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.

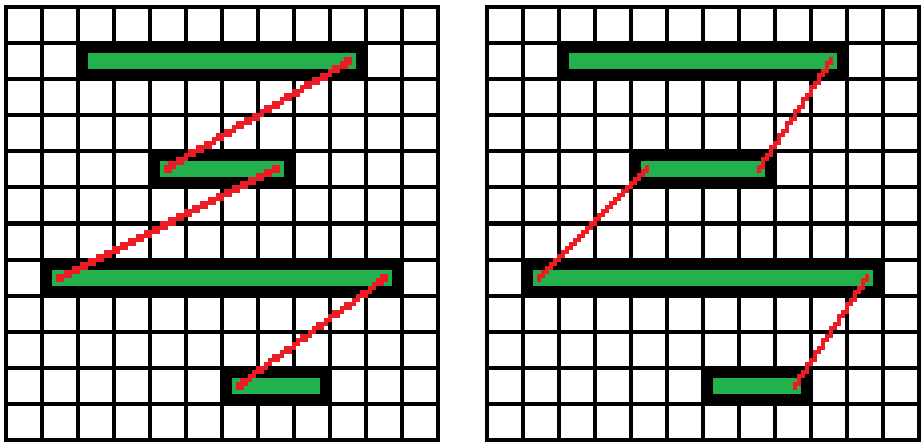

Figure 1: Left-To-Right algorithm on the left, Snake algorithm on the right

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 approximates a global optimal solution in a reasonable time. Commonly greedy strategy for the travelling salesman

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 of its good results it was used to construct further hybrid algorithms.

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.

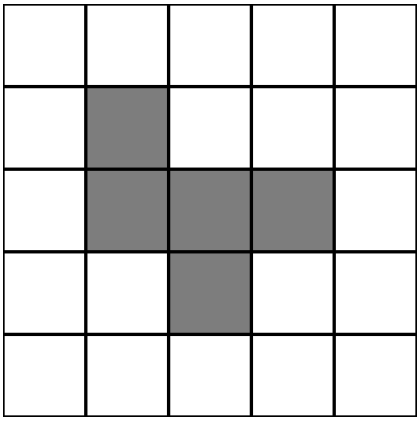

Figure 2: Example points on layer

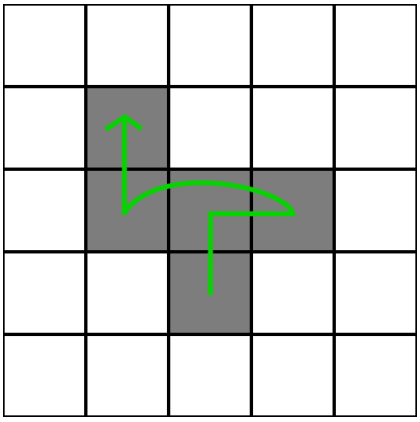

Figure 3: Path number 1 for example layer

$$L_1 = 5 \ ,$$

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.

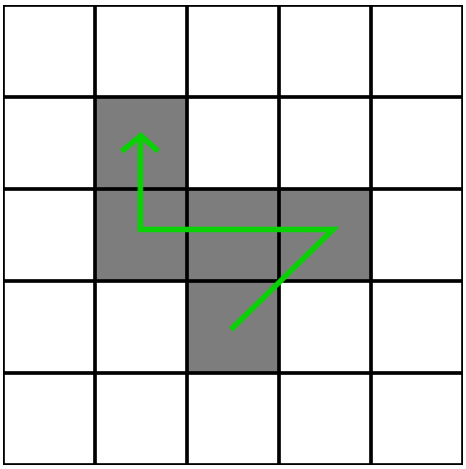

Figure 4: Path number 2 for example layer

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

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.

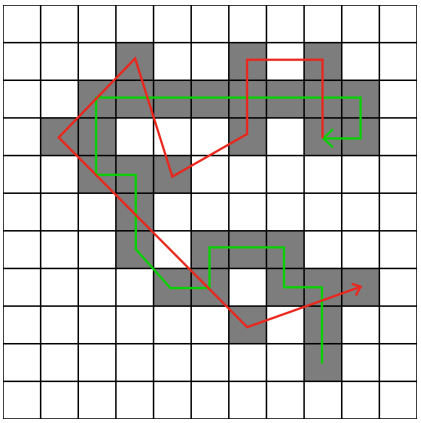

Figure 5: Path computed without considering number of adjacent points (the red one is an inefficient part of path)

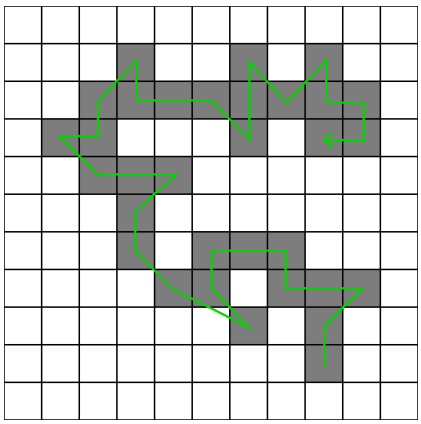

Figure 6: Path computed with considering number of adjacent points

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.

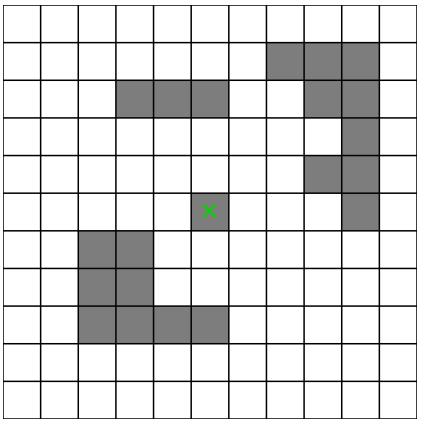

Figure 7: Starting position for searching neighbourhood (green cross represent currently printed point)

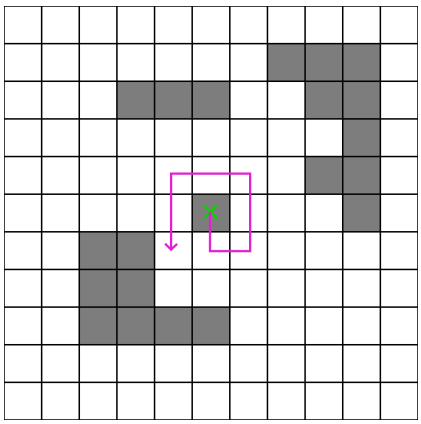

Figure 8: First iteration of searching

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).

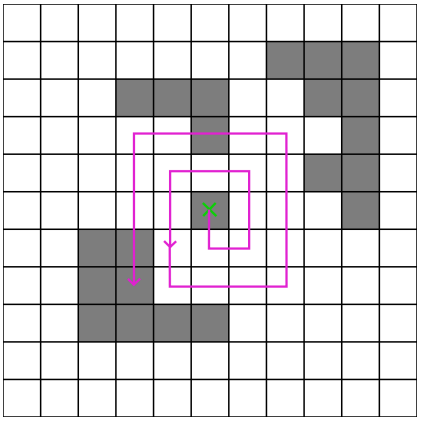

Figure 9: Second iteration of searching

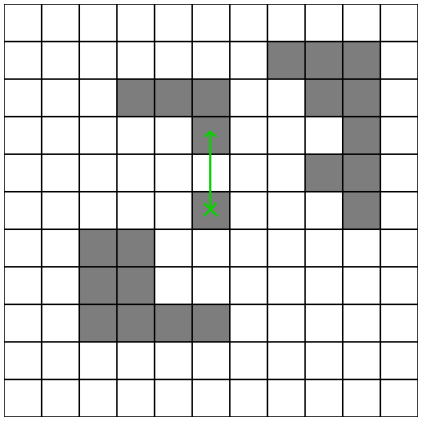

Figure 10: Move of the printing tool

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 greater 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.

3.4.4 Greedy procedure

The overall procedure of finding path realized by this algorithm:

1. create and fill table of neighbours - the table which contains amount of adjacent points for each of them
2. set starting point for each thread
3. start threads:

3.1. if there are more unvisited points than `nrOfPointsNeededToCheckArray`:

3.1.1. search for the nearest point in the neighbourhood

3.1.2. if point wasn't found search for optimal point in the list of all remaining points.

3.2. otherwise:

3.2.1. search for optimal point in the list of all remaining points

4. after joining threads get all result paths from them

5. return path with lowest distance

Procedure of finding optimal point ($d[point]$ is the distance from current point to that point, $n[point]$ is the number of adjacent points):

1. take distance and number of adjacent points from first point as a best ones ($best \leftarrow point$)

2. if $d[best] \leq 1 +$ `weightOfNeighbours` and $n[best] \leq$ `bestNrOfNeighbours`, return this point as an optimal (`return` $best$)

3. while there are unchecked points:

   3.1. if $d[next] < d[best] -$ `weightOfDistance` take this point as a best one ($best \leftarrow next$)

   3.2. else if $d[next] < d[best] +$ `weightOfNeighbours` and $n[next] < n[best]$ take this point as a best one ($best \leftarrow next$)

   3.3. if $d[best] \leq 1+$`weightOfNeighbours` and $n[best] \leq$ `bestNrOfNeighbours` return this point as an optimal one (`return` $best$)

   3.4. else continue while loop

3.5 Two Opt algorithm

This algorithm is based on the exchange or swap of a pair of edges. To swap two edges (a, b) and (c, d), the nodes of the pair of edges are rearranged as (a, c) and (b, d). The algorithm tries to untangle crossing edges. There were implemented two variety of this algorithm. In the first one it searches for first swap which will result in shortest total length of path and executes it. The procedure is repeated till no improvements are found. The difference between the second one and the first one is that it searches at each iteration for the best possible swap and then executes it. The idea of the algorithm is presented in the figures 11 and 12.

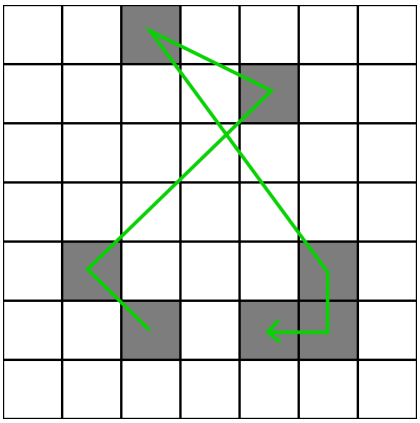

Figure 11: Path before swapping edges

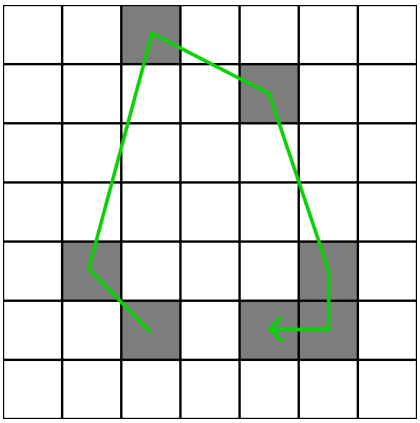

Figure 12: Path after swapping edges

3.5.1 Two Opt parameters

- `isGreedy` - is greedy flag. When `isGreedy` is `false` algorithm searches for the best possible swap, it is it checks all possible pairs of edges and chooses the most profitable. If flag is `true` it executes swap with first found pair of edges which will shorten the total length of path.
- `maxNrOfIterations` - maximal number of iterations. If it is set to -1 algorithm improves path till it cannot find any more beneficial swaps. If it is greater than 0, algorithm makes exact number of swaps.

3.5.2 Two Opt procedure

1. find first random path
2. if `isGreedy` == `true`:
   find first beneficial pair of edges
   else:
   find the most profitable pair of edges
3. if wasn't find such a pair return from algorithm
4. otherwise swap edges
5. if maximal number of iterations is not defined or is defined but `currentIteration` < `maxNrOfIterations` jump to 2nd step
6. else return from algorithm

3.6 Greedy Two Opt algorithm

Greedy Two Opt is composed of two algorithms. The first stage is to find path through Greedy algorithm. This path is passed as an initial path to Two Opt algorithm. The idea behind it is that it is much better to improve path close to the optimal solution than totally randomly selected. It takes much less iterations for Two Opt to finds path which it cannot improve any more.

3.7 Harmony Search algorithm

The Harmony Search is an algorithm inspired by the improvisation process of Jazz musicians. If musician plays he can choose to select a sound from his memory or randomly select other note. If it 'sounded' good he can 'try' to remember it for future use. The memory is a list of travelling salesman problem solutions (they must start from the same point). Algorithm find new solution with the following way: it starts from the first point and with certain probability chooses next point from memory or chooses from rest of the available points. It is repeated till all points are used exactly once. If it is generated new solution it is compared to the solutions within the memory. If found solution is better than the worst from the memory, the one from the memory is removed, and this one is added. This steps are repeated for particular number of iterations (time of jazz improvisation).

3.7.1 Harmony Search parameters

- `seed` - seed of the random. Similarly to the Greedy algorithm's seed.
- `memorySize` - size of the memory. Amount of best paths being stored in the memory.
- `nrOfIterations` - number of iterations. How many times algorithm repeats finding path and comparing if it is better than paths in the memory.
- `memoryProbability` - probability that algorithm chooses point from the memory rather than randomly from the rest of unvisited points.

3.7.2 Harmony Search procedure

1. choose starting point
2. fill memory with random paths, each starts from the starting point
3. create a *newpath* containing only starting point
4. if next point is taken from memory:
   - 4.1. randomly select one of the paths within the memory
   - 4.2. take point with the corresponding place, it is, if is searched $n-th$ point for the *newpath* it is taken $n-th$ point from path from the memory

4.3. if that point is already in *newpath*, the point is exchange with the one being chosen randomly from the rest of unvisited points in this iteration

4.4. add point to the *newpath*

4.5. if *newpath* is not containing all points from the layer, algorithm jumps to 4th step

else

4.1. randomly select point from the list of unvisited points in this iteration and add it to *newpath*

4.2. if *newpath* is not containing all points from the layer, algorithm jumps to 4th step

5. if *newpath* is better than the worst one in the memory, they are swap, otherwise this new solution is rejected

6. steps from 3 to 5 are repeated `nrOfIterations` times

7. return best path from the memory

3.8 Greedy Harmony algorithm

The first stage of this algorithm is to find solution with Greedy algorithm and then it is passed as an initial solution to Harmony Search. The same solution is added to the harmony's memory as many times as it is necessary. Then Harmony Search algorithm tries to find solution closer to optimal. Because of the highly randomness of Harmony Search, setting starting path as already close to optimal solution may highly improves results of an algorithm.

3.9 Simulated Annealing algorithm

Simulated Annealing algorithm is inspired by annealing process in metallurgy. It emulates the physical process in which solid is firstly heated and then slowly cooled. With higher temperature metal is more vulnerable to deformation because of providing energy needed to break bonds. While the temperature becomes lower and lower it is harder to deform metal. In case of Simulated Annealing algorithm the temperature determines the probability of taken worse solution than currently obtained one in the future consideration. With lower temperature algorithm less likely takes worse solution. During whole process the best found solution is kept in memory, and replaced if better one is obtained. New solutions are found by swapping two randomly chosen edges. When temperature reaches earlier define minimum, algorithm stops. The probability of acceptance worse solution is calculated as follows:

$$e^{-\frac{J(j)-J(i)}{T(t)}},$$

where $J(i)$ is total distance of currently found solution, $J(i)$ is total distance of the best found solution and $T(t)$ is the temperature of time where the new solution was obtained.

3.9.1 Simulated Annealing parameters

- `seed` - seed of the random. Similarly to the Greedy algorithm's seed.
- `temperatureMin` - minimal temperature. Temperature at which algorithm stops.
- `coolingRate` - cooling rate. It determines how fast temperature is decreasing. New temperature is get by multiplying currently one by this factor.
- `iterationsOnTemperature` - how many iterations algorithm does on each temperature state, it is, how many times it tries to find better solution with the same probability of acceptance worse solution.

3.9.2 Simulated Annealing procedure

1. obtain randomly first solution
2. set this solution as the *best* one and as a *currentpath*
3. select randomly two edges and swap them ($newpath \leftarrow swap(currentpath)$)
4. if $distance[newpath] < distance[currentpath]$:
   4.1. $currentpath \leftarrow newpath$
   4.2. if $distance[currentpath] < distance[best]$:
   $best \leftarrow currentpath$

   else:

   4.1. take *newpath* as a *currentpath* with calculated probability for this temperature
5. repeat steps 3 to 4 `iterationsOnTemperature` times
6. calculate new temperature ($temperature \leftarrow$ `coolingRate`$*temperature$)
7. assign best path to current path ($currentpath \leftarrow best$)
8. repeat steps 2 to 7 while $temperature > minimal\_temperature$
9. return *best* path

3.10 Greedy Annealing algorithm

In this algorithm initial path for Simulated Annealing is obtained by executing Greedy algorithm. All procedure steps and parameters are the same as in both of these algorithms. Simulated Annealing may be much more efficient if it starts with solution not o far from the optimal 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 13, 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.

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.

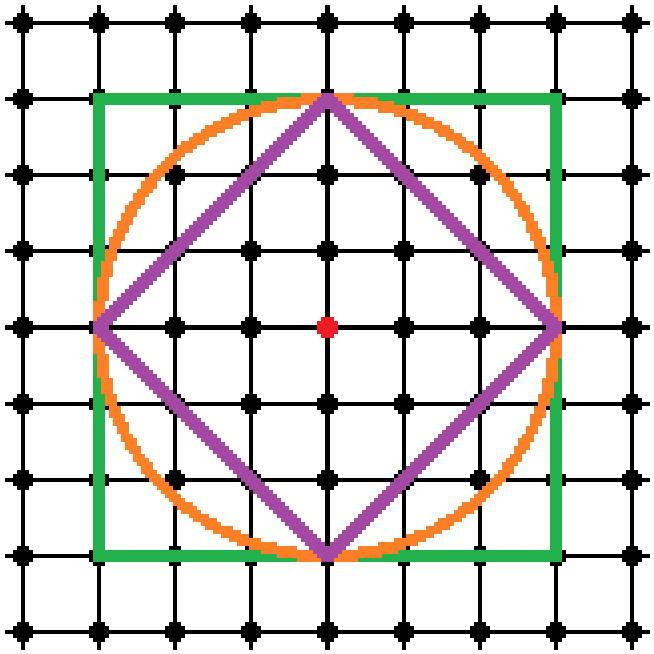

Figure 13: Points with cost from red point equal to 3.
Orange for distance, green for time, and violet for energy

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