For homework 3, I choose my small graph to be a representation of San Francisco’s municipal transportation system (Muni for short). Muni was chosen because it is a complex system of buses, trains and trolleys that is connected enough to service San Francisco seemingly with ease. It runs in some capacity at all hours of the day and makes for an optimal use case for the effectiveness of different algorithms. Edges were selected based on routes that were operational as of March 15th, 2024. While much more complex than was originally required for the small graph, my recent trip to GDC made the project personally important to me. The data required extensive cleaning and preparation to ensure the graph as depicted would be fully connected, including adding additional incoming edges to every node as well as ensuring each node had at least three outgoing edges. The added edges were chosen based on proximity to attempt to simulate the idea that, if necessary, one could walk a short distance from one stop to another to change bus routes. Originally, I had planned for a heuristic where edges would be penalized and incur a higher cost if they required changing routes. However, the difficulty factor for tracing back what a previous edges route was proved to be beyond the scope of the project. It is something that I am deeply interested in investigating further.

The large graph is created by adapting a script made by Jayden Sansom in Python to generate a specified number of nodes and edges. The script was edited to generate the nodes into a specified geographic range, the number of edges and how they connect is also randomly generated and sampled. The graph is composed of 500,000 nodes, with efficient data structures this graph is able to be run on even the worst algorithms in under 15 seconds. My worst performing algorithm by far temporally was attempting to run Dijkstra’s algorithm without early exit. In fairness there is really no reason to continue to run the algorithm after the goal node is found and is the cheapest node on the list of nodes to consider. This forces the algorithm to check the entire list every single run and is entirely unnecessary in my opinion for a static environment. However, I can see some kind of environment where perhaps stepping on a tile causes an event to trigger and perhaps somehow the algorithm is set such that it looks for those events.

Completing even the worst algorithms in under 15 seconds feels very successful as at one-point, alternative graph representations were experimented with, one such representation being a vector-based adjacency list. It was difficult to work with and while it ran well enough for the Muni graph, completing around 10 trials each with six individual runs in them in 3 minutes it completely fell apart for the large graph. It caused frequent seg faults and even when it completed took over ten minutes for a single run. Subsequently the algorithms were rewritten to accept an unordered adjacency map instead and found much greater success with this data structure. Even the large graph can be processed in a hundred trials in around 10 to 15 minutes and the small graph of 50 nodes can complete 100 trials with ease often in under 3 minutes. The program does still seg fault occasionally especially with the large graph. Currently with the data collected, the large graph runs a single algorithm in a max of 13 seconds as recorded. I have seen higher however.

For the large graph all of the Astar algorithms with any and all modifications run for an average of less then 1.5 seconds, with a max run time in 100 trials of 5.5 seconds. All algorithms that do not use early exit will by necessity open every single node and therefore have a fill of the full node list. On average all of the Astar variants open around half the Nodelist (250,000). Their fringe is on average around 180,000 or about a third of the Nodelist. On a particularly difficult run however Astar will still end up opening almost the entire list. Interestingly even on the poorest of runs only 2/3 of the fringe will be left in fringe.

I tested three different heuristics, one being classic A\* Euclidean distance between from the start to the current node added to the Euclidean distance between the current node and the goal node. This is geospatial coordinate Euclidian distance that is adjusted to account for the idea that the earth is not a flat coordinate system. The second heuristic I tested was to use a naive Manhattan distance instead of the Euclidean (after converting the latitude and longitude to radians and then multiplying by the size of the earths radius to attempt to bring it to the same weightiness as the geospatial distance). This heuristic is always inadmissible especially as we consider that our graph is not based on any kind of tile system. It will always overestimate sometimes significantly so. The last heuristic I attempted to use was the number of neighbors (multiplied by 5). Ideally, I would use the number of neighbors negatively with a minimum weight being calculated as the floor of the edge weight between two nodes. This idea was based upon the original graph being from the San Francisco transportation system. I wanted to incentivize the algorithm to choose to travel to perhaps less than ideal nodes with the concept of them possibly being hub nodes that may have connections which allow faster and more direct transportation to the goal node. However, for analysis I also tested making the number of neighbors heuristic positive to guarantee myself an inadmissible heuristic as applying a negative modifier to an admissible heuristic would never end in an inadmissible heuristic. This neighbor-based heuristic is most likely to become inadmissible when there are many neighbors, especially in the very large graph some of the nodes have a fairly large number of neighbors.

As the chart below shows the AStar early on average found a path with fewer nodes but with around a 700 KM difference in distance. Notably, however, the time to find that path, even on this very large graph, was under 1.5. Again, this is my very large graph of 500, 000 Nodes and around 3,000,000 edges. Comparatively Dijkstra was averaging around 3.2 seconds. This is about 2.5 times slower than AStar. Adding in the inadmissible heuristics did not make them better at all in any meaningful way. I feel that due to the geospatial distance being such a good admissible heuristic it was unlikely that I would have found a spatially based heuristic that would have performed comparatively well. Notably the Neighbor heuristic on this graph made little difference. This is likely due to the overall random nature of the graph. I was unable to make the Manhattan distance work in a way that would feel meaningfully comparative when compared to the Euclidean distance and as such will refrain from doing so. They appear to be in different units, making comparison difficult.

When analyzing the graphs against the Muni graph we see a surprising change in algorithm heuristic efficiency. Here the addition of the inadmissible heuristics improves efficiency slightly. Adding the neighbor count to the normal Geospatial distance allows the Astar algorithm to on average perform about as well as an early exit Dijkstra. This is unexpected as I believed when I designed the heuristic that making the neighbors have a negative influence would improve their efficiency. Additionally switching to a Manhattan distance in the Muni graph also provides a reasonable amount of improvement. My reasoning for this would be that the Muni graph is not randomly connected. Edges were chosen based on either pre-existing routes, or the nearest stop to the host stop for adding additional graph edges. In doing so we’ve naturally promoted a structure which makes the neighbor heuristic more favorable, perhaps it is because the busses tend to travel in roughly one direction. If we are already going in the correct direction for our desired goal node, heading to the hub may not make sense as opposed to attempting to stay on the line until we reach our stop, even if we’ve taken a round about way home.

With regards to the implementation for the level in part 4, I found that because I was using a tile-based system, I was once again forced to modify my algorithms. I introduced new classes to accommodate our world walls and floors. Additionally, I found I had to slow down my max velocity by a good amount to avoid overshooting my incremental goals. This slow down revealed major flaws in me align system which required a major overhaul to fix. Thankfully, however I was able to seek help to clean up that algorithm quickly. I also realized as I built the room map, why level editors are so heavily valued. Many times, I wanted to quickly adjust something but because of my build doing so would prove tricky. In the end my graph for my level is made by simply leaving out any edges which have walls, and by implementing diagonals on the condition that the tiles which have a wall in one or more directions may not have a diagonal path in that direction.

A screenshot of a graph

Description automatically generated

Large Graph Avgs

A screenshot of a number

Description automatically generated

Large Graph Maximums

A screen shot of numbers

Description automatically generated

San Fran Graph Avgs

A map of a city

Description automatically generated

Graph of the san Francisco Municipal transportation system with added edges, rendered in python.

