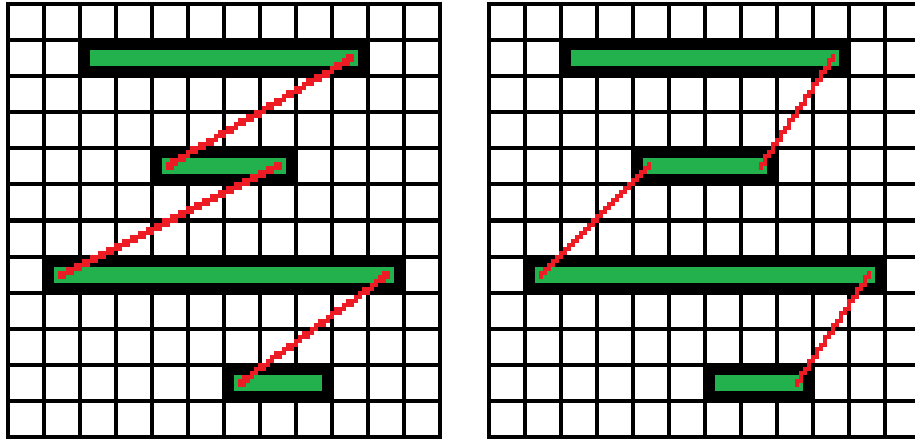
Figure 1: Left-To-Right algorithm on the left, Snake algorithm on the right

3.2 Snake algorithm

This algorithm is an improvement of Left-To-Right algorithm. Instead of going always from left to right, it finds a closer end of next line containing points, and then goes to another end of it. Differences between them are show on figure 1.

3.3 Edge Following algorithm

This algorithm tries to start from edge of a shape on a layer, and tries to get to the middle of it, and later choose next shape. It's made in few steps:

1. Check, if there's any points left.

2. Find all edges on layer.

3. Go to closest point belonging to found edges.

4. As long, as there is any neighbour of current point, go to it.

5. Go back to 1.

Finding edges and neighbours depend on chosen cost function. For example, diagonal neighbours exists only for time cost type.

3.4 Greedy algorithm

A greedy algorithm is an algorithm which always makes the choice that looks best at the moment. It makes a locally optimal choice with the hope of finding a global optimum. Greedy algorithms do not always yield optimal solution nevertheless they may give solution that approximate a global optimal solution in a reasonable time. Commonly greedy strategy for the travelling salesman

4

problem is as follows: at each stage visit an unvisited point nearest to the current city. In case of this particular problem of path finding in 3D printer, locally optimal solution was slightly modified. It has to be well described because it is used in further hybrid algorithms as a first step.

3.4.1 Greedy adjacent points modification

Lets consider an example where locally optimal solution is based only on distance between two points. In all examples distance between two adjacent points equals to one. Figure 2 shows example layer with few points. If optimal choice is made only by considering distance, the path would look as it is shown in the figure 3.
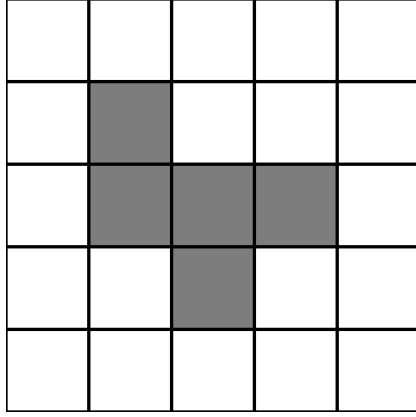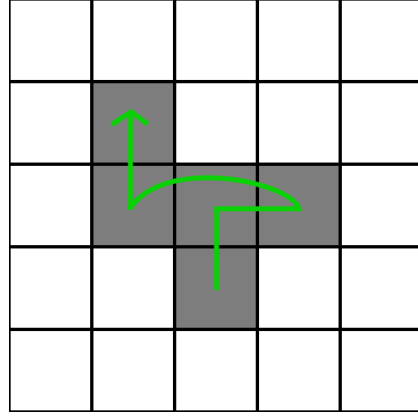

Figure 2: Example points on layer

Figure 3: Path number 1 for example layer

$$L_1 = 5 \ ,$$

where L_1 is length of the path in the figure 3

On the other hand if we begin to consider locally optimal solution not only based on distance between source and target points but also on number of adjacent points to target point, we can notice that in some cases it is better to choose point a little further but with lower number of adjacent points. It is illustrated in the figure 4.
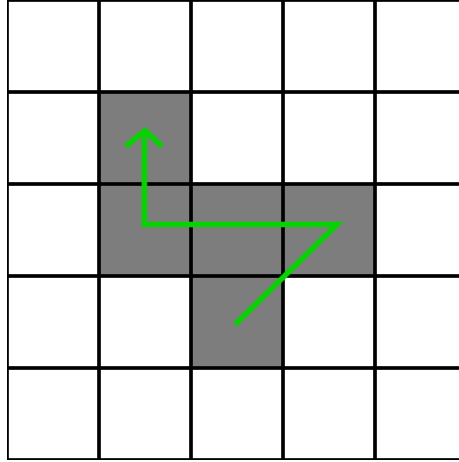
Figure 4: Path number 2 for example layer

$$L_2 = 3 + \sqrt{2} \approx 4.41 \; ,$$

where L_2 is length of the path in the figure 4

Comparing distances from figure 3 and 4 we can notice that point which is further but has only one adjacent point is a better choice than point which is closer but has two adjacent points. That's why in algorithm was introduced parameters describing how much algorithm will prefer points which are further but has lower number of adjacent points.

In all tests were used assumption that if distances are similar, it is each one of them isn't greater than the other one more than 0.42 (it comes from subtraction of distances of these two paths on figures 3 and 4), the best choice is to visit point which has the lowest number of adjacent points. If there is a point which has no adjacent point, it is the optimal choice. This way of proceeding helps to eliminate isolated points, which may finally determine result to be unsatisfying. Justification of this way of thinking is presented in figures 5 and 6.
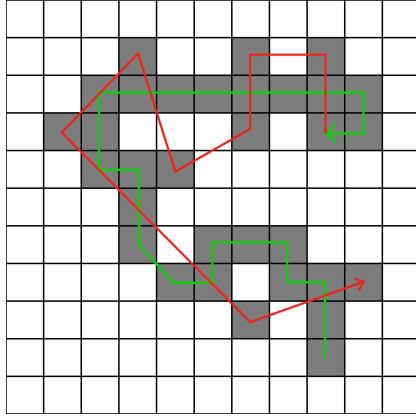
Figure 5: Path computed without considering number of adjacent points (the red one is an inefficient part of path)
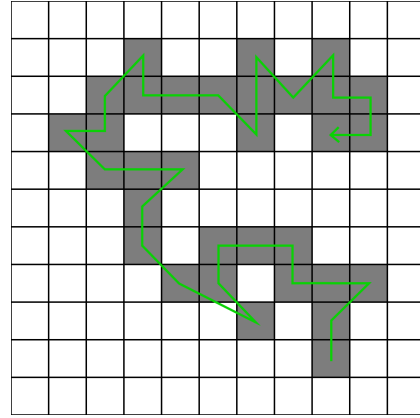

Figure 6: Path computed with considering number of adjacent points

In the figure 5 total length of path is $L \approx 47.87$ and in the figure 6 is $L \approx 35.96$. As we can see with the greater number of points to print the importance of not leaving isolated ones increases.

3.4.2 Greedy neighbour searching modification

Another introduced improvement in the algorithm is the modification of process of searching the nearest unvisited point from the currently visited one. Most of the printing layers have rather densely distributed points, e. g. most of them are located in the centre of the layer. In order to reduce time needed for checking distances to all possible point to find the nearest one, algorithm firstly search the neighbourhood of currently visited point, it is places for possible points, which are not further than one unit in each side. In case of negative result algorithm extends searches to two units in each side. If in that process any point was obtained, algorithm starts looking for next move by searching a list of all possible points. Figures from 7 to 10 illustrate this part of algorithm.
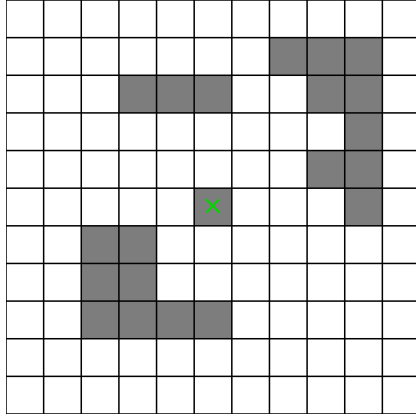
Figure 7: Starting position for searching neighbourhood (green cross represent currently printed point)
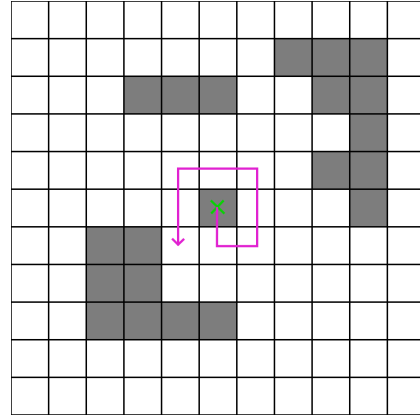

Figure 8: First iteration of searching

In the figure 7 we have starting point which was the latest one visited. Algorithm starts to search neighbourhood of the point. In the figure 8 is shown first iteration. Because algorithm didn't find any point it extends searches which is presented in the figure 9. Because there are three possible solutions for the next move, algorithm has to chose the most optimal one. It considers both criteria - distance and number of adjacent points. The result is the upper one (figure 10).
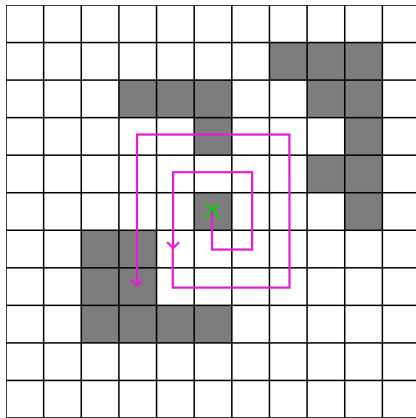

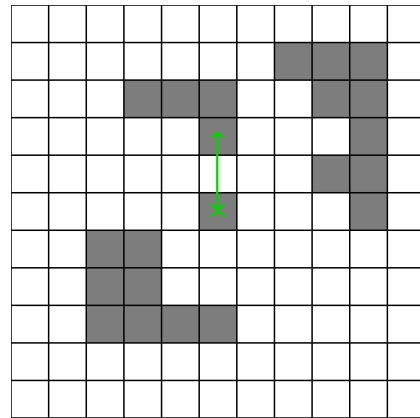Figure 9: Second iteration of searching

Figure 10: Move of the printing tool

3.4.3 Greedy parameters

Implemented greedy algorithm allows parametrization in order to adapt it to different kinds of layers. The list of parameters with its description:

- `seed` - seed of the random. Some parts of algorithm are randomized. Instance of Java class `Random` is used to generate a stream of pseudorandom numbers. It uses a 48-bit seed as the initial value of the internal state of pseudorandom number generator. If parameter is set to -1 the seed is the current time in miliseconds since January 1, 1970.

- `nrOfThreads` - number of threads. Algorithm can consider different points as a starting point. Number of threads determine the amount of concurrently finding solutions.

- `nrOfNeighboursForStartingPoint` - number of neighbours for starting point. Preferable amount of adjacent points for the starting one. If there are no more such points, remaining starting points are generated randomly.

- `nrOfPointsNeededToCheckArray` - number of points needed to check array. Normally algorithm searches neighbourhood for finding nearest point. When number of remaining points to print is lower than this parameter algorithm straight away starts checking all possible points for finding the nearest one, without considering neighbourhood of the point.

- `bestNrOfNeighours` - best number of neighbours. Sets preferable number of adjacent points for the next moves of printing tool.

- `weightOfNeighbours` - weight of neighbours. It determines how much algorithm would prefer points which are further but have closer number of adjacent points to the parameter `bestNrOfNeighbours`. It is simply acceptable difference of distances between this point and the nearest one.

- `weightOfDistance` - weight of distances. It determines how much algorithm would prefer points which are nearer but their number of adjacent points is farer than defined in parameter `bestNrOfNeighbours`. It is simply acceptable difference of distances between the nearest point and the point with closer number of neighbours to the preferable one.

4   Experiments

4.1   Environment

Test environment was implemented, to provide:

- delivering chosen layers to algorithms,

- graphical representation of layer and path,

- displaying and showing to file results of algorithms,

- easy adding of new algorithms (by inheriting one class and adding it to algorithm's list),

- methods for reading algorithm's parameters from file.

4.2   Test layers

Images used as test layers are in "img" directory added to this report.

5   Results

For very short calculations, measured time can has very big differences between one measurement and another one with the same algorithm/params.

All result images and text files are added to this report in "results" directory.

6   Conclusion

As we can see on figure 11, when algorithm is seeking for closest point, it finds different ones, depending on cost function type. Therefore, if algorithm makes use of this, the results will vary.

Snake algorithm usually provides almost 2 times better results than Left-to-Right, in similar time. For some layers it also provides optimal (or close to it) paths, whereas for other it can give results multiple times worse than optimal.
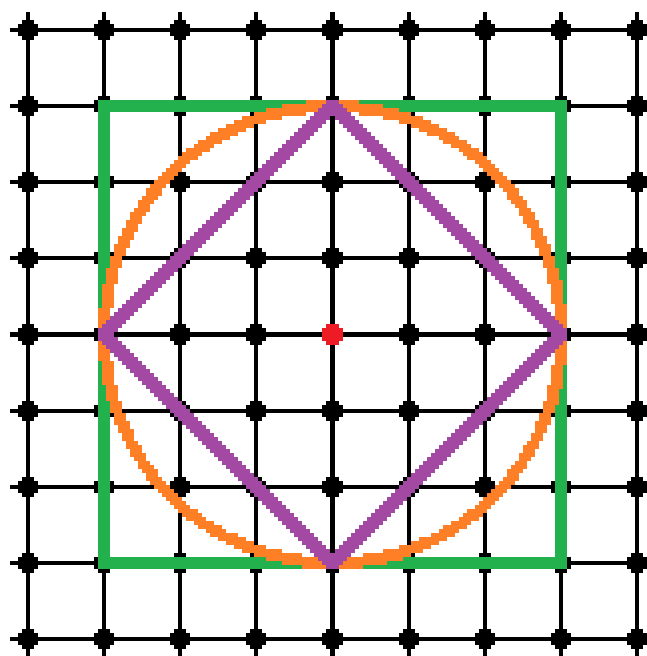
Figure 11: Points with cost from red point equal to 3. Orange for distance, green for time, and violet for energy

At least for some algorithms, not only count of points, but also size of whole layer (even if points are only in the middle) can have influence on results or time of calculations, e.g. when algorithm is searching for closest point.

Edge Following algorithm can provide very good results in very short time.

7 Own contribution

Michał's contribution:

- application for tests and graphical representation of results,
- Left-To-Right algorithm,
- Snake algorithm,
- Edge-Following algorithm,
- layers sp-5 and sp-6,
- time and energy cost functions,
- experiments and this report.

Piotr's contribution:

- layers from sp-7 to sp-9,
- experiments and this report.

8 References