Analyzing Path finding Algorithms

COMP 4202 Project Report

Dawson Theroux (SN: 101106602)

Due: December 6th 2022

Introduction

Algorithms and graph theory are some of the fundamental pillars of Computer Science. Understanding how algorithms work and their running times can be simple at times from a general point of view, but it can sometimes be hard to fully visualize what they are doing on a real dataset. The goal of this project was to implement various path finding algorithms used by route planning apps such as google maps to see analyze their average running times on real world data along with their search areas. In this case, search area is defined as the edges and nodes explored by the path finding algorithm while finding the path in the graph. The algorithms that were implemented and tested where Dijkstra’s Algorithm, A\* search with Euclidian distance heuristic, A\* search with Manhattan distance heuristic, Breadth Frist Search, and Depth First search. These algorithms were implemented in python and analyzed on road network data for downtown Ottawa, downtown Toronto, and New York City. Although the results of this analysis where predictable from the start, it was very interesting to see the search area created by each of the different algorithms, and how long each algorithm took with respect to each other on real world road data.

About the path finding algorithms

Path finding algorithms are exploratory algorithms in which a graph built with edges and nodes is explored until some end goal is achieved. In this case, the end goal is to find a path of nodes and edges in the graph that connects two points. Some path finding algorithms such as Breadth First Search, Dijkstra’s Algorithm, and A\* search can guarantee that they find the shortest path in the graph because they are able to guarantee that the shortest possible path is always the one being explored at a given time. \*\*Explain what a frontier is\*\*

Breadth First Search (BFS) works by starting at the start node and adding all its neighbours to a FILO (First In Last Out) queue. It then chooses the first node in the FILO, sets itself as the parent of the node chosen, and explores the node from the FILO queue by adding all its neighbours to the FILO queue. This is repeated until the end node is achieved and it is then simply a task of reversing through the parent nodes of all the nodes starting from the end node to get the path from start to finish. Since a node at depth 2 is only explored after all nodes at depth 1 are explored, BFS can guarantee that the path found from the start node to the end node traverses the least number of edges, since no other path in the graph with less edges remains unexplored. For this project, this does not necessarily mean that it will find the shortest path since it does not take edge length into account. Instead, it means that it will find the path that traverses the least number of edges.

Dijkstra’s Algorithm works similarly to Breadth First Search, but with one small difference. Instead of using a FILO queue, it instead uses a priority queue to store the frontier to keep it sorted by the path cost to get a given node. This priority queue allows Dijkstra’s algorithm to guarantee that the shortest path between two nodes (based on edge weight) will be found. This is guaranteed in a similar way to BFS in that since no unexplored shorter path exist in the graph, it is impossible that Dijkstra’s finds a path that is not the smallest from the start node to the end node.

The A \* search algorithm differs the most from the other algorithms as it is a heuristic-based path finding algorithm. Heuristics allows \*\* Talk about what heuristics Do\*\* \*\*Talk about consistency in heuristics and the other word I am blanking on\*\* \*\*Talk about the heuristics used for this assignment (Manhattan Distance and Euclidean distance)\*\*

Finally, the last search algorithm explored is Depth First Search. This search algorithm was added to explore the effects of search algorithms that do not find the shortest paths and how that effects the running time of the algorithms. Depth First Search (DFS) works in the opposite way of BFS in that the frontier is implemented using FIFO (First In First Out). This means that DFS will start by exploring the start node and adding all its neighbours to the frontier. It will then explore the first node in the frontier and add all its neighbours to the frontier, and so on. Since DFS uses a FIFO, it ends up exploring to the deepest depths of the graph. This leads to some interesting running times on the different datasets since this algorithm does not necessarily find the shortest path.

Implementation

The implementation of this project was achieved using Python and its many libraries, as well as ArcGIS for dataset acquisition. The project repository can be broken down into three components: The input data, the Path finding algorithms, and the Path Analyzer. All the data is stored in shapefiles in the “OttawaData”, “NYCData” and “TorontoData” files. These files are read by the PathAnalyzer.py program and are used to generate the output data, such as algorithm execution time, and explored area images. The path finding algorithms are implemented though four python files *BreadthFirstSearch.py, DepthFirstSearch.py, DijkstrasWithExplored.py,* and *AStarWithExplored.py*. These files all contain the respective path finding algorithm with an extra return value which represents its search area. Finally, the *PathAnalyzer.py* is a script which reads in all the Shapefile data, runs the path finding algorithms, and produces the output data and images.

*PathAnalyzer.py* is the main Python file which manages, reading in all the Shapefiles and generating their graphs, running the various path finding algorithm on the generated graphs, and displaying the results. Reading the Shapefiles and creating their graphs is done using a package called NetworkX. This package, along with some data augementation, can be used when running the path finding algorithms and can even help when displaying the paths generated. Displaying the search area created by the various search algorithms was done using the matplotlib library.

Since the implementation was meant to use real world road data, my first task was to generate road and street data. To save computing time, I used ArcGIS to get road data and trim it down to a specific area of interest. This helped me limit the number of nodes and edges in the graph to save computer resources. Converting the shapefile data to python graph data was done using the NetworkX package. This package can take in shapefiles and with some modification can create a Graph object which can be used to run the path finding algorithms.

Although there exist many python packages that can run the various path finding algorithms talked about in this project, I was forced to implement them by hand to get the necessary data. The standard Dijkstra’s, A\*, and BFS algorithms are not able to return the search area as part of their default implementation. To mitigate this, I Implemented the algorithms myself and added an extra NetworkX Graph object which adds all explored edges and nodes. This graph can then be displayed along side the shortest path to show the search area of all the algorithms.

Displaying the output was done using the Matplotlib library for python. This library is a standard python library used in many projects to plot all kinds of graphs. Firstly, all the graph data is projected onto the plane. Then the explored area is drawn on top of that in a different colour to show the search area of the algorithm. Finally, the shortest path is displayed to show the shortest path found by the algorithm. (ADD THE COLOURS THAT THINGS ARE BEING DISPLAYED AS.)

Algorithms Analyzed

DFS:

BFS:

A\* (Euclidan and Manhattan Distance)

Dijkstra’s

Findings

With the analyzer script implemented and the test data generated, we are able to run the various path finding algorithms on real work data to see their running time and search areas.

Ottawa:

Toronto A Star:



Chart

Description automatically generated

Manhattan:

Insights

Talk about best case big O and worstcase bigO

Talk about non-

**Bibliography**

Python

NetworkX

Where I got the algorithms from.