Montreal Crime Analytics (Project 1)

Dhwani Sondhi

40083894 d_sondhi@encs.concordia.ca

1. Project Description

In this project, we need to analyze the crime data of the center of Montreal downtown area given in 'crime_dt.shp'. The data is to be stored in a grid and accumulated using statistics such as total count, mean or standard deviation. An input for threshold will be taken from the user. Depending on the user input, we need to define the high and low crime rate blocks. The high crime rate blocks are represented in 'Yellow' color(high risk) and low crime rate blocks are represented in 'Blue' color(low risk). Using this map, we need to implement a heuristic search algorithm to find an optimal path between two given coordinates on the map, considering yellow blocks as obstacles. The program also needs to display the mean, median and standard deviation of the data. This project will use two different heuristics algorithms and compare their efficiency and output. This project is made using Python programming language and related libraries. The libraries used in the project are sys, heapq, shapely, geopandas, pandas, matplotlib and numpy.

2. Proposed Approach

2.1 Data representation in a grid

GeoPandas library was used to represent the geospatial data given. This library is very helpful when applying operations on different geometrical objects. It has two main data structures: GeoSeries and GeoDataFrame. Here, I have used GeoDataFrame as the data structure which contains GeoSeries which is a set of shapes(like Points, Lines and Polygons). All the spatial operations applied on a GeoDataFrame, will always lay the effect on the 'geometry' column(this column can have different names).

The data is extracted from 'crime_dt.shp' into a GeoDataFrame. This GeoDataFrame will contain a set of Point shapely objects which are further stored in a grid. To get the edges of the grid, the 'total_bounds' function is applied on to the geometry attribute of the GeoDataFrame stored.

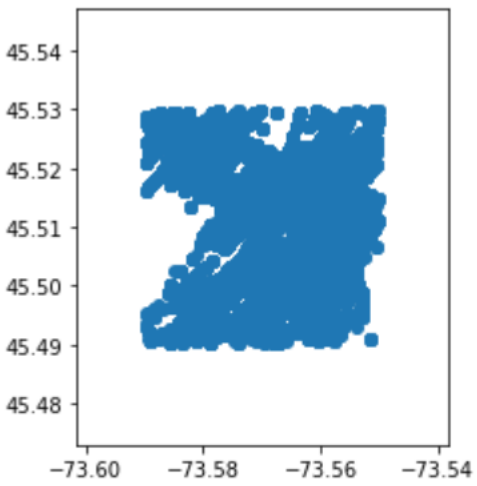


Figure1: Input data(plotted)

	CATEGORIE	QUART	timestamp	geometry
0	Vols qualifiés	soir	2017-08-02 00:00:00	POINT (-73.55952 45.51703)
1	Introduction	jour	2017-08-03 00:00:00	POINT (-73.55510 45.52185)
2	Méfait	jour	2017-08-23 00:00:00	POINT (-73.58359 45.49012)
3	Vol dans / sur véhicule à moteur	jour	2018-09-05 00:00:00	POINT (-73.58794 45.52297)
4	Vol dans / sur véhicule à moteur	jour	2017-08-25 00:00:00	POINT (-73.55280 45.52430)
...	...	...	...	
19005	Introduction	jour	2019-03-29 00:00:00	
19006	Vol dans / sur véhicule à moteur	jour	2019-06-12 00:00:00	
19007	Vol de véhicule à moteur	soir	2019-06-20 00:00:00	
19008	Introduction	soir	2019-01-09 00:00:00	
19009	Vol dans / sur véhicule à moteur	jour	2019-05-13 00:00:00	

Figure 2: Input data(printed)

To create a grid, arrays for both x and y axis are made using ‘arange’ function from numpy library. This function creates the arrays depending on the given grid cell size(step_size). This size can be changed as per the user input. Smaller the grid cell gives better output. Thus, I have taken 0.002X0.002 as the grid size for now. These arrays are iterated to create a list of polygons which are stored with their indexes in a new GeoDataFrame. Through this, I have converted the given map into a set of polygons which represent the grid where each gird cell is a polygon.

Now, to differentiate which grid cell as a high and low crime rate block, we must determine the count of the crime data points which lie in each cell. Until now, the GeoDataFrames used have two geometrical objects: Polygons and Points. We need the count of the Points that lie in each Polygon which can easily be done using ‘sjoin’ function. I tried using ‘contains_points’ and ‘contains’ functions also, but they were taking a lot of time in creating this count whereas ‘sjoin’ was taking less time. The ‘sjoin’ function provides a spatial join in which the geometrical objects are joined/merged depending on their spatial relationship to one another. This function when applied to the above two GeoDataFrames give the points that lie on each polygon/grid cell. Further, ‘group_by’ and ‘agg’ functions are used to get a count with respect to each cell.

	a	count	geometry
0	0	0	POLYGON ((-73.58983 45.49007, -73.58913 45.490...
1	1	8	POLYGON ((-73.58913 45.49007, -73.58843 45.490...
2	2	0	POLYGON ((-73.58843 45.49007, -73.58773 45.490...
3	3	0	POLYGON ((-73.58773 45.49007, -73.58703 45.490...
4	4	0	POLYGON ((-73.58703 45.49007, -73.58633 45.490...
...	...	...	...
3244	3244	7	POLYGON ((-73.55343 45.52927, -73.55273 45.529...
3245	3245	0	POLYGON ((-73.55273 45.52927, -73.55203 45.529...
3246	3246	0	POLYGON ((-73.55203 45.52927, -73.55133 45.529...
3247	3247	7	POLYGON ((-73.55133 45.52927, -73.55063 45.529...
3248	3248	3	POLYGON ((-73.55063 45.52927, -73.54993 45.529...

Figure 3: Count of points per polygon

Now, the input from the user is taken for threshold percentage. This percentage when increased, decreases the number of blocked cells. Here, I have sorted the calculated counts to determine the value for the threshold. When sorted in an ascending order, we can determine the index using the following formula, for a given threshold percentage and number of polygons or grid cells :

*index = (threshold percentage/100) * count of grid cells* (1)

Using this formula, we get the index of the considered threshold value. And the value or the count saved on this index is used as the value to determine high and low crime rate areas. If the count of the cell or polygon is greater than or equal to the threshold value, then color it as 'Yellow' otherwise as 'Blue'.

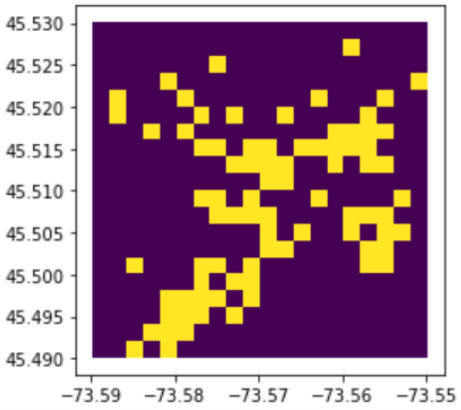


Figure 4: Example of 0.002X0.002 grid in 80% threshold

This data is further converted into a graph, where each grid cell is represented as a node by saving the left down coordinates of each grid cell in a list. The edges can be drawn further to other nodes or coordinates by applying following rules:

- No edge can pass through the blocked(or Yellow) areas.
- Any edge between two blocked(or Yellow) areas and or diagonals through them are not allowed.
- Any edge between one blocked(or Yellow) and other unblocked(or Blue) areas has a cost 1.3.
- Any edge between two unblocked(or Blue) areas has a cost 1.
- Any diagonal through an unblocked(or Blue) area has a cost 1.5.
- The edges on the boundary are not accessible.

Using these rules, a graph is created. This graph is saved as an array for neighbors of each nodes(neighbors per polygon index).

2.2 Heuristic function

The heuristic search is done to analyze the best alternative depending on the selection function applied. This selection function also known as a heuristic function gives a score or evaluation of a node as per the goal node. In our case, we can move towards 8 neighbors if the blocks are 'Blue'. Thus, here a heuristic based on 'Chebyshev distance' is used. Following function is used which takes the x value is the x coordinate and y as y coordinate of the node:

hValue(node) = max{ abs(node.x - goal.x),
abs(node.y - goal.y)} (2)

Here, the heuristic value we get is the maximum of the absolute values of difference in both x and y coordinates.

2.3 Heuristic Algorithm: Best-First Search

Best-First Search unlike the BFS, DFS, Uniform Cost and Iterative deepening search, makes an informed search. This selects the node or the alternative path to be explored based on a heuristic function to find the solution in the search space. This algorithm is based on a priority queue which provides the node with the lowest heuristic to be explored further regardless of its position in the search space. To implement this, I have used following data structures:

- Open List: This list contains all the nodes that needs to be explored further. This list is explored every time to give the node with smallest heuristic value.
- Closed List: This list contains all the nodes that have been visited or explored and whose children are considered for the open list.
- hValue list: This list contains the hValue corresponding to each polygon or node index.
- Parents list: This list contains the parents corresponding to each polygon or node index. It is -1 for source node.

Initially the open list consists of the source node and a loop is run till either we reach the goal node, or the open list is empty. For each loop, we get the node from the open list with lowest heuristic to the goal node and we run it through the Goal Test function to check if it is the goal node. If it is, the loop breaks, otherwise, we take children of this node and add them to the last of the list if there are already not present in the open or the closed list. The parents array indexes of the neighbors are given the value of node index. The visited or explored node is then added to the closed list.

As we exit from the loop, we run another loop from the goal to the parent to explore the path found. If any of the 'parents' array value for the node being explored is equal to '-1', this means we have not found the path, and this is shown in the output.

This search is quite fast and finds the output in a very small span of time. Following are some outputs for this algorithm:

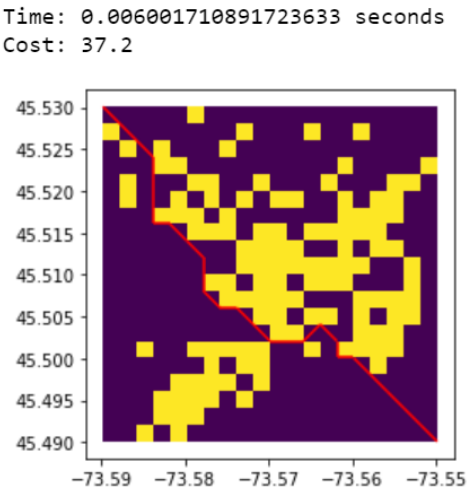


Figure 5: For grid Cell size= 0.002X0.002 and threshold percentage: 70

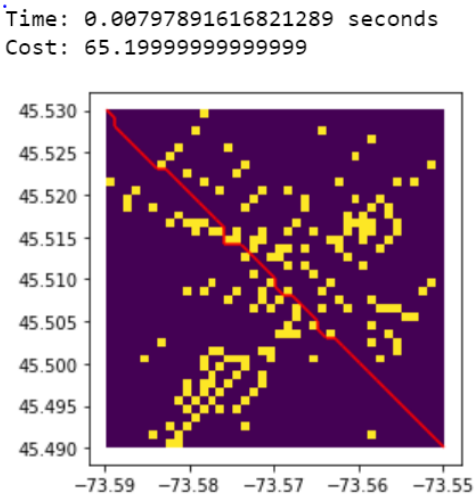


Figure 6: For grid Cell size= 0.001X0.001 and threshold percentage: 90

2.4 Heuristic Algorithm: A* algorithm

The A* algorithm is similar to the Best-first search that using a priority queue that instead of returning a node index with least heuristic value, returns a node index with least 'f(node)' where

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

where h(n) is the estimated cost for the path from the node n to the goal state
and g(n) is the actual cost for the path from the source node to the node n

Here, h(n) is admissible that it never overestimates the actual cost and so is A* that is, it always finds the lowest cost path to the goal node.

In my data structures, I have replaced the 'hValue list' with 'fValue list' that stores the f(n) values for each node index. Also, I have converted open list to a minimum heap using 'heapify' function of 'heapq' library. This heap takes less time to extract the lowest fValued node index. And an additional list named 'distance' for storing the g(n) for each node index. Following are some outputs for this algorithm:

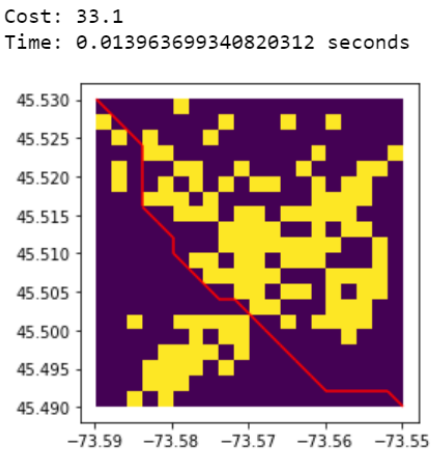


Figure 7: For grid Cell size= 0.002X0.002 and threshold percentage: 70

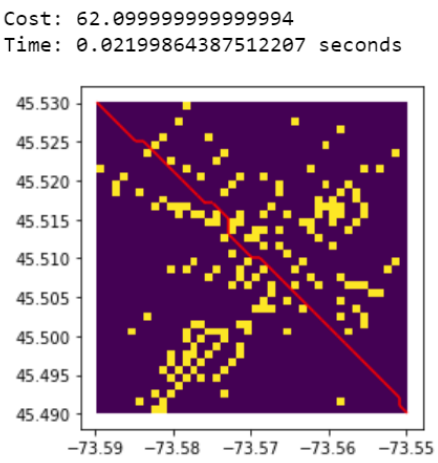
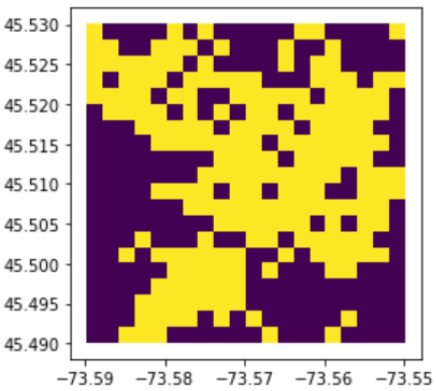


Figure 8: For grid Cell size= 0.001X0.001 and threshold percentage: 90

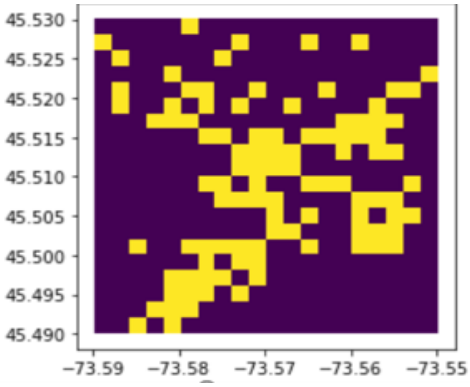
3. Results

3.1 Data represented using threshold

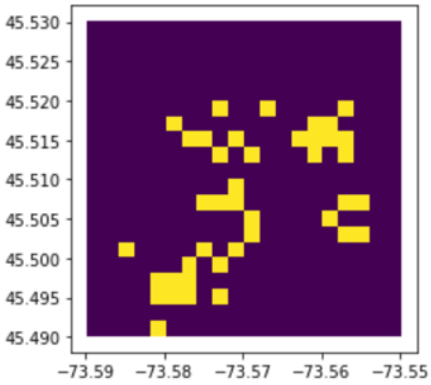
The threshold determines the blocked cells in a grid. As the threshold increases, the number of blocked cells is decreased(that is the yellow part is decreased). The results also rely on the grid cell size. Smaller the grid cell size, gives better exploration. Following are some outputs:



(a) For threshold: 50%



(b) For threshold :75%



(c) For threshold :90%

Figure 9: For grid cell size as 0.002 x 0.002

3.2 Comparison of two heuristic algorithms

There are two types of searches: Informed and uninformed. The uninformed is systematic and takes more time to explore. The informed search on the other hand, uses the selection or heuristics functions for searching the best solution. Thus, having a large search space, it was better to use the informed searches. In this, hill climbing could have been an option, but it has a disadvantage that it only explores its neighbors and move forward with the first neighbor that has better heuristic than the current one.

In this case, best-first search gives a better search by saving the next steps in a priority queue or the list(in ascending order) to get the node with smallest heuristic value. But, with an advantage that it is fast, it also has a disadvantage that it does not guarantee the best or optimal solution in the first attempt. The main reason is that it only compares the heuristics and not the cost from the initial node. In this case, if a node has two neighbors who has same heuristic values but reaching the first one is costlier than the second one, it goes with first one as it is inserted first in the queue. In this case, the cost increases to reach the goal, and a cheaper path could have been found if cost would have been considered. Also, heuristics can go wrong, so relying on only these can be risky.

In this scenario, A* algorithm gives an edge by including the cost with the heuristics and using a f(n) to rely on. When we compare Fig. 5 to Fig. 7 and Fig. 6 to Fig. 8, we see that even with the same input, the cost of A* algorithm is lower as it is optimal and finds the lowest cost path in first attempt unlike Best-first search which only relies on heuristic values.

4. Difficulties Faced

I faced two major difficulties:

- Being a beginner to the GeoPandas and python, it took me time to understand the libraries and in-built functions. In start, I read the GeoPandas description and read various other blogs to understand the working to these data structures. I tried various methods and came over to the conclusion to use spatial join for counting the points.

- Second difficultly was the graph representation, initially I made all the coordinates of each polygon as the nodes and paths between them as edges of a graph. But it was taking a lot of time to process them for smaller grid cells. For that, I read various blogs(see URL [9] and URL [3]) which guided that I should instead take the whole grid cell as a node and make the graph accordingly.

5. Conclusion

We were able to convert the given crime data into grid with 'Yellow' blocked blocks and 'Blue' unblocked blocks. Further, we were able to find the paths using different heuristic algorithms and formed a comparison between them. Also, from the above figures we noticed that the when we decreased the grid size, it gave a more intuitive image and description of data and a better path could be found.

There were some cases where there was no path found from the source node to goal node. Following figure shows the example of one:

```
Due to blocks, no path is found. Please change the map and try again
```

Figure 10: For Grid cell size:0.002 x 0.002 and threshold value: 50%

Also, for plotting this path onto the graph, I took 'LineString' function which joins two points and further, converted this to a GeoDataFrame. This GeoDataFrame was plotted with the above GeoDataFrame using 'subplots' function of matplotlib.pyplot.

References:

1. (n.d.). Retrieved from http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html.
2. Data Structures¶. (n.d.). Retrieved from http://geopandas.org/data_structures.html.
3. (n.d.). Retrieved from http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html.
4. How to join (merge) data frames (inner, outer, right, left join) in pandas python. (n.d.). Retrieved from http://www.datasciencemadesimple.com/join-merge-data-frames-pandas-python/.
5. Lee, A. (2019, August 15). Why And How To Use Merge With Pandas in Python. Retrieved from https://towardsdatascience.com/why-and-how-to-use-merge-with-pandas-in-python-548600f7e738.
6. (n.d.). Retrieved from http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html.
7. Choosing Colormaps in Matplotlib¶. (n.d.). Retrieved from https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html.
8. 8.4. heapq - Heap queue algorithm¶. (n.d.). Retrieved from https://docs.python.org/2/library/heapq.html.
9. (n.d.). Retrieved from https://www.redblobgames.com/pathfinding/grids/graphs.html.
10. Indexing and selecting data¶. (n.d.). Retrieved from https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html.
11. Spatial join¶. (n.d.). Retrieved from https://automating-gis-processes.github.io/CSC18/lessons/L4/spatial-join.html.
12. Creating a GeoDataFrame from a DataFrame with coordinates¶. (n.d.). Retrieved October 14, 2019, from https://geopandas.readthedocs.io/en/latest/gallery/create_geopandas_from_pandas.html.
13. numpy.histogram2d¶. (n.d.). Retrieved October 14, 2019, from https://docs.scipy.org/doc/numpy/reference/generated/numpy.histogram2d.html.
14. VanderPlas, J. (n.d.). Aggregation and Grouping. Retrieved October 14, 2019, from https://jakevdp.github.io/PythonDataScienceHandbook/03.08-aggregation-and-grouping.html.
15. numpy.linspace¶. (n.d.). Retrieved October 14, 2019, from https://docs.scipy.org/doc/numpy/reference/generated/numpy.linspace.html.
16. Data type objects (dtype)¶. (n.d.). Retrieved October 14, 2019, from https://docs.scipy.org/doc/numpy/reference/arrays.dtypes.html.
17. (n.d.). Retrieved October 14, 2019, from http://cs231n.github.io/python-numpy-tutorial/#numpy.
18. Siddiqi, A. (2017, November 23). Data Visualization in Python - Histogram in Matplotlib. Retrieved October 14, 2019, from https://medium.com/python-pandemonium/data-visualization-in-python-histogram-in-matplotlib-dce38f49f89c.