Piercson Thomas
CMPT 307
Nov.5th

The milestones for the project are

1. How to organize the data
2. Programming Dijkstra's algorithm since it is a template for the 2 other algorithms

Milestone 1. How to organize the data

The way I organized the data was to make a class for all 1000 nodes and store them in an array. Node 1 was to be in array[0] and Node 1000 be in array[999], each class has coordinates and 'edges to nodes' for every Node. This array is passed to all search algorithms as a reference for the information about the nodes.

> Array[i] where 0 <= i <1000, is a node that has coordinates [latitude,longitude] and a list of nodes it is connected to (Eg. [145,678,...] and note that all the nodes it is connected to are scaled down by 1 since the array starts at 0).

Milestone 2. Programming general search (Dijkstra)

To write the code for all of the search algorithms, an important feature is the priority queue. The priority queue I have used was from an assignment I did this semester for CMPT310 on getting Pac man to his goal using BFS, DFS, and A* (the creators are credited in the source code shortest_path.py). The priority queue will have the indexes of the nodes to be visited and the priority is the distance to each node from the start node.

Another key element is an array of size 1000 that contains the distances of the nodes from the start node that the algorithm has visited, that will be returned when the algorithm finishes.

Dijkstra's algorithm takes a start node index, goal node index and the array of information about the nodes. It starts by initializing both the priority queue and the list of distances (where all the distances are set to infinity). We add the start node to the priority queue with a priority of 0 and set the distance of the start node to 0. It will continue looping until the priority queue is empty or we have reached the goal node. Within the loop, it first removes the node with the lowest priority and checks if it's the goal node or else it will begin looping through the nodes that it can travel to. The distance is calculated by adding the removed node's distance plus the distance between the removed node and the node it is connected to. It compares the calculated distance with the distance of the node in the distance array, if the calculated distance is less than the distance of the node in the distance array, we update it with the calculated distance. It then adds the 'node we can travel to' to the priority queue with the

priority of the calculated distance and then continues until there are no more nodes to travel to and then continues the original loop.

A* is very similar to Dijkstra's except when adding node indexes to the priority queue. When we add the node indexes to the queue it has the priority equal to the distance of the node plus the straight line distance from the node to the goal node. This small difference greatly decreases the nodes visited compared to Dijkstra's (in one case, Dijksta visits 523 Nodes and A* visits 188 Nodes).

Landmarks search is also very similar to Dijkstra's but with a little more work. We create landmarks (which is a list of shortest distances from a start node) by using Dijksta's algorithm. However we set the goal node to a value that would never exist in the list (like 1234) so it finds the shortest distance to all the nodes from a start node. In this implementation we choose 4 landmarks, which are the largest latitude, smallest latitude, largest longitude, smallest longitude. I produce the landmarks just once. Each node also has a different priority when added to the priority queue, which is equal to the maximum of the distances between the current node and the goal node from all the landmarks plus the node's distance.

Here is the output of the program:

**dijkstra Avg nodes Visited: 490.45**
**A* Avg nodes Visited: 224.0**
**Landmark search Avg nodes Visited: 144.65**
**A* visits 45.67234172698542 Percent less Nodes visited than Dijkstras**
**Landmark visits 29.493322458966258 Percent less Nodes visited than Dijkstras**