Univerza *v Ljubljani*
Fakulteta *za elektrotehniko*

Report on Elective project

Pathfinding algorithms and visualization using Python

Course: Artificial Intelligent Systems
Author: Jakob Veršnjak
14. january 2020, Ljubljana

1 Introduction

The main aim of the elective project was to program different search algorithms, which would try to find the shortest path from point A to point B on the grid of squares. The algorithms were programmed with a raw Python code without using any additional libraries. I only used a Numpy module for setting a random seed. Then I tried to visualize the performance of each algorithm using **Python module Pygame**. At the end I analyzed the performance of different algorithms on weighted and unweighted problem.

Final product has 5 different search algorithms that user can choose from by clicking on corresponding button:

1. Breadth-first search
2. Depth-first search
3. Dijkstra's algorithm
4. A* search
5. Greedy best-first search

Breadth-first search (BFS) and Depth-first search (DFS) are unweighted algorithms. Dijsktra's algorithm, A* search, Greedy best-first search (Greedy BFS) are weighted algorithms. The difference between weighted and unweighted algorithms will be explained in the third chapter.
I implemented and visualized the search algorithms on two different problems:

1. Unweighted map
2. Weighted map

The code is separated into 4 different files: **Main.py**, **Grid.py**, **SearchAlgorithms.py** and **DrawFunctions.py**. You need to run Main.py file to start a program. You can set the unweighted or the weighted map and then you run a code. The GUI will display and you can select an algorithm by clicking on the button.
In Grid.py two classes are described: Grid and Spot. Class Spot represents each square on the grid. Class Grid inherits the class Spot and it creates a grid of n*n squares (Spot).
Search algorithms are written in SearchAlgorithms.py. The functions which are being used for visualizing the performance of the algorithms in real time are saved in the last python script DrawFunction.py

2 Results

In this chapter I'll explain two different problems which could be solved by using search algorithms and how does the pathfinding visualization work.

2.1 Unweighted map

On the figure 1 you can see the display of unweighted map. An unweighted map can be displayed by writing a line of code in Main.py:

```
Graph = Grid(NUM_SQUARES, [start_y, start_x], [finish_y, finish_x],
barrier_per=25)
```

NUM_SQUARES is the size of the map (for example: NUM_SQUARES = 50 means grid with 50*50 squares). We need to add the position of starting and finish square. And we can also determine percentage of the grid covered with obstacles (black squares).
The user can choose from 5 different algorithms by clicking on the button on the display screen. The algorithms are then searching for the path from starting node (red square) to finish node (blue square) avoiding obstacles. Not all the algorithms guarantee the shortest path. This map is called **unweighted** because all nodes (squares) has the same cost (i.e. 1, because each square represents the distance 1).

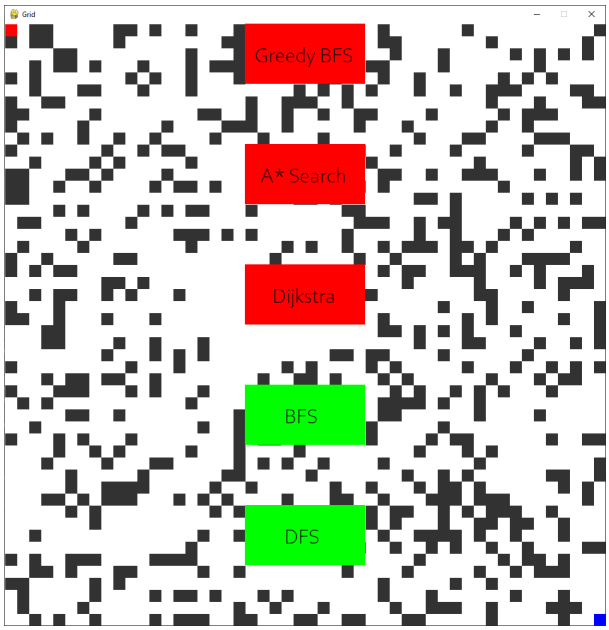

Figure 1: Unweighted map

2.2 Weighted map

On the figure 2 we can see the display of weighted map. The brightness of the squares represents the cost of each square, which is not constant. The brighter squares represent lower cost and darker regions are weighted with higher cost as seen on figure 3. We can imagine the cost of squares as an example of the real map of the terrain. Black squares could represent hills and brighter spots could be seen as valleys. If you want to avoid climbing hills, the path with the lowest cost is the way to go (the path across brighter squares). The weighted map is created by this line of code:

```
Graph = Grid(NUM_SQUARES, [start_y, start_x], [finish_y, finish_x],
barrier_per=0, weights=True, max_weight=4)
```

You just need to set the parameter weights on True and choose the maximal weight (squares with cost in range from 1 to maximal weight are then generated). You can also add obstacles that are colored with purple with parameter barrier_par. For this case only the weighted algorithms are applicable because they consider the cost of the path which is not necessarily the shortest (with the shortest distance). Unweighted algorithms (DFS and BFS), on the other hand, are just expanding the tree structure no matter the cost. More on that in the third section.

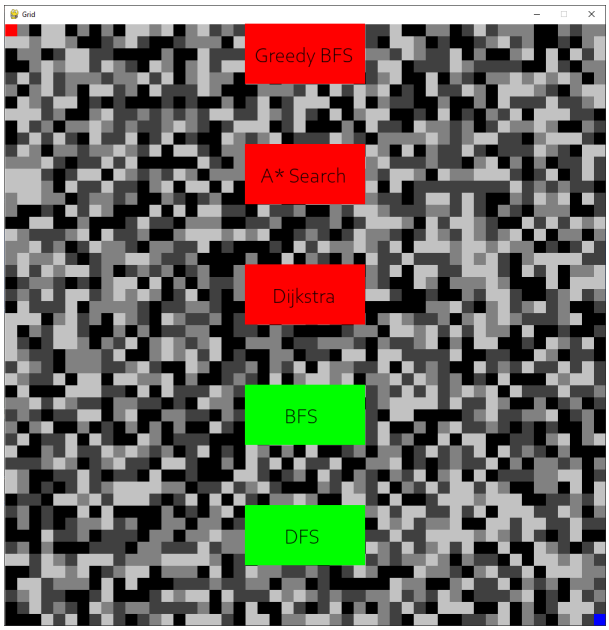

Figure 2: Unweighted map

Figure 3: Cost of each position with the maximal weight of 4

2.3 Pathfinding visualizator

By clicking the button, the visualization of algorithm starts. We can see in real time how do different algorithms search for the finish node. When it gets to the finish the algorithm stops and the found path is drawn with the yellow color as seen on figure 4.
The colored circles represent the area already searched by an algorithm (closed set). The color of circle indicates the distance from finish (the redder the further, the greener the closer). Then we can also see the connection between circles which illustrates us that this problem is a tree. Each square can have a parent and a child which my program keep track of. This is urgent for reconstructing the final path from start to finish by backtracking (we can see that yellow circles are connected, the last circle is actually finish and the first one is start).
Finally, few blue circles can also be spotted. They represent so called open set which means that algorithm has located these nodes as valid, but they haven't been checked yet (the cost hasn't been calculated). In next iteration an algorithm takes the nodes from open set that hasn't been checked yet and adds them to closed set (circle is then colored). This applies only for weighted algorithms.

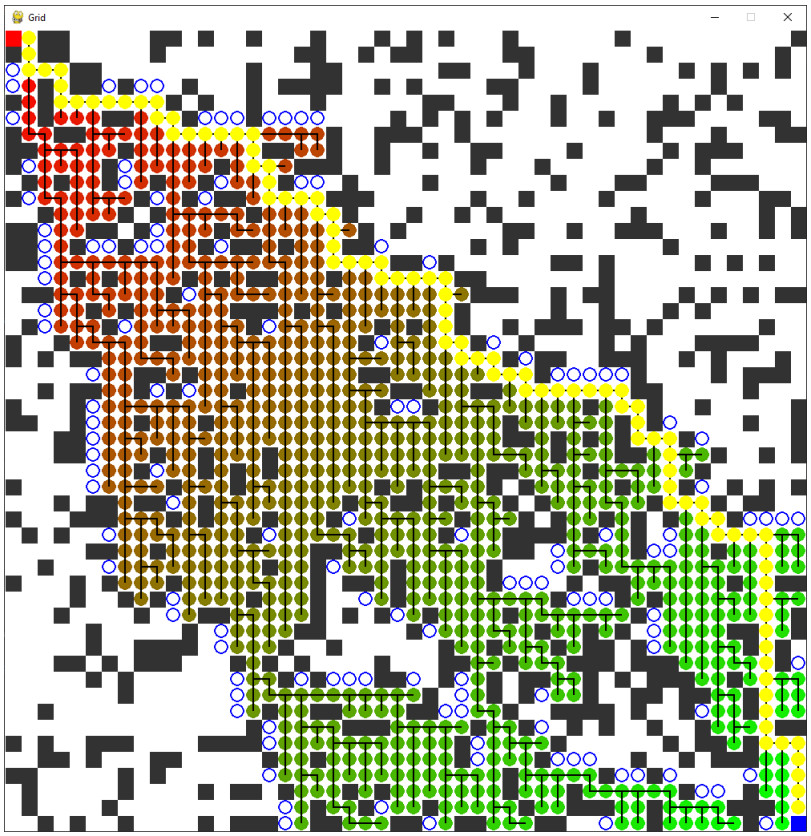

Figure 4: Search algorithm visualization

3 Performance of search algorithms

In this section I will analyze the performance of each search algorithm on the both maps. I considered the optimality (Is the found path the most optimal one?), completeness (Does it always find a solution, if there is any), space and time complexity. I used the constant seed so that all the maps have the same position of obstacles (both maps) and costs (weighted map). This is necessary for objectivity of analyzation. I created the grid with 50 by 50 squares. The starting square was set to upper-left corner and finish is located at bottom-right. The Manhattan distance is equal to 100. That is the shortest path (it is actually 99 in terms of how many squares does the shortest path consists of) if there were no obstacles.

3.1 Performance on unweighted problem

3.1.1 Breadth-first search

BFS does perform excellent on unweighted map due to its optimality. The most optimal path is guaranteed. On the figure 5 we can see the shortest path which is exactly 99 squares long. BFS works like this: Firstly, it expands and checks all the nodes with depth 1 (depth = distance for unweighted problem). Then it checks all the nodes with depth of 2 and so on. Inevitably the finish node will be found, and by backtracking parent nodes from the finish will reconstruct the optimal path. The only downfall is the space complexity as seen on figure 5. We can see that almost all the squares on the map were checked.

3.1.2 Depth-first search

DFS does not guarantee optimal path, it performs poorly on the problem of finding the shortest path (figure 6). The found path is 242 squares long which is far from optimal. DFS

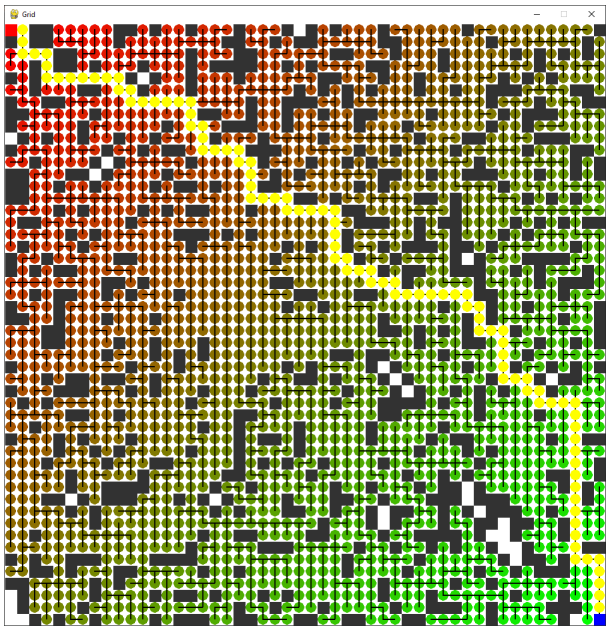

Figure 5: Breadth-first search on unweighted map

main idea is to expand the furthest node that can be expanded. Therefore, the distance is too long. We could almost say that DFS finds the least optimal path. But if there is only one possible solution (e.g. maze), DFS is the suitable algorithm to use as it is also used for generating the maze.

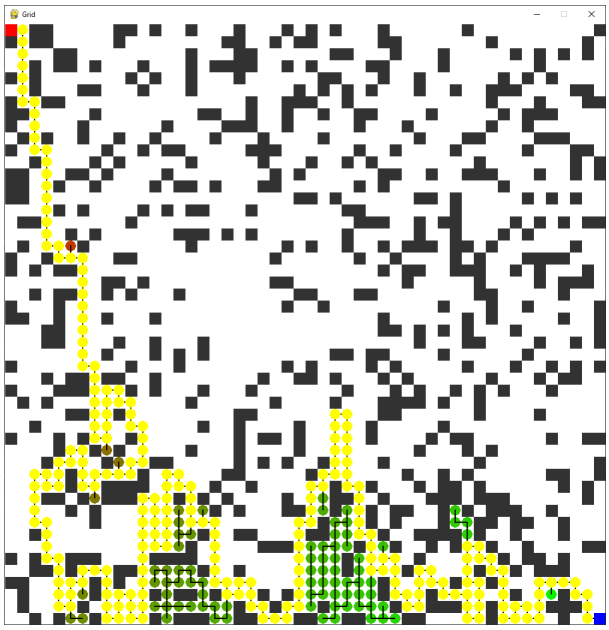

Figure 6: Depth-first search on unweighted map

3.1.3 Dijkstra's algorithm

Dijsktra's algorithm is weighted algorithm and for unweighted graph this algorithm simplifies itself to Breadth-first search (figure 5 and 7 are identical). Dijkstra's algorithm works similar to BFS: it always checks the node with the lowest cost which in the case of unweighted map

means the nodes with the shortest path. There is no reason to use Dijkstra's algorithm for unweighted graph, but it has its place on solving the weighted problems.

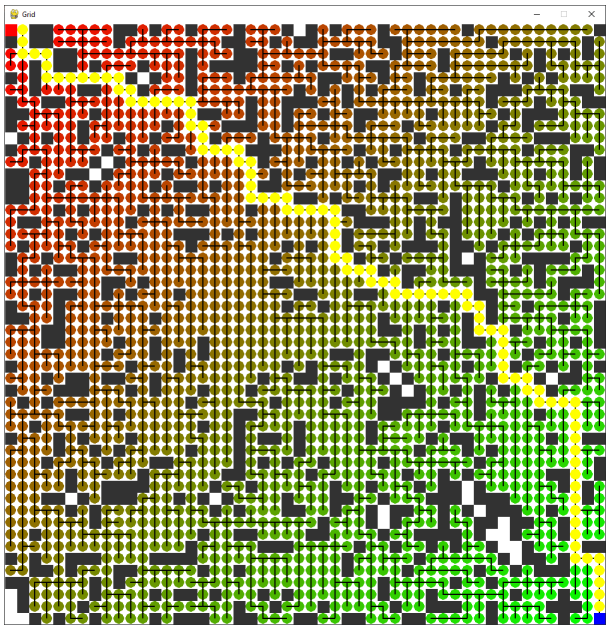

Figure 7: Dijsktra algorithm on unweighted map

3.1.4 A* search

A* search performs the best. It finds the most optimal path and it checks fewer squares as BFS 8. It is more intelligent because it considers the distance from the current square (which is being checked) to the finish. It is expanding the node with the smallest $f(n)$ which is calculated for each visited spot as $f(n) = g(n) + h(n)$. g(n) is the cost of already path which in our case means the distance from start to current spot. $h(n)$ is a heuristic function which estimates the path cost to the final state (I used Manhattan distance). Therefore, the states closer to final state are more likely to be checked as distant spots. It is interesting that this method guarantees the optimal solution if the following conditions are satisfied:

- $h(n) \leq h^*(n)$, where $h^*(n)$ is the actual cost
- $h(n) \geq 0$
- $h(G) = 0$, where G denotes the final state

With Manhattan distance as heuristic function I satisfy all the conditions, therefore the solution will always be optimal.

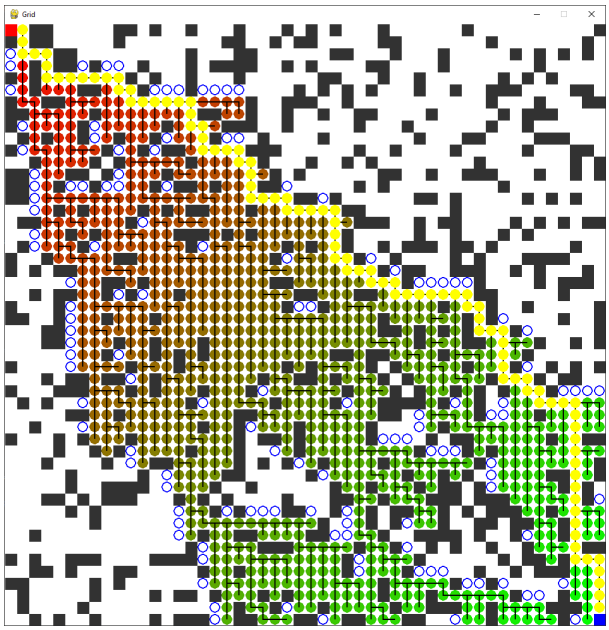

Figure 8: A* search on unweighted map

3.1.5 Greedy best-first search

As the name suggests, this algorithm can find the solution very quickly. On figure 9 is visible that very few spots are being checked. This algorithm is very similar to A* search with one major difference. It calculates $f(n)$ using only heuristic function not considering the current cost of the path. The main goal is to choose the next node that is closer to finish (if we use Manhattan distance as heuristic function). But it has its downfalls. The path is not optimal (111) and it is not complete (solution can be not found although it exists). But it is very fast, and the path is not far from optimal.

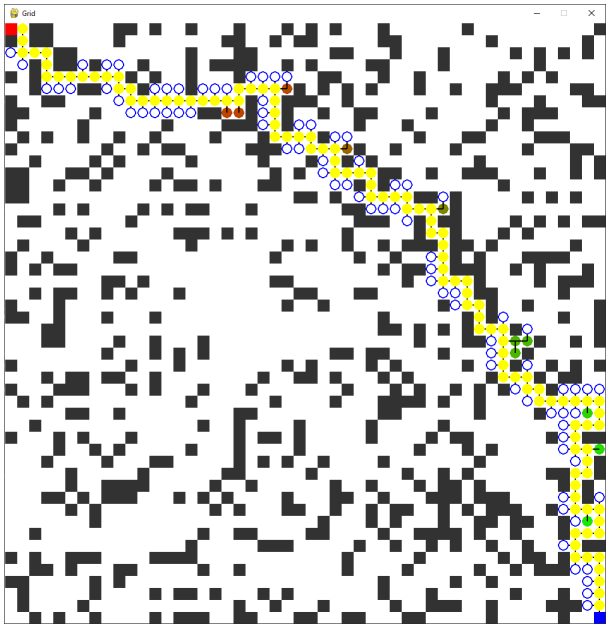

Figure 9: Greedy best-first search on unweighted map

3.2 Performance on weighted problem

3.2.1 Breadth-first search

BFS is an unweighted algorithm, therefore it is not suitable for weighted graph. The solution on figure 10 shows us that BFS is acting like all the positions has the same cost and that there are no obstacles. The only logical path is then the Manhattan distance. The whole cost of the path is equal to 241 (the most optimal is 156 for this example).

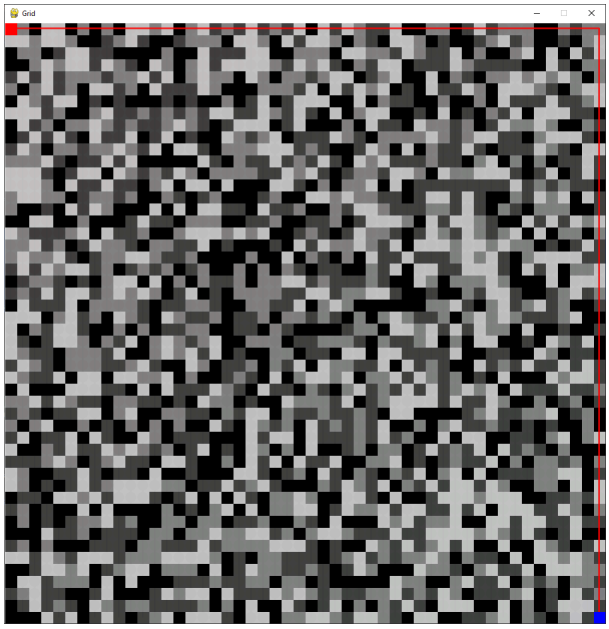

Figure 10: Breadth-first search on weighted map

3.2.2 Depth-first search

DFS is also unweighted algorithm and doesn't take into the account the actual cost of the spots. We can certainly say that it finds the least optimal path covering the whole grid with the snake-like form path (figure 11). The whole cost is equal to 6155.

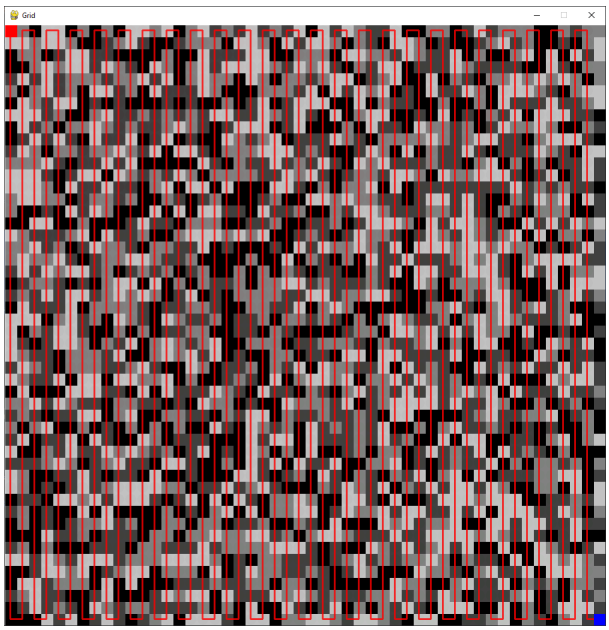

Figure 11: Depth-first search on weighted map

3.2.3 Dijkstra's algorithm

Dijkstra's guarantee the optimal path, but it is kind of slow because it checks all the possible solutions. The cost of the path is 156 and we can see that the red line follows the brighter squares which represent the lower cos nodes. Dijkstra's algorithm is very commonly used in many applications of similar problems.

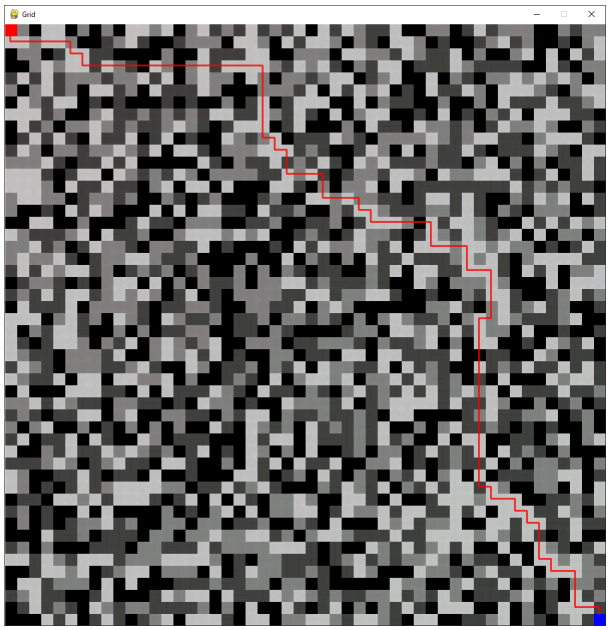

Figure 12: Dijsktra algorithm on weighted map

3.2.4 A* search

A* finds the most optimal path and it is also quicker as Dijkstra's algorithm as it does not check every possible path. I choose Manhattan distance for heuristic function also for this example and it satisfies all the conditions for optimality. If we compare figure 12 and 13, we can notice that the paths are not the same. There are two different paths that has the same cost. We can clearly see that these algorithms will find you only **one solution**. If there are more possible solutions, the algorithms must be slightly modified.

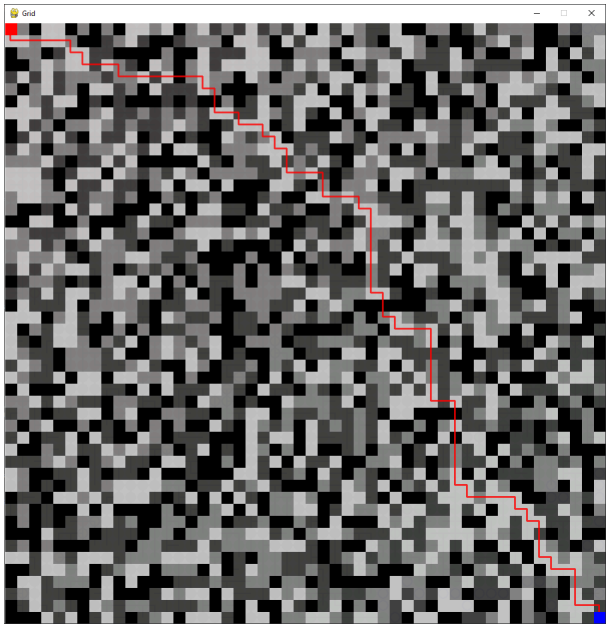

Figure 13: A* search on weighted map

3.2.5 Greedy best-first search

This algorithm is also not suitable for weighted problem because heuristic function is not considering the actual cost but only the distance from final state. It finds the same path as BFS because it tries to reach the finish line as fast as possible. If I would choose different heuristic function which would consider the cost, the performance could be much better.

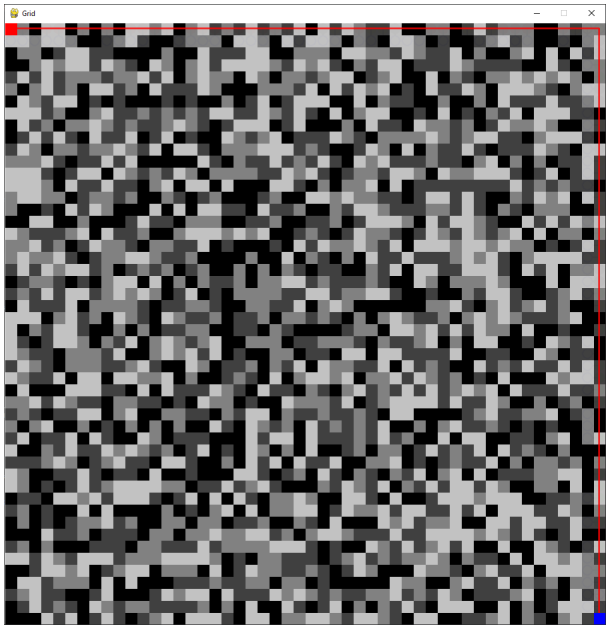

Figure 14: Greedy best-first search on weighted map

4 Conclusion

I learned a lot from this project. I gained a deep understanding on how different search algorithms performs and how to implement these strategies on a particular problem. This was the first bigger coding project (around 500 lines of code) and I've came to a realization how important is the good base structure of the code on which you can built upon.