Concordia University

COMP 6721 Introduction of AI – Fall 2019

Project 1

Student: chenwei Song
40024460
2019/10/14

1. Introduction to technical details

In the project, we will generate the crime risk map based on file "crime_dt.shp", which includes all the crime data in a specified area. Using this local area, we generate grids in this area to compute the number of nodes in each grid. Each grid has the size less or equal of 0.002*0.002. A threshold would be introduced as a partition to find the median. The crime rate of a block such that equal or higher than median would be labelled in yellow block, and blocks with the crime rate lower than median are labelled in dark blue. According to the threshold and grid size, we would generate a grid graph that each grid colored with yellow and dark blue.

The goal of project is that trying to find an optimal path from start point and end point with heuristic algorithm. The optimal path should not go through the yellow block and boundary.

A* algorithm

Our algorithm is mainly based on A* algorithm. A* is a computer algorithm that is widely used in pathfinding and graph traversal, which is the process of finding a path between multiple points, called "nodes". A* is an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs; staring from a specific starting node of graph, it aims to find a path to the given goal node having the smallest cost.

The secret to its success is that it combines the pieces of information that Dijkstra's Algorithm uses (favoring vertices that are close to the starting point) *and* information that Greedy Best-First-Search uses (favoring vertices that are close to the goal). In the standard terminology used when talking about A*, g(n) represents the *exact cost* of the path from the starting point to any vertex n, and h(n) represents the heuristic *estimated cost* from vertex n to the goal. In the above diagrams, the yellow (h) represents vertices far from the goal and teal (g) represents vertices far from the starting point. A* balances the two as it moves from the starting point to the goal. Each time through the main loop, it examines the vertex n that has the lowest f(n) = g(n) + h(n).

2. Description and justification of heuristics

Heuristic function

The heuristic evaluation function is an important part of the AI in the area of data statistics, and it determined the decision-making steps of AI. The main task is to evaluate the importance of each node in the state space tree.

In general, the objective of heuristic is to produce a solution in a reasonable time frame that is good enough for solving the problem at hand. The solution may not be the best of all the solutions to this problem, or it may simply approximate the exact solution. But it is still valuable because finding it does not require a prohibitively long time.

Since my heuristic algorithm based on A* algorithm, the pseudo code looks like as follow:

```
Initialization
Start point, end point
The whole map as a matrix
Open_list $\leftarrow \emptyset$
Close_list $\leftarrow \emptyset$
Open_list $\leftarrow$ Open list $\cup$ {start point}
While open list is not empty do
    Current_node $\leftarrow$ ExtractMin(Open_list)
    Close_list$\leftarrow$ Close_list $\cup$ Current_node
    For each direction_point that the current node could pass do
        If the current_node could not arrive the direction point or the direction_point exists
        in Close_list
            Ignore the direction_point
        Else
            If the direction_point is end point
                Find path
                Print_path()
                break
            If the direction_point is not in open_list
                open_list $\leftarrow$ open_list $\cup$ direction_point
                direction_point.father $\leftarrow$current_node
                record f(n),g(n) h(n) for the direction_point
            if the direction_point is in open_list
                next_point $\leftarrow$More_optimal(direction_point)
```

In my heuristic function for Extract_Min(), I choose the minimum f(n) with formula:

$$\mathrm{f(n)} = g(n) + h(n)$$

g(n) is the current cost from start_point to node n

h(n) is the estimate of the cost from n to goal

f(n) estimate of the total cost of the solution path passing through n

Now consider $f^*(n) = g^*(n) + h^*(n)$

$g^*(n)$ cost of shortest path from start to node n

$h^*(n)$ actual cost of shortest path from n to goal

$f^*(n)$ actual cost of shortest path from start to goal through n

Proof heuristic:

Assuming that we have a graph G=(V,E), the weight w(u,v) is strictly positive and G is connected graph.

As we know, Dijkstra algorithm can find the shortest path by greedy algorithm or dynamic programming. For the graph G=(V,E). The time complexity as follow

$$\text{Time} = \Theta(\text{V}) \times T_{EXTRACT-MI} \;\; + \Theta(\text{E}) \times T_{DECREASE-KEY}$$

Q	$T_{EXTRACT-MIN}$	$T_{DECREASE-KEY}$	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\log V)$	$O(\log V)$	$O(E \log V)$
Fibonacci Heap	$O(\log V)$ Amortized	$O(1)$ Amortized	$O(E + V \log V)$ worst case

Space complexity is $O(V + E)$

In the A* algorithm

I use two algorithm to define $h(n)$

Algorithm 1: Euclidean Distance

$$H(x_n, y_n) = \sqrt{\left((x_n - x_g)^2 + (y_n - y_g)^2\right)}$$

Algorithm 2: Manhattan Distance

$$H(x_n, y_n) = |(x_n - x_g)| + |(y_n - y_g)|$$

Time complexity of worst case $=O(|E|)$

Space complexity of worst case $=O(|V|)$

Clearly, in A* algorithm, h(n) is estimate of the total cost of the solution path passing through n. less than $h^*(n)$.while, the h(n) in Disjkstra algorithm is the actual total cost of the solution path passing through n, equal to $h^*(n)$

Consequently, my algorithm is heuristic.

Proof optimal solution:

Compared with Disjkstra's algorithm, my algorithm may not find the shortest path from start node to goal. It means that the total weight of path may be higher than the shortest path. Therefore, my algorithm is an optimal path rather than exact shortest path, and my solution is an optimal solution rather than exact correct solution.

About the coordinate of each point:

Each point would be located in the leftdown corner of grid. If the point in one grid, we process its coordinate in order to the coordinate of this node in coordinate of leftdown corner of this grid.

About h(n):

In order to calculate h(n), I only focus on 8 directions for the current node (figure 1).

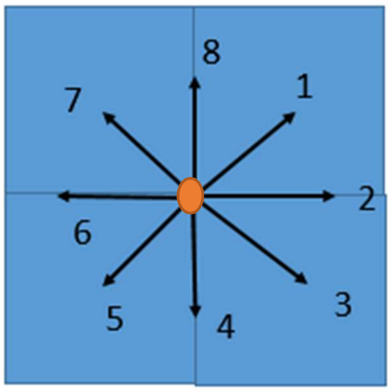


Figure 1

In those directions, only the direction in three special condition can be calculated (figure2).

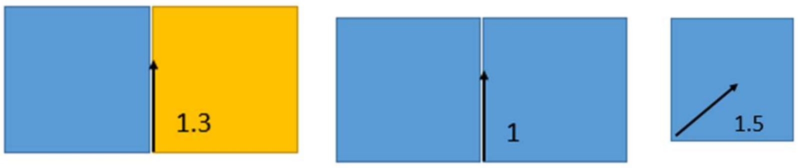


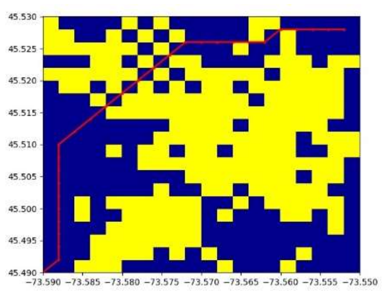
Figure 2

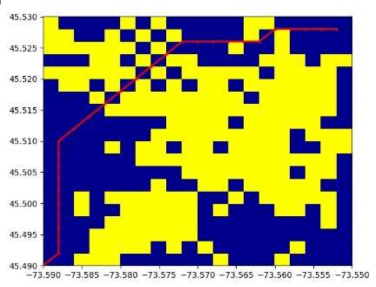
3. Description and justification of experiment result

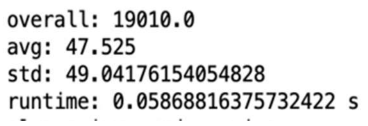
Base Map

Threshold : 0.5   grid size: 0.002*0.002   start point (-73.59,45.49) end point(-73.55,45.53)

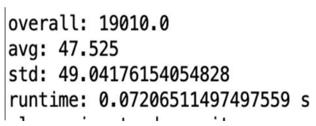
Algorithm 1: Manhattan Distance   Algorithm 2: Euclidean Distance







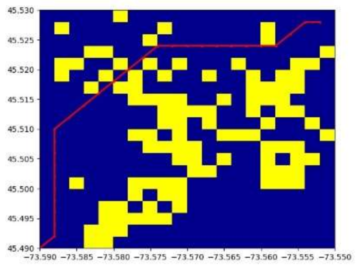
```
overall: 19010.0
avg: 47.525
std: 49.04176154054828
runtime: 0.05868816375732422 s
```

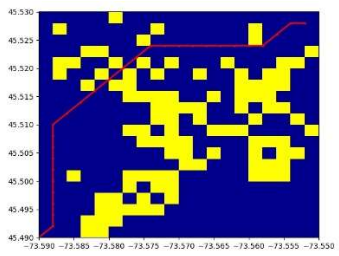


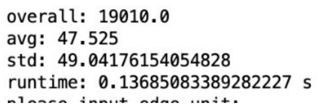
```
overall: 19010.0
avg: 47.525
std: 49.04176154054828
runtime: 0.07206511497497559 s
```

Threshold : 0.75 grid size: 0.002*0.002 start point (-73.59,45.49) end point(-73.55,45.53)

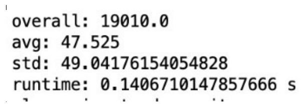
Algorithm 1: Manhattan Distance Algorithm 2: Euclidean Distance







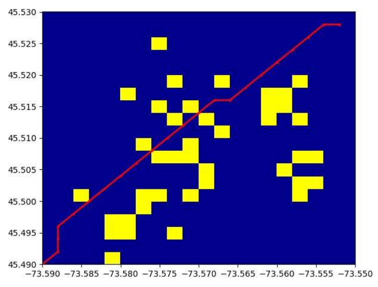
```
overall: 19010.0
avg: 47.525
std: 49.04176154054828
runtime: 0.13685083389282227 s
```

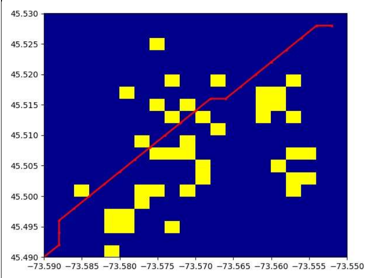


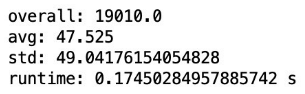
```
overall: 19010.0
avg: 47.525
std: 49.04176154054828
runtime: 0.1406710147857666 s
```

Threshold : 0.9 grid size: 0.002*0.002 start point (-73.59,45.49) end point(-73.55,45.53)

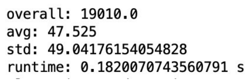
Algorithm 1: Manhattan Distance Algorithm 2: Euclidean Distance





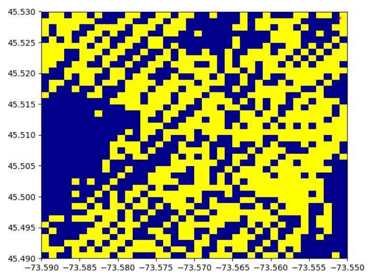


```
overall: 19010.0
avg: 47.525
std: 49.04176154054828
runtime: 0.17450284957885742 s
```



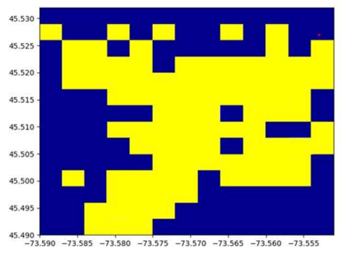
```
overall: 19010.0
avg: 47.525
std: 49.04176154054828
runtime: 0.1820070743560791 s
```

Threshold : 0.5 grid size: 0.001*0.001 start point (-73.59,45.49) end point(-73.55,45.53)



no path can be found

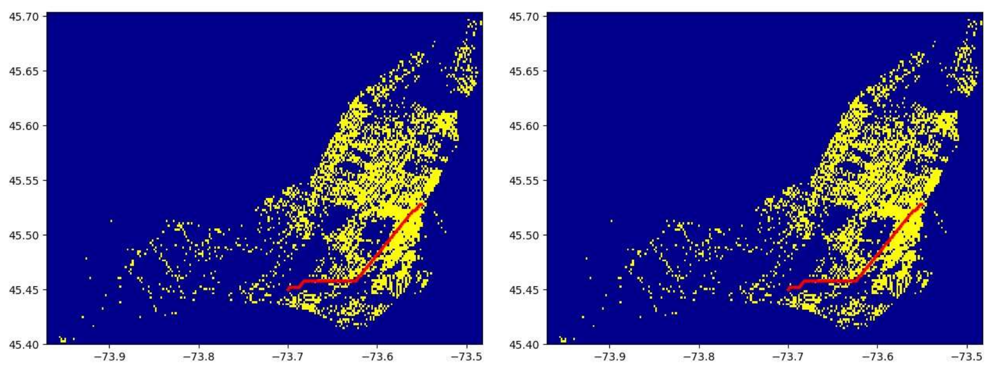
Threshold : 0.5   grid size: 0.003*0.003   start point (-73.59,45.49) end point(-73.55,45.53)



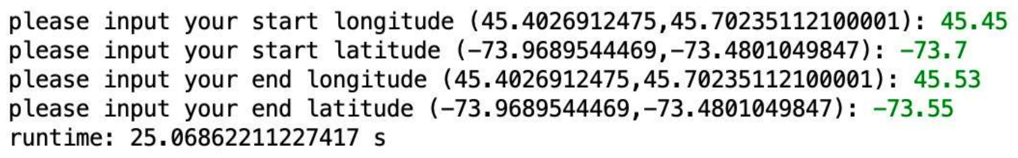
No path can be found

Montreal map

Threshold : 0.9   grid size: 0.002*0.002   Montreal map start point (-73.59,45.49) end point(-73.55,45.53)

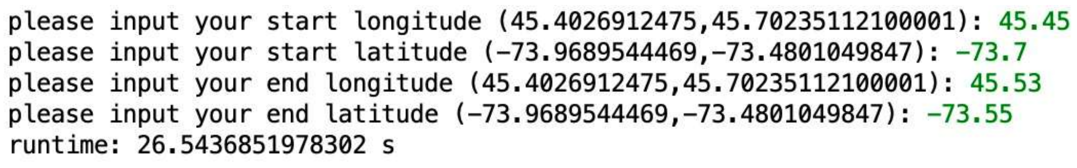


Algorithm 1: Manhattan Distance



```
please input your start longitude (45.4026912475,45.70235112100001): 45.45
please input your start latitude (-73.9689544469,-73.4801049847): -73.7
please input your end longitude (45.4026912475,45.70235112100001): 45.53
please input your end latitude (-73.9689544469,-73.4801049847): -73.55
runtime: 25.06862211227417 s
```

Algorithm 2: Euclidean Distance



```
please input your start longitude (45.4026912475,45.70235112100001): 45.45
please input your start latitude (-73.9689544469,-73.4801049847): -73.7
please input your end longitude (45.4026912475,45.70235112100001): 45.53
please input your end latitude (-73.9689544469,-73.4801049847): -73.55
runtime: 26.5436851978302 s
```

Experiment analysis

Summary for Two algorithm running time

Since the Euclidean Distance uses Arithmetic square root, Manhattan Distance only uses addition and minus. Therefore the running time Manhattan Distance for would be higher than Euclidean Distance. Moreover, the basemap only 20*20 grids, the time difference is not obvious enough. When we choose Montreal map, the difference would be larger.

Summary for different threshold

We can clearly see that with increasing of threshold, the running time is higher. We can explain the reason of this symptom is if the current node has more directions can choose, more calculating time the algorithm needs to cost.

Secondly, if the threshold is low, the median would be about approximate 0. It is hard to find a path. Therefore, we need to increase the value of threshold.

Summary for different grid size

We can see that finding a path does not straightly depend on the grid of size. No matter the grid size is large or small, we cannot find the optimal path.

Summary for the strength/weaknesses of the heuristics

Since my algorithm depends on A* algorithm, it is faster than using Dijkstra and uses best-first search to speed things up. If I need the algorithm to run in milliseconds, when does A* become the most prominent choice. It is complete and optimal. It is used to solve very complex problems.

However, this algorithm is complete if the branching factor if finite and every action has fixed cost. The speed execution of my algorithm is highly dependent on the accuracy of the heuristic algorithm that is used to compute h(n).

Summary for define the start point and end point

Most pathfinding algorithms from AI or Algorithms research are designed for arbitrary graphs rather than grid-based games. We'd like to find something that can take advantage of the nature of a game map. There are some things we consider common sense, but that algorithms don't understand. We know something about distances: in general, as two things get farther apart, it will take longer to move from one to the other, assuming there are no wormholes. We know something about directions: if your destination is to the east, the best path is more likely to be found by walking to the east than by walking to the west. On grids, we know something about symmetry: most of the time, moving north then east is the same as moving east then north.

4. Difficulties parts

Definition of h(n)

we try to define a formula of h(n) to avoid the g(n) is same for several nodes in the open list. If g(n) is same and minimum for several nodes in the open list, our algorithm would consider all those nodes. It could lead our algorithm looks like consider all node path, and the is not efficient.

Definition coordinate for each nodes

Since the start point and goal could be in the grid, on the edge of grid, or on the corner of grid, we process each node that located on the leftdown corner of current grid.

How to resolve boundary problem

Depending on the project description, the node cannot be located on the boundary. Therefore, we expand the map, the value of grid with boundary is -1. When we find a path, we would not consider this grid because its weight is negative value -1.

Some more optimal choices

1) Find alternatives in the open list. In the previous step, we used a loop of bubble sorting to sink the smallest number (one) to the end, but this method of finding does not seem to be fast. The open list is in a later iteration. It is built step by step, so every time we add something to the open list, we can do it according to the addition of the minimum binary heap, so that the last element of multiple is naturally the largest element (a) in the list.

(2) Insert a new element into the open list. The open list is the smallest binary heap, so the insertion element is based on the nature of the minimum binary heap.

(3) Delete the smallest element in the open list. The open list is the smallest binary heap, so deleting elements depends on the nature of the minimum binary heap.

(4) Delete the element specified in the open list. The f value of the element in the open list may change by recalculating the value of NewG, so we will delete the specified element first. This process can be implemented in a number of ways. Here, my example chooses to completely reorder the open list, which is not very efficient.

(5) Adding the current element to the close list, and merging the closed list in a convenient way to find it becomes easy to find.

(6) Search for the close list and improve the efficiency with the closed list structure.

5. Teamwork

It is an individual project, I complete all requirements for this project (coding and reporting).

6. Reference

https://en.wikipedia.org/wiki/A*_search_algorithm
https://blog.dominodatalab.com/interactive-dashboards-in-jupyter/
https://docs.python.org/3/tutorial/index.html
http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html