COMP 6721 Introduction to AI : Project 1

Subhannita Sarcar[1]

[1] 40059367 subhannitasarcar1994@gmail.com

1. Introduction

1.1. Problem definition

The problem presented for this project is not only unique but highly interesting since it has varied applications in the practical field. The objective is to determine an optimal route from a start and end point in a map containing information about crime reports at different locations. The business priority is to find the shortest path while avoiding the high crime rates within a timeframe. The definition of optimality and measure of efficiency of the path is left open for analysis.

The input is a shape file of a section of the map of Montreal containing "points" indicating the locations where a crime has been reported. An optional extension of the project is to work with a data sheet containing the locations and additional information regarding crime in the entire city of Montreal. The programming language used for this purpose is Python 3.7.3 and the environment is Jupiter notebook 6.0.1. The libraries used are mainly Numpy, pandas, Geopandas and Matplotlib.

1.2. Background research

The problem demands an informed state-space search algorithm and [1] discusses the pros and cons of several heuristic based path planning approaches. This paper classifies the path planning methods into two main categories - classical methods and heuristic based. Classical methods such as cell decomposition have not been found much useful in real applications because of their incapability to handle uncertainty and computationally intensive. Moreover, they focus more on finding any possible solution rather than the most optimal solution. Heuristic based methods such as neural networks and genetic algorithms are generally found to be more reliable since they are more dependent on the behaviour of the space which is suitable for accurately mapping changing and real time environments. [2].

It has been found that other than the search algorithm, the data formatting often plays a major role as well. An interesting approach is hierarchical approximate cell decomposition [3] where the search spaces is decomposed at successive levels of abstraction and each cell is labelled as either free (if completely outside obstacle area) or full (if completely inside the obstacle are) or mixed (if neither free or full) until one of the termination criteria is met :

- a "chain" of adjacent cells are found from the start to end point that are completely free of obstacles
- It is concluded that no solution exists

- The size of mixed cells drops below a pre specified size

The report is structured as follows : preparation of the data into usable information, analysis of the data to determine the relationships between each data point, description of the heuristic algorithms used and the reasoning behind them, analysis of the results.

2. Data Analysis

There are two sets of input data.

1. D1 : A shape file of the map between (-73.59, 45.49), (-73.55, 45.49), (-73.55, 45.53), (-73.59, 45.53) with crime points as shown in Fig 1a.
2. D2 : A comma separated file with crime information reported all over Montreal and arbitrary start and end locations. The corresponding map is shown in Fig 1b.

The first step is to convert the data points of the map into a location that is easily accessible by an algorithm. The obvious conclusion would be to treat the entire map like a 2D array where each cell covers a fixed area of the map. Hence, any geolocation can be translated as a reference to the cell in the 2D array that contains the location. For the sake of simplicity, throughout the course of this project, we will refer to the location of any point inside a cell as the bottom-left corner of the cell.

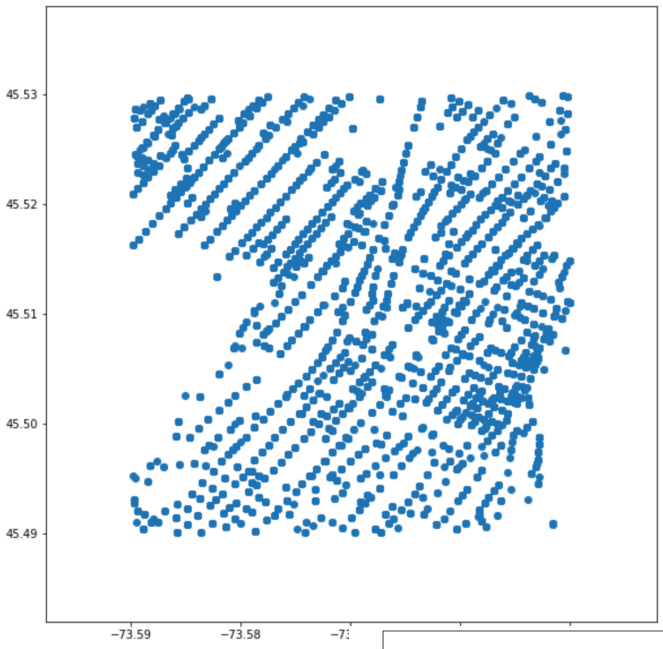

Fig.1a

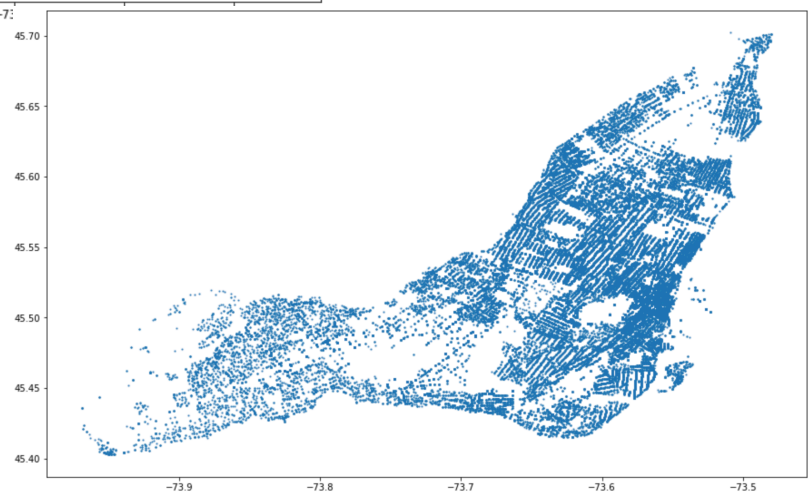

Fig.1b

Fig. 1. Maps of Montreal with blue markers indicating locations of crime reports.
a. D1
b. D2

Intuitively, a finer cell size would result in higher efficiency of the geolocation translation.

In case of D1, this is proven correct. A smaller cell size maps a geolocation to x-y coordinates of the generated grid (bottom-left corner of the cell containing the point) with more accuracy. A cell size of 0.001 by 0.001 px is chosen. Also, the mean and median of the dataset changes by the same amount, thereby indicating that a proportional threshold will not make any difference in the classification of the dataset into high-crime and low-crime cells.

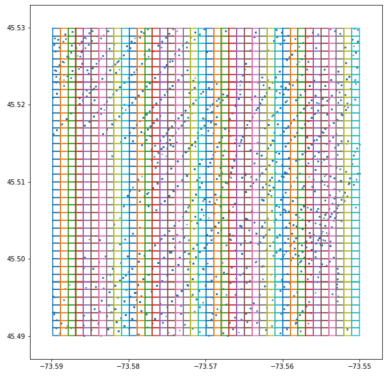

Fig.2a

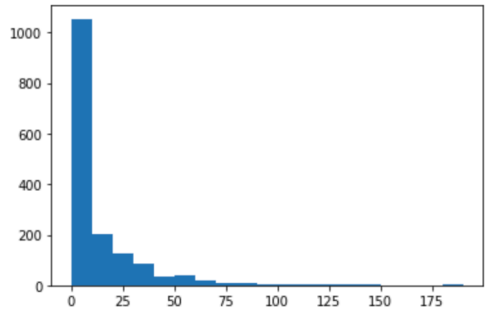

Fig.2b

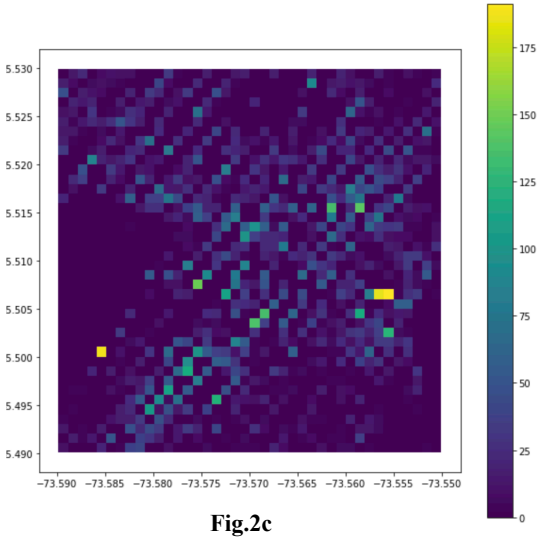

Fig.2c

Fig. 2. Data formatting and analysis for D1
a. Superimposing grid of cell size 0.001 by 0.001 px over map
b. Histogram plot of crime points in D1
c. Crime distribution plot

The following choice of thresholds have been found to construct a suitable grid for evaluation of a heuristic algorithm :

Table 1. Thresholds for D1

Thresholds	Formulae	Value
T1D1	m` + 0.1 (max - m`)	20
T2D1	m` + 0.2 (max - m`)	39
T3D1	m` + 0.3 (max - m`)	58

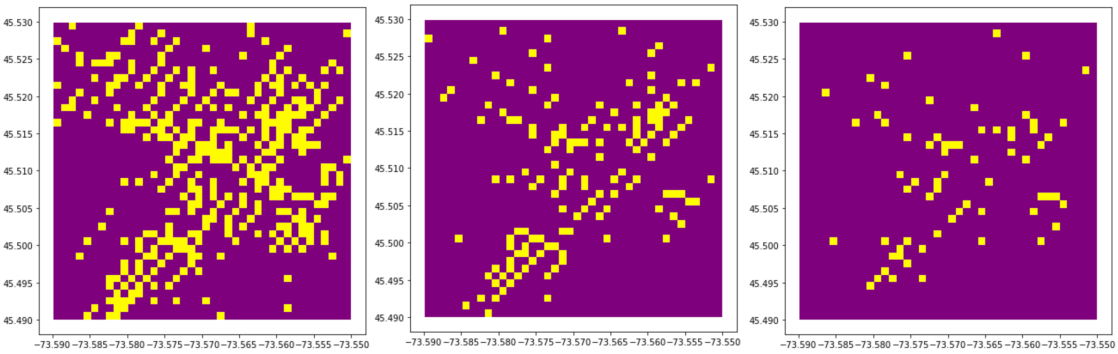

Fig. 3. Map of D1 for low to high thresholds (from left to right) where the purple areas are free of obstacles whereas yellow are obstacles.

However, in case of D2, a smaller cell size generates large number of cells with almost equal number of crime points. Therefore, in order to attain a dataset with a clear boundary between high and low crime cells, a cell size of 0.003 by 0.003 px has been chosen. The histogram plot for D2 (Fig 4a) has not been proven to be much helpful since it takes into account the vast number of cells that have 0 crime. This could have been avoided by truncating the cells that fall outside the actual city boundaries.

However, the crime distribution plot (Fig 4b) shows an approximate number of crime points in each area.

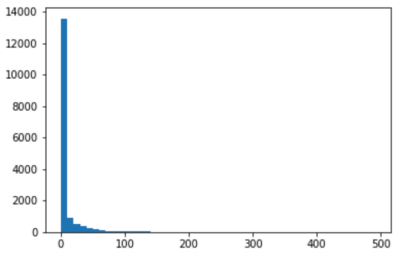

Fig. 4a. Histogram plot of crime points in D2

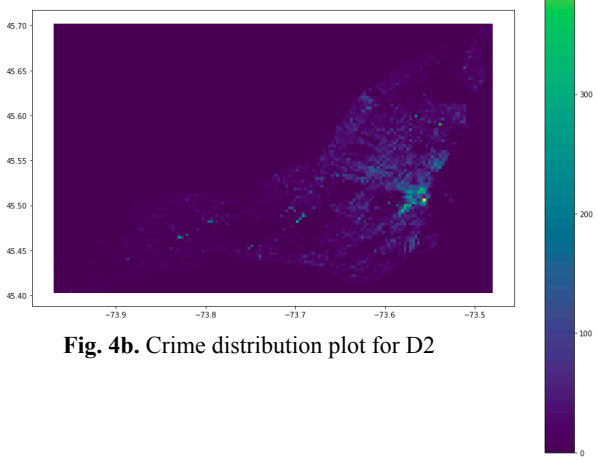

Fig. 4b. Crime distribution plot for D2

This has been useful in choosing the following three thresholds after some experimentation:

Table 2. Thresholds for D2

Thresholds	Value
T1D2	25
T2D2	50
T3D2	75

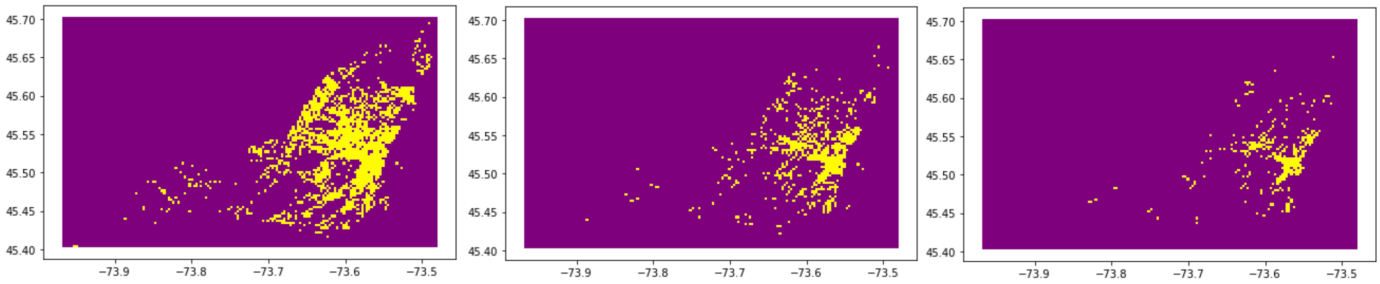

Fig. 5. Map of D2 for low to high thresholds (from left to right) where the purple areas are free of obstacles whereas yellow are obstacles.

3. Experiment Setup

3.1. A* search algorithm

A* search algorithm is one of the fastest and popular algorithms used for graph traversals. It is similar to Greedy-Best-First-Search as it uses a heuristic to guide itself, but different in the sense that it takes the cost or distance already traversed into account.

Since traversal is along the edges, we can refer to each intersection or corner of a cell as a node. The starting and destination nodes are the bottom left corner of the cell containing the starting and destination locations. Therefore, the map can be extended to include a layer of cells on top and on the right to designate the intersections at the edge. Starting from the source node, there are 8 maximum possible nodes to be traversed. Among these, the node with the minimum heuristic will be selected as the next node. The heuristic f(n) to be used for selecting the next node n can be a combination of the distance from source node to the current node n (g(n)), Manhattan distance from current node n to destination (h(n)) and cost associated with path due to crime (i(n)).

$$f(n) = h(n) + g(n) + i(n) \quad (1)$$

The measure of i(n) is calculated as follows :

Table 3. Cost for traversal through different types of blocks

Path	i(n)
Diagonally through an obstacle cell	Not possible
Diagonally through a free cell	1.5
Straight between two free cells	1
Straight between one free and one obstacle cell	1.3
Straight between two obstacle cells	Not possible
Along the edge of map	Not possible

It must be noted that this algorithm does not guarantee the shortest path if h(n) is more than the actual cost to reach goal, i.e., the manhattan distance is more than the cost to traverse the actual path. This will almost never occur in this context because of the following two scenarios:

1. There exists no high crime blocks between the current node n and destination and therefore, it is natural to cut diagonally through all the cells in between. In this case, h(n) is a 100% accurate estimation of the distance between n and goal. Therefore, A* will only follow best path, thereby making it very fast.
2. There exists high crime blocks between the current node n and destination and therefore, the actual path covered will always be more than the manhattan distance since f(n) also constitutes i(n) which is a measure of cost due to crime.

The practical performance of an A* search algorithm largely depends on the quality of its heuristic function as it allows the algorithm to leave many of the branching nodes unexplored, which would have been otherwise expanded for uninformed search. It has been shown [4] that A* is more optimally efficient, i.e., expands a lower set of nodes, as compared to other similar algorithms, provided the heuristics used are both admissible(guaranteed to return shortest path) and consistent.

Algorithm 1. A* search algorithm

Input : 2D array *maze* where each cell is 0 (free) or 1 (obstacle), *start* and *end* locations

Output : optimal path from *startNode* to *endNode* and associated *cost*

1: *openList* := list of nodes visited but not expanded
2: *closedList* := list of nodes visited and expanded
3: *startNode.g* = *startNode.h* = *startNode.i* = *startNode.f* = 0
4: *openList*.push(*startNode*)
5: **while** *openList* is not empty :
6: *currentNode* := *openList*.remove(node with minimum f)
7: **for** each node *n* in 8 neighbour nodes of *currentNode* :
8: **if** *n* is outside map or *n* is on straight path between 2 obstacles or *n* is on straight path along edge or *n* on diagonal path through obstacle :
9: **continue**
10: **else** :
11: *n.parent* := *currentNode*
12: *n.g* := *currentNode.g* + 1
13: *n.h* := Manhatten distance(*n,endNode*)
14: *n.i* := crimeWeightage(*currentNode,n*)
15: *n.f* := *n.g* + *n.h* + *n.i*
16: **if** *n* is *endNode* :
17: **return** path, cost
18: **if** *n* is in *openList* with lower *f* or *n* is in *closedList* with lower *f*:
19: **continue**
20: **else** :

21: *openList*.push(*n*)
22: **end for**
23: *closedList*.push(*currentNode*)
24: **end while**

3.2. Dijkstra's algorithm

The Dijkstra's algorithm is a very popular solution to single source shortest path problem for weighted (nonnegative) and connected graphs. The Dijkstra's algorithm can be easily proven to give the optimal path because of the following two reasons :

1. The subset of any shortest path is itself a shortest path
2. Considering three nodes as u, v and x, the shortest path between vertices (u,v) is always less than or equal to the sum of the shortest distances between (u,x) and (x,v). This is maintained since the distance to a node from the previous node is always the sum of the distance of the previous node from source and the weight of the edge connecting the nodes. Also, if the distance marked on an existing node is found to be larger than the distance found during an iteration, it is modified to the lower distance and the shortest path to that node is also modified consequently.

The traversal and crime weightage is similar to the previous algorithm. The original Dijkstra's algorithm was found to exceed the runtime requirements, therefore, a modification to the heuristic was introduced : while choosing the best node among the unvisited nodes, priority was given to the node having lowest distance to the source. However, if multiple nodes were found to be equidistant to the source, then the node that is closest to the destination (according to Manhattan distance) is chosen as the current node in the next iteration.

Algorithm 2. Dijkstra's algorithm

Input : 2D array *maze* where each cell is 0 (free) or 1 (obstacle), *start* and *end* locations
Output : optimal path from *startNode* to *endNode* and associated *cost*
1: *unvisitedList* := list of nodes visited but not expanded
2: *visitedList* := list of nodes visited and expanded
3: *startNode.f* = 0
4: *openList*.push(*startNode*)
5: **while** *visitedList* does not contain all nodes :
6: **if** *unvisitedList* is empty :
7: **return** no path found
8: *currentNode* := *openList*.remove(node with minimum f)
9: **if** *currentNode* is *endNode* :
10: **return** path, cost
11: *neighbours* := list of valid neighbouring nodes of *currentNode* that are on map and can be navigated to (considering obstacles)
12: **for** each node *n* in *neighbours* :

```
13:      n.f := currentNode.f + crimeWeightage(currentNode,n)
14:          if n is in visitedList or n is in unvistedList with lower f:
15:              continue
16:          else if n is in unvistedList with higher f:
17:              unvisitedNode = node in unvisitedList in same position as n
18:              unvisited.parent := currentNode
19:              unvisited.f := n.f
20:          else:
21:              unvistedList.push(n)
22:      end for
22:      closedList.push(currentNode)
24:  end while
```

4. Experiment Results

4.1. Dataset D1

A* search algorithm performs remarkably well on this dataset. The change in thresholds has minimal effect on the runtime performance. Optimal efficiency, i.e., the set of nodes expanded is seen to be high since the algorithm not only moves in the right direction of the end node, but also finds the shortest path between each pair of nodes with each iteration. It was noted that the heuristic i(n) which is a measure of the crime rate in the path may not be having much effect since the order of the other heuristics (such as Manhattan distance) is not less than 10 times. Therefore, an attempt was made by introducing a crime weightage which could be multiplied with i(n).

$$i`(n) = C\ i(n) \quad (2)$$

$$f(n) = g(n) + h(n) + i`(n) \quad (3)$$

where C = crime weightage

In order to determine an effective value of C, the magnitudes of g and h need to be analyzed throughout the execution. We can assume that in the beginning, h(n) plays a more significant role in the heuristic since it is a sum of two squares whereas g systematically increases by 1 with each traversal. Since the start and end nodes in this case are the two diagonally opposite corners of the grid, h(n) ranges from 1 to 98 and g(n) ranges roughly from 1 to 56.4. Hence, we can experiment with a crime weightage of 20 to see a visible difference. This analysis is worthwhile to be conducted only for the lowest threshold of crime rate since the other two thresholds already indicate a clear optimal path between the start and destination nodes. The experiment resulted in an average difference of 8% in the number of steps taken in optimal path.

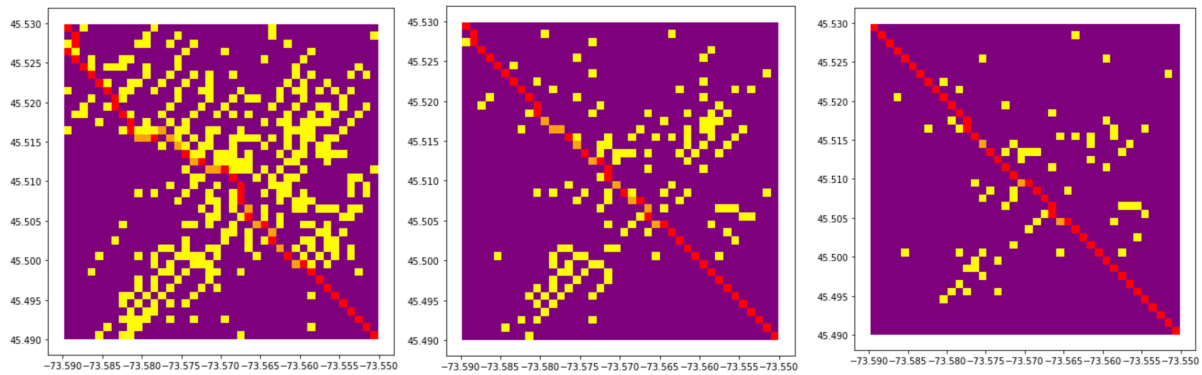

Fig. 6. Map of D1 showing optimal path as found by A* algorithm by red cells from high to low thresholds (from left to right). It must be noted that the red cells indicate that traversal was made through the bottom-left corner of that cell, and not diagonally though the cell

Dijkstra's algorithm took significantly more time to find the optimal path on the same dataset for reasons mentioned in the previous section. An interesting observation is that Dijkstra's search always finds a path with lower cost than A* because it surveys all the unvisited nodes before selecting the node to expand in the next iteration.

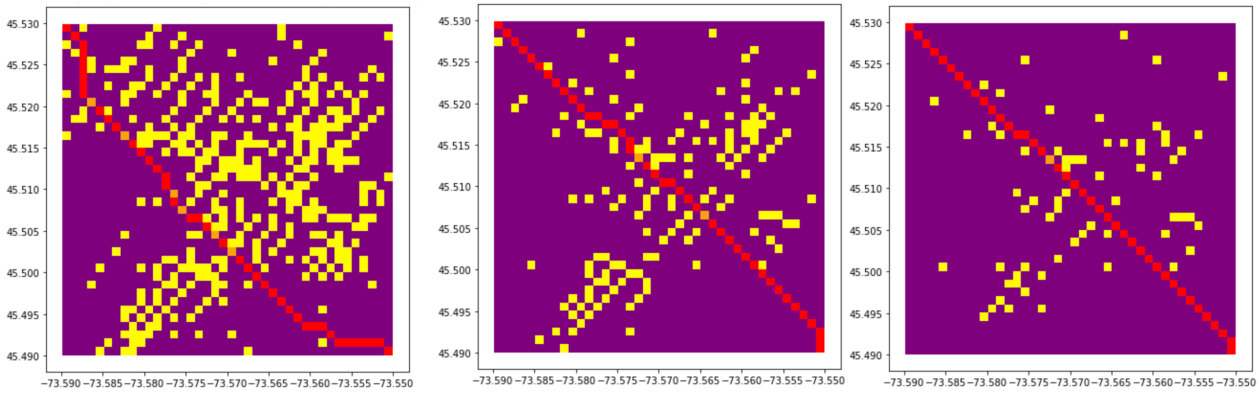

Fig. 7. Map of D1 showing optimal path as found by Djikstra's algorithm by red cells from high to low thresholds (from left to right). It must be noted that the red cells indicate that traversal was made through the bottom-left corner of that cell, and not diagonally though the cell

The relative cost was clearly in proportion to the number of obstacles encountered.

4.2. Dataset D2

The results for experiments on D1 is repeated on a larger scale for D2. The start and end locations are chosen to be (45.586635, -73.567552) and (45.498342, -73.701500) since the path would necessarily cover a high crime density area. The observations remain same and reasons behind them are more coherent. A* algorithm is successively able to find a sufficiently optimal path within time out of 10 seconds.

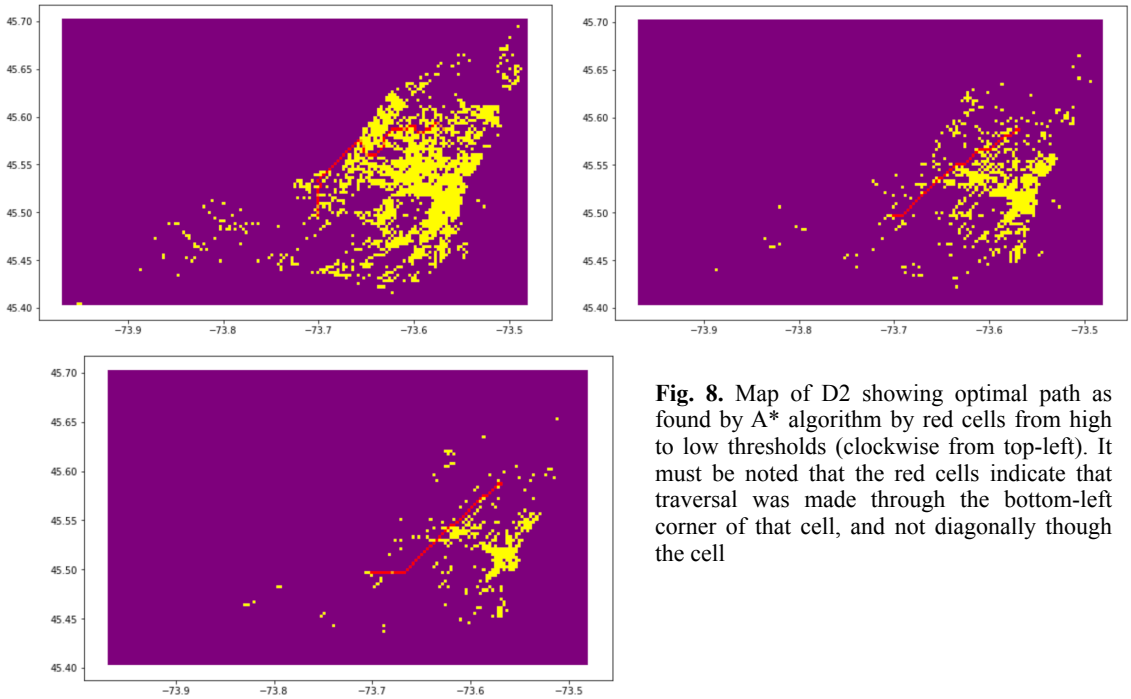

Fig. 8. Map of D2 showing optimal path as found by A* algorithm by red cells from high to low thresholds (clockwise from top-left). It must be noted that the red cells indicate that traversal was made through the bottom-left corner of that cell, and not diagonally though the cell

Dijkstra's algorithm takes almost 10 times the needed in the former algorithm, but is always able to find a better path.

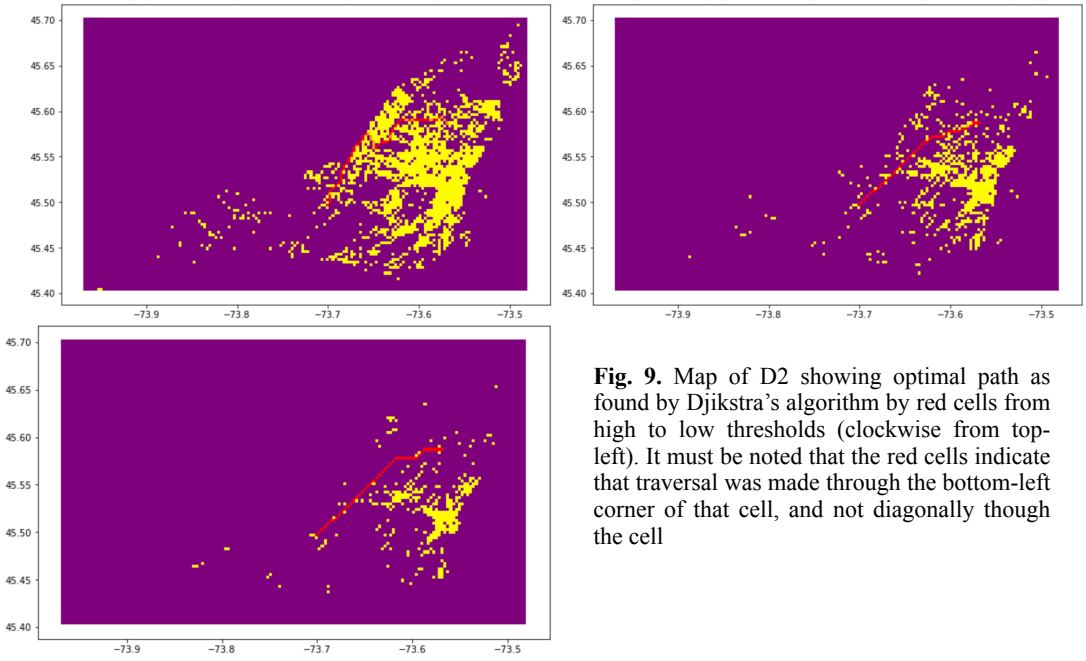

Fig. 9. Map of D2 showing optimal path as found by Djikstra's algorithm by red cells from high to low thresholds (clockwise from top-left). It must be noted that the red cells indicate that traversal was made through the bottom-left corner of that cell, and not diagonally though the cell

5. Conclusion

The major differences between the two algorithms has been found to lie in the following areas -

1. A* algorithm expands only the best child node among the immediate children and goes on to expand from there. On the other hand, Dijkstra's algorithm expands the node with best heuristic among all the unvisited nodes and does not construct the desired shortest path with each iteration, but rather constructs the path by re-tracing once the destination node is found.
2. A* algorithm has been observed to find a sufficiently optimal path for this problem statement since the heuristic that accounts for the distance between the current node and destination node is a good estimation of how close the algorithm is to finding the shortest path. Dijkstra's algorithm has given preference to the distance from the source node which is nothing other than the combination of distance and crime weightage.
3. Dijkstra's algorithm always finds the path that is most optimal since it marks the shortest distance to each node before retracing a path from the destination back to the starting node.

There is ample scope to improve the results better. One such direction would be cleaning the data so as not to include the vast areas in the map that have no terrain in reality. This problem statement can be extended to include the constraints of roads and the type of vehicle which would further impose restrictions on the possible route choices.

References

1. Ahmadi, S., Kebriaei, H. and Moradi, H. (2018). Constrained coverage path planning: evolutionary and classical approaches. *Robotica*, 36(6), pp.904-924.
2. Gordon Cheng , Alexander Zelinsky, “A physically grounded search in a behavior based robot”. In Proc. Eighth Australian Joint Conference on Artificial Intelligence, pp. 547-554, 1995.
3. D. Zhu and J.-C. Latombe, “New heuristic algorithms for efficient hierarchical path planning,” IEEE Transactions on Robotics and Automation, vol. 7, no. 1, pp. 9–20, 1991.
4. Rina, Dechter, first, and , “Generalized best-first search strategies and the optimality af A*”, vol. 32, Jul. 1985.
5. LNCS Homepage, http://www.springer.com/lncs, last accessed 2016/11/21.