Carl Klier
CSC584 Game AI
3/31/2021

Homework 2: Pathfinding and Path Following

This homework assignment asked us to implement search algorithms on graphs and to then use those search algorithms to create a shortest path that a character could follow as part of a path following implementation. There was a lot of freedom given in this homework assignment. We were able to create our own graphs and to implement our own heuristics. Throughout the process of coding the algorithms and trying out different parameters and heuristics, I really gained a better understand of the difficulties and tradeoffs made when trying to optimize a shortest path algorithm.

The first task of this assignment was to create two different graphs. The first graph was to be relatively small so that we could verify that our search algorithms were working correctly, and the second graph was to be very big so that we could test the limits of the search algorithms. I decided to create, for my first small graph, a graph of the cities of North Carolina from the census of 1860. I created the nodes of my graph using the cities shown in the image in Figure 1 of the Appendix. I made the assumption that the largest towns of 2000-5000+ people were all connected to each other and the towns of <2000 people were connected to the closest large town. In order to make this graph a digraph, an edge connected to a city represents both an incoming and outgoing edge of that city. In order to make this graph weighted, an edge between two cities has a weight equal to the Euclidean distance between the two points of the cities relative to the size of the image. The edges that were created based on this assumption are seen in the graph in Figure 2. This graph has 25 nodes and 68 edges. I chose this graph because distance between two places is an intuitive way to assign a weight to an edge. Also, with the way the large towns are connected, there are multiple different paths to reach certain nodes, some longer than others. Because of this, I will get to see if my implementations of Dijkstra's and Astar search are able to find the actual shortest path. The specific location of the cities also give some interesting results when looking at the difference between heuristics for the Astar algorithm.

My second graph which is very large consisting of 23,113 nodes and 93,439 edges was created from a dataset I was able to find online. This graph, according to the networksciencebook.com where I go the dataset from represents a "Scientific collaboration network based on the arXiv preprint archive's Condense Matter Physics category covering the period from January 1993 to April 2003. Each node represents an author, and two nodes are connected if they co-authored at least one paper in the dataset. "

The second task of this homework was to implement both Dijkstra's and Astar algorithms. For the small graph when running Astar, I used the Euclidean distance heuristic. For the large graph, since the nodes don't have a meaningful physical distance apart, the Euclidean distance heuristic isn't usable. So, for the large graph, I had to invent my own heuristic. One heuristic that came to mind was the difference between the average shortest path length between any two nodes and the length of the shortest path from our start node to the current now we were examining. This would, in some way, give us an estimate of how much further was needed, on average, to get to the goal node. However, this would require running an algorithm like Dijkstra's on every pair of nodes which would take way to long. So, using my understanding of network science, realized that the collaboration graph I was using was an example of a co-occurrence network which is itself an example of a small world network. This means my large graph is likely to have certain properties that small world network graphs have. The property that I used is the property that the distance between any two randomly chosen nodes is proportional to the logarithm of the number of nodes in the graph. So, for my heuristic, to estimate the average shortest path length between any two nodes, I simply made the assumption that the average distance is log base 2 of the total number of nodes. I ran into a lot of difficulty actually combining this heuristic with my existing Astar algorithm, so I created a new Astar algorithm called AstarLarge for this heuristic that is in the AstarLarge.pde file.

We expected Dijkstra's algorithm to give the optimal shortest path and we can see that it does. Also, since our heuristic, the Euclidean distance, is consistent and admissible, we know that Astar will get the optimal path, just as with Dijkstra's. This can be seen in Figure 4 and Figure 6. Since the Astar algorithm has more information available to it though its heuristic function, we expect that Astar will need to explore less nodes and thus will find a shortest path quicker. And indeed that is the case as seen in Figure 3 and Figure 5 where Astar visits less nodes and finds the optimal shortest path in less time. Also of note is that, as seen in Figure 3 and Figure 5, Astar actually used less memory than Dijkstra. This can be seen when comparing the maximum number of objects stored in the three main data structures: the open set, the closed set, and the hashmap of tags.

When looking at the large graph, we again see in Figure 7 and Figure 9 that Astar expands fewer nodes when compared to Dijkstra but doesn't always find a path in less time. This could be due to the max size of the open set being much larger for Astar looking for a path from "0" to "64" (seen in Figure 7) than compared to the max size of the open set with Dijkstra for those same start and goal nodes. Also of note is that since my custom heuristic is not admissible, Astar doesn't return the optimal shortest path as seen in Figure 8 and Figure 10. One of the benefits of Astar is that the memory usages in terms of the size of the closed set is much smaller when compared to Dijkstra's and this can be seen in Figures 7 and 9.

When nodes have a high degree, many incoming and outgoing edges, Astar will appear to approach near the goal much quicker when using a heuristic like the Euclidean distance. Dijkstra's, on the other hand, will look at the nodes the shortest distance away from the start node, thus appearing to stay closer to the start node at first. We see this happening on our graph of the towns in NC when we look at a start node on the far left (Hendersonville) and a goal node on the far right (Elizabeth City). Compare the closed set represented by the white dots in Figure 11 which ran Dijkstra to the closed set in Figure 12 which ran Astar and see that the closed set in Figure 12 is much closer to the goal node after the same number of nodes expanded. Thus, the average degree of nodes plays a big role in determining performance.

The third question asked us to implement two heuristics for our smaller first graph. I implemented the Euclidean distance as my admissible heuristic and the Manhattan distance as my inadmissible heuristic. With an inadmissible heuristic, Astar is not guaranteed to provide the optimal shortest path. If we pick two nodes that are not leaf nodes in the graph, then Manhattan will always overestimate the distance to the goal. This is because the non-leaf nodes are connected by a straight line and the shortest distance between two points is always a straight line. So in this case the Manhattan distance overestimates by a factor of 2/sqrt(2).

One interesting output I got was that the Manhattan heuristic actually found the optimal path quicker, and while using less memory, than the Euclidean heuristic when calculating the path from Charlotte to Oxford. This can be seen in Figure 13. Look at Figure 14's closed set (which used Euclidean) compared to Figure 15's closed set (which used Manhattan). Because Raleigh is almost directly below Oxford, the Manhattan distance is actually less then having to go up to Salisbury and then diagonal to Oxford.

I also explored using a random value heuristic on the large graph and compared this to running Dijkstra's. I was able to see that the random value heuristic, which is an inadmissible heuristic, gave a suboptimal path when trying to find the shortest path from "0" to "604". See Figure 17 for the two paths and notice their different lengths. The data table in Figure 16 shows the difference in the time required to find a path between the two nodes. Dijkstra was able to find the optimal path in a way shorter amount of time than Astar was able to find a suboptimal path. Also, notice in Figure 16 that the memory usage required was much higher for Astar running the random value heuristic. This confirms what we learned in class that Astar is only better when you have an admissible heuristic. In order to increase performance, you need to spend time analyzing your data to come up with the best heuristic to use for your data in order to take advantage of the benefits of the Astar algorithm. Another point worth noting is that when we set the heuristic value function to return 0, we actually get the same result as just running Dijkstra's algorithm.

The last problem of the homework was to combine our Astar pathfinding algorithm with our movement algorithms that we learned about in Homework 1. I created a simple room layout with a couple of obstacles implemented with a 2D graph and grid structure like the example code provided. I decided to use the dynamic seek and dynamic look algorithms combined with Astar to have the boid object follow the shortest path to some point selected by the user by mouse click. It took a while to play around with the dynamic movement parameters until I got a movement that was smooth and didn't oscillate too much. One of the hardest challenges was quantization when I had to map the location of a mouse click to a node in the graph. However, I was finally able to get it working by dividing the location of the mouse click by some scalar and taking the floor of that value. Please see the attached video in the file folder called PathFollowing.webm to see path following in action. This file can be played by dragging and dropping into a Chrome tab.

In conclusion, by the end of this assignment, I felt like I had gained considerable knowledge and experience with Dijkstra's and the Astar algorithms. I enjoyed looking at the various possible heuristics and seeing if they could find the optimal path just as Dijkstra's could. In addition, the "Putting it All Together" section of the assignment was a great way to build something bigger than just an algorithm. I now have a better understanding of how pathfinding and path following are implemented and am much more aware of the effort required to implement these algorithms when I see them at work in video games.

Appendix

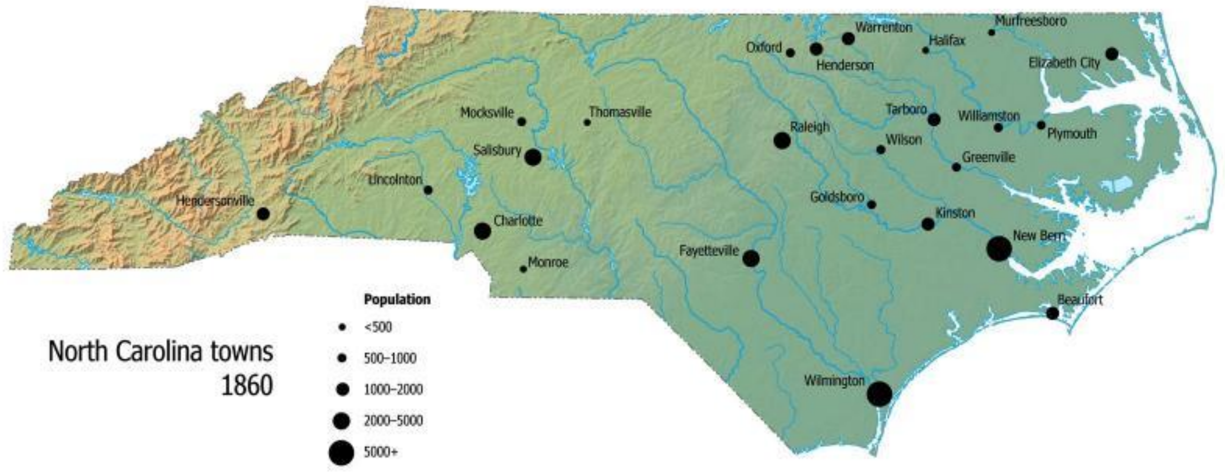


Figure 1 – The base image used as inspiration for my small graph

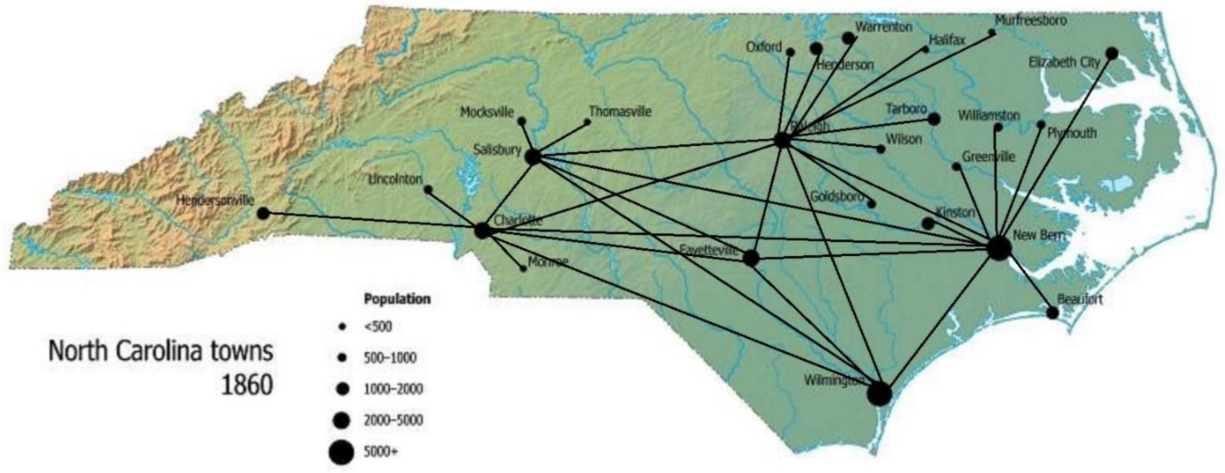


Figure 2 – The small graph

	Dijkstra	Astar (Euclidean)
Nodes in shortest path	4	4
Run time (millis)	1211	867
Num nodes expanded	25	18
Max num in Open Set	10	14
Max num in Closed Set	25	18
Max num in tags HashMap	25	25

Figure 3 – Comparing Dijkstra and Astar on small graph looking at shortest path between Hendersonville and ElizabethCity

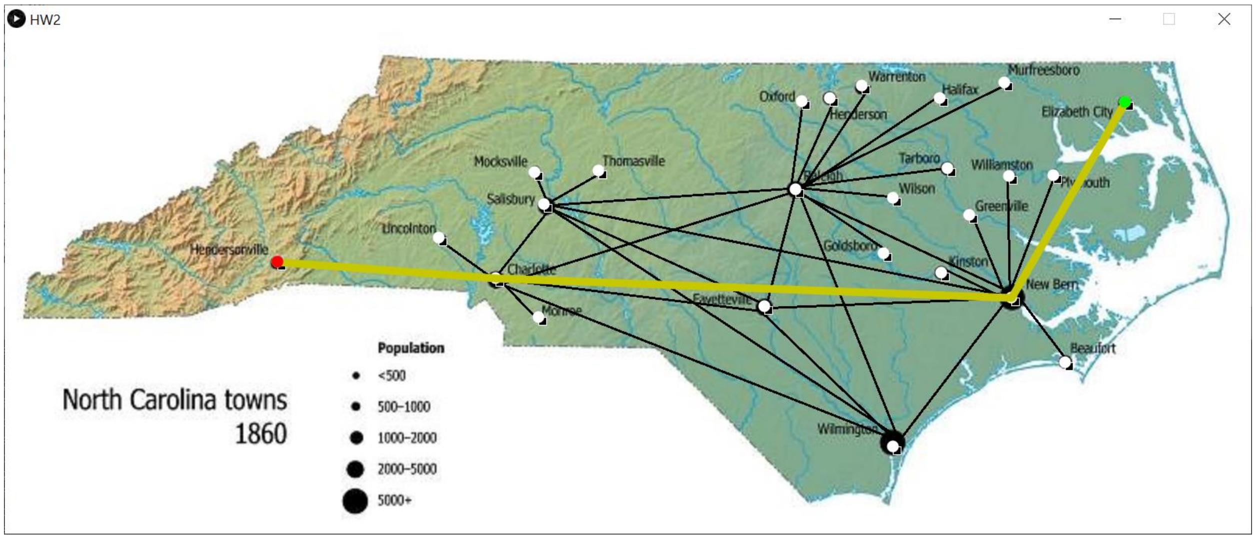


Figure 4 - Shortest path from Hendersonville to ElizabethCity given by both Dijkstra and Astar

	Dijkstra	Astar (Euclidean)
Nodes in shortest path	4	4
Run time (millis)	768	366
Num nodes expanded	16	8
Max num in Open Set	11	13
Max num in Closed Set	16	8
Max num in tags HashMap	25	25

Figure 5 - Comparing Dijkstra and Astar on small graph looking at shortest path between Williamston and Tarboro

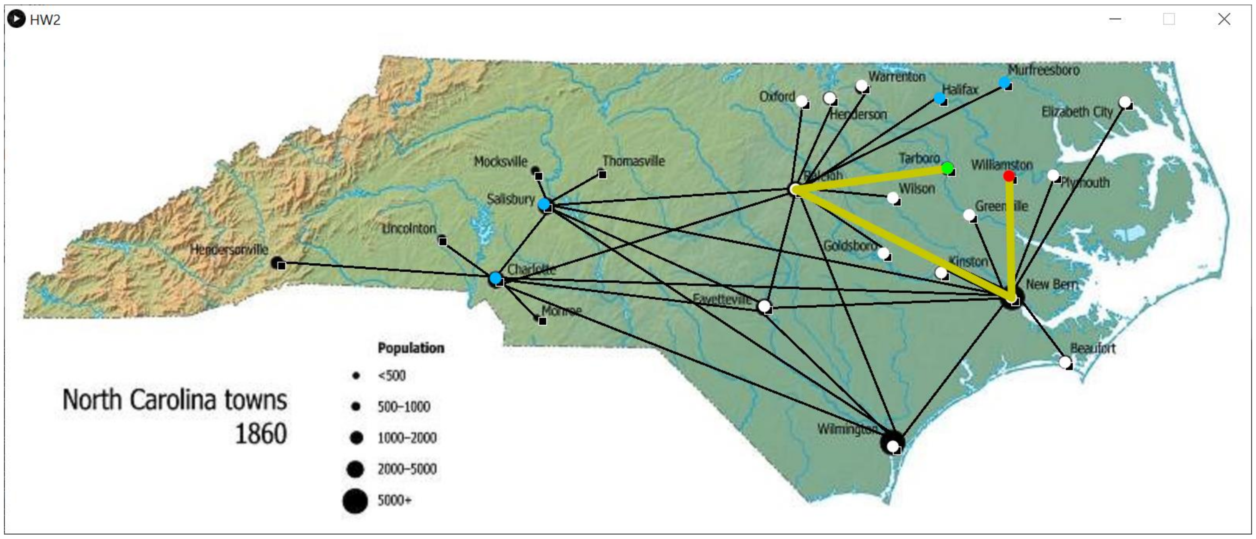
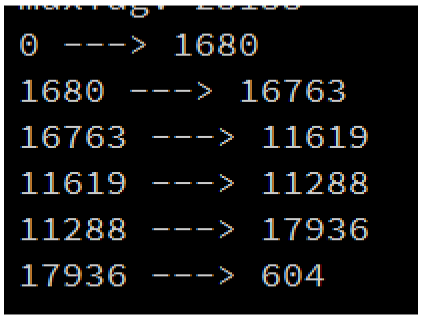


Figure 6 - Shortest Path for Williamston to Tarboro given by both Dijkstra and Astar

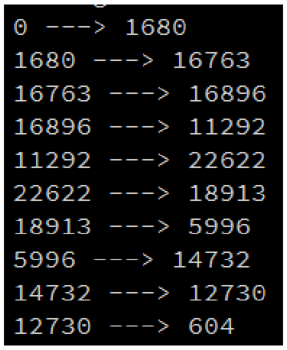
	Dijkstra	Astar (custom heuristic)
Nodes in shortest path	7	11
Run time (millis)	24470	38855
Num nodes expanded	8020	6781
Max num in Open Set	9024	17591
Max num in Closed Set	8020	5350
Max num in tags HashMap	23133	23133

Figure 7 - Large graph looking at shortest path between nodes “0” and “64”



```
0 ---> 1680
1680 ---> 16763
16763 ---> 11619
11619 ---> 11288
11288 ---> 17936
17936 ---> 604
```

Shortest path using Dijkstra



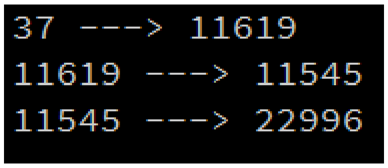
```
0 ---> 1680
1680 ---> 16763
16763 ---> 16896
16896 ---> 11292
11292 ---> 22622
22622 ---> 18913
18913 ---> 5996
5996 ---> 14732
14732 ---> 12730
12730 ---> 604
```

Shortest path using Astar

Figure 8 – Astar returns a suboptimal path between nodes “0” and “64”

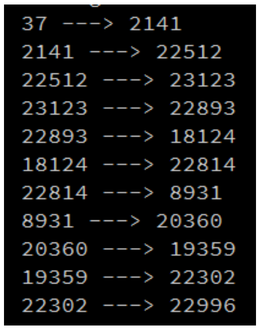
	Dijkstra	Astar (custom heuristic)
Nodes in shortest path	4	12
Run time (millis)	7349	2527
Num nodes expanded	2382	636
Max num in Open Set	7104	4669
Max num in Closed Set	2382	600
Max num in tags HashMap	23133	23133

Figure 9 - Large graph looking at shortest path between nodes “37” and “22996”



```
37 ---> 11619
11619 ---> 11545
11545 ---> 22996
```

shortest path using Dijkstra



```
37 ---> 2141
2141 ---> 22512
22512 ---> 23123
23123 ---> 22893
22893 ---> 18124
18124 ---> 22814
22814 ---> 8931
8931 ---> 20360
20360 ---> 19359
19359 ---> 22302
22302 ---> 22996
```

Shortest path using Astar

Figure 10 - Astar returns a suboptimal path between nodes “37” and “22996”

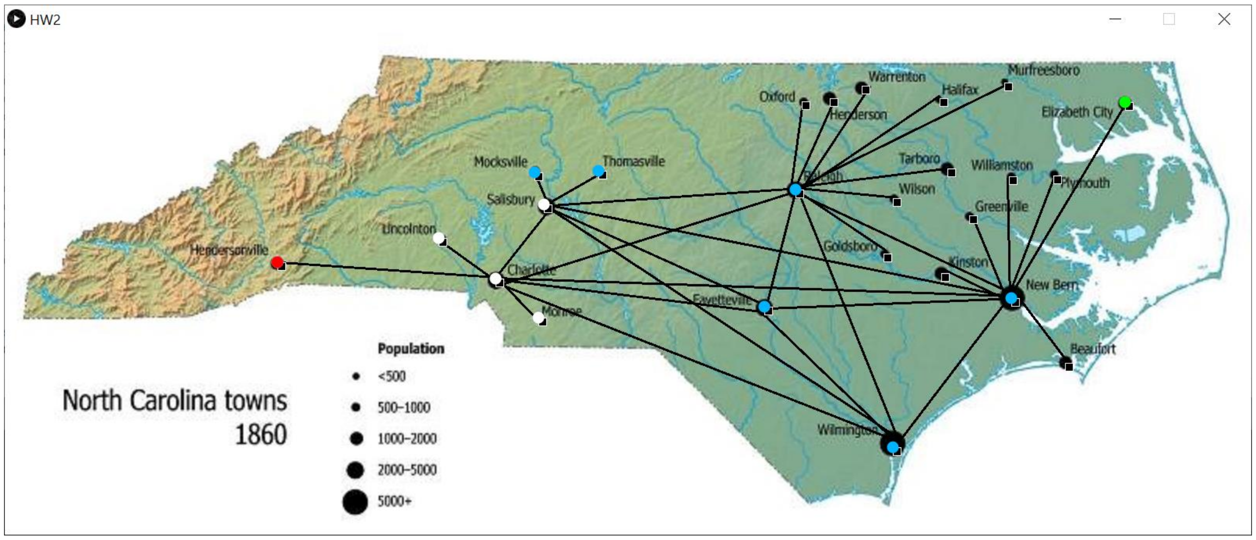


Figure 11 - Dijkstra exploring nodes closer to the start node

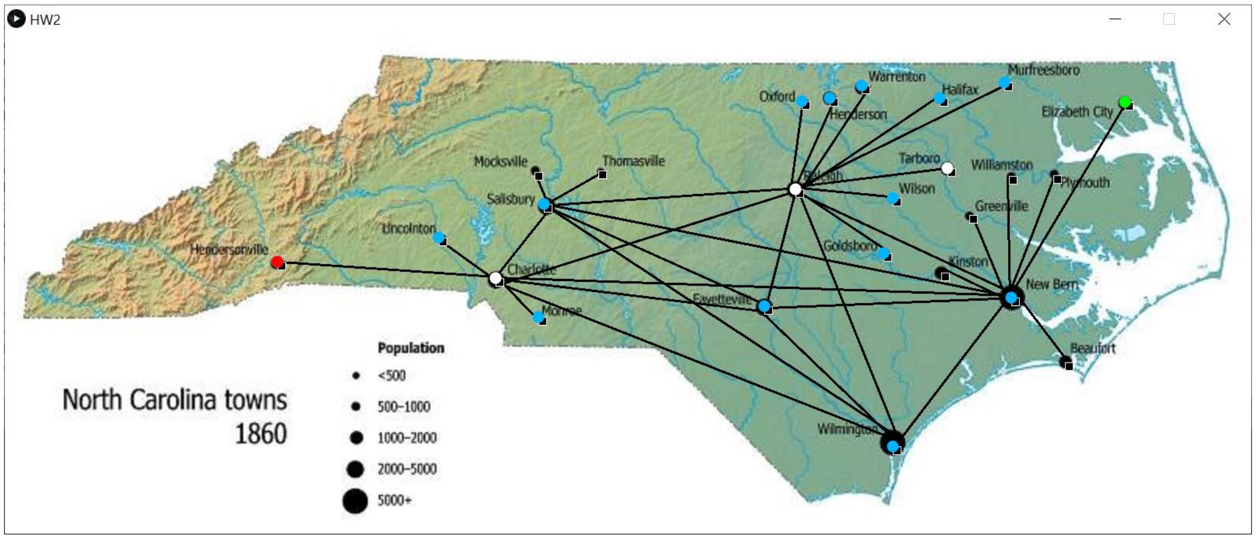


Figure 12 - Astar jumping very quickly to nodes closer to the goal node

	Astar (Euclidean)	Astar (Manhattan)
Nodes in shortest path	2	2
Run time (millis)	312	110
Num nodes expanded	7	3
Max num in Open Set	13	15
Max num in Closed Set	7	2
Max num in tags HashMap	25	25

Figure 13 – Comparing Euclidean heuristic vs Manhattan heuristic on the path from Charlotte to Oxford

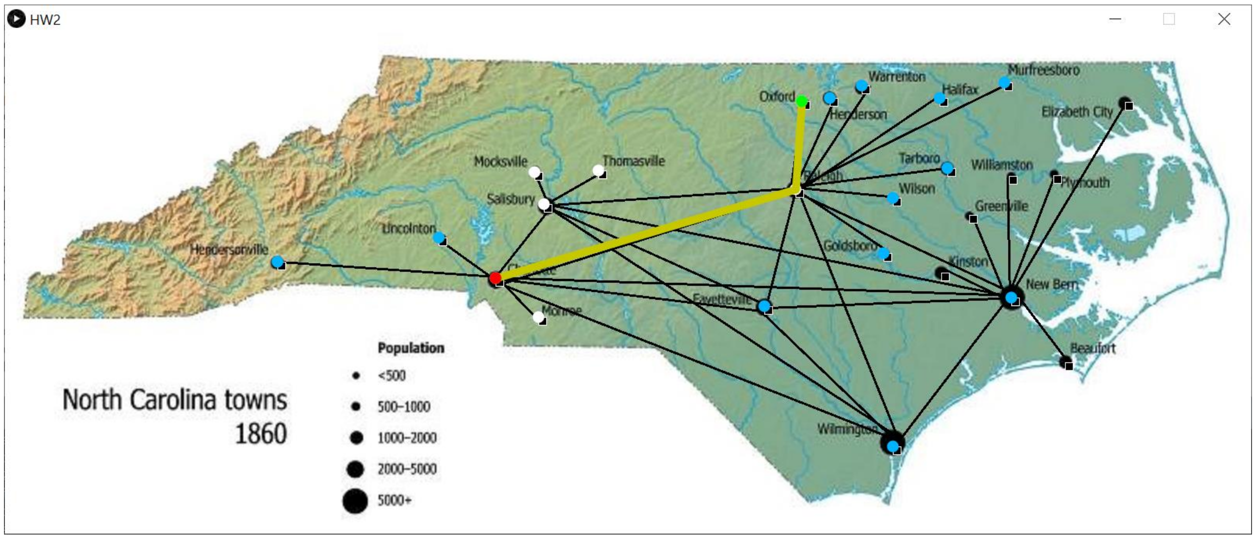


Figure 14 - Astar using Euclidean heuristic – notice size of closed set compared to Figure 14 which used Manhattan distance

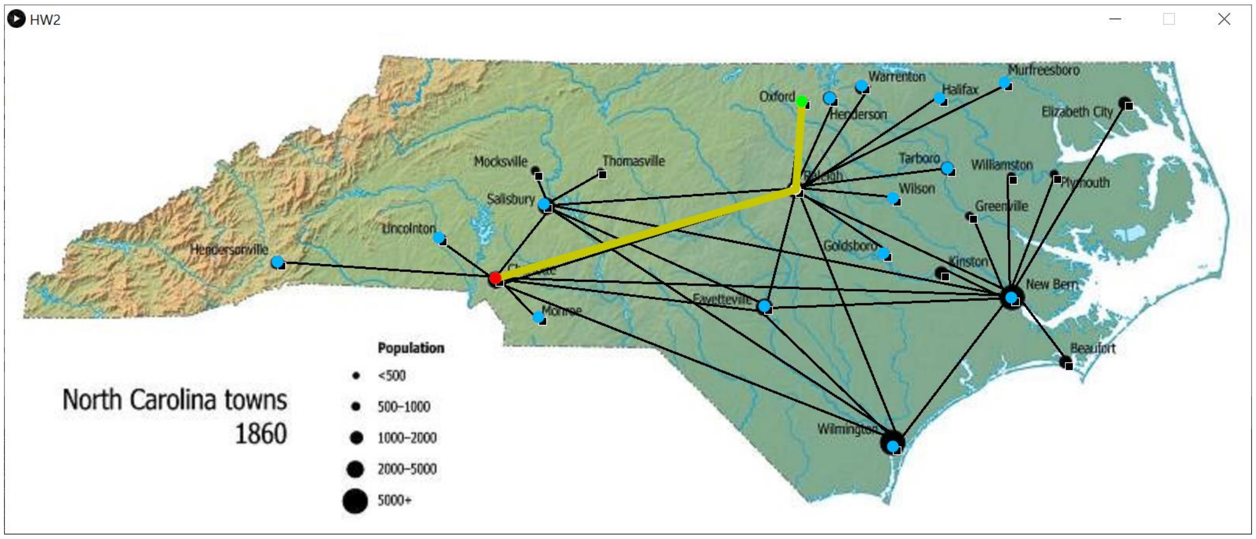
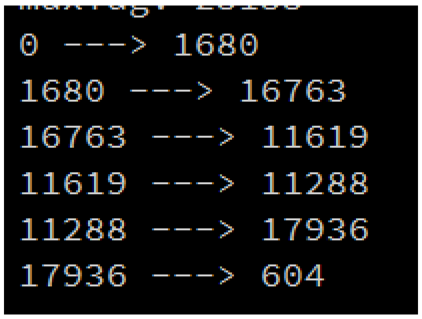


Figure 15 - Astar using Manhattan distance – notice size of closed set is smaller than that in Figure 14

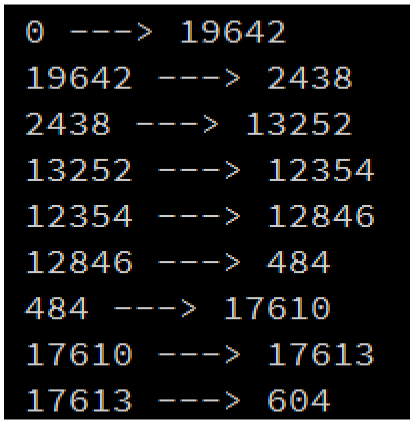
	Dijkstra	Astar (random value heuristic)
Nodes in shortest path	7	10
Run time (millis)	24470	63049
Num nodes expanded	8020	9967
Max num in Open Set	9024	27188
Max num in Closed Set	8020	7192
Max num in tags HashMap	23133	23133

Figure 16 – Comparing Dijkstra and Astar using a random value heuristic to find the shortest path between “0” and “604”



```
0 ---> 1680
1680 ---> 16763
16763 ---> 11619
11619 ---> 11288
11288 ---> 17936
17936 ---> 604
```

Shortest path using Dijkstra



```
0 ---> 19642
19642 ---> 2438
2438 ---> 13252
13252 ---> 12354
12354 ---> 12846
12846 ---> 484
484 ---> 17610
17610 ---> 17613
17613 ---> 604
```

Shortest path using Astar

Figure 17 – The shortest paths output from Dijkstra and Astar using random value heuristic for the path between “0” and “604”

References

My small graph was based off of a map of the cities of North Carolina towns in 1860 found at https://www.ncpedia.org/media/map/north-carolina-towns-1860. I also used that image as the background in my processing file HW2.pde.

The dataset for my large graph was found at http://networksciencebook.com/translations/en/resources/data.html and is the file collaboration.edglist found in the downloadable zip file at that website. The provided reference is: Ref: Leskovec, J., Kleinberg, J., & Faloutsos, C. (2007). Graph evolution: Densification and shrinking diameters. ACM Transactions on Knowledge Discovery from Data (TKDD), 1(1), 2.

https://en.wikipedia.org/wiki/Small-world_network and https://en.wikipedia.org/wiki/Co-occurrence_network were used to determine how to create a custom heuristic based off of the expected shortest path distance between any two nodes for my large graph.

Parts of the code for this project were given to us by Dr. Loftin and some parts of the code were based off lecture materials and PowerPoints provided.