Clayton Baker, Caitlin Hall, Nathaniel Moore, Dylan Womble
COP6416: Advanced Algorithms
Term Project Report
April 16th, 2022

Shortests Path and Minimum Spanning Trees

I. Introduction

The first objective of this project was to implement Dijkstra's algorithm with the A* heuristic with two separate data structures: a linked list priority queue and a min-heap. The A* heuristic algorithm will search for the shortest path between a start and end node, much like google maps or other navigation applications that determine the fastest path from your starting point to your final destination. The main difference of A* compared to Dijkstra's is that it takes into consideration the location and surrounding nodes of the final destination.

In addition to the A* algorithm objective, this project also met the goal of successfully implementing Prim's Algorithm in order to find the minimum spanning tree. This algorithm as well was implemented using the min-heap data structure and a linked list priority queue. This algorithm is a greedy algorithm that finds the minimum spanning tree.

In order to meet these two objectives, the team utilized python programming language to code the algorithms and then display the algorithms functionality through the combination of a graphical user interface and a graph. The user will be able to calculate both the A* algorithm with a choice of start and end nodes. The user will also be able to calculate Prim's algorithm with a choice of starting node. Finally, the user can also incrementally add edges of their liking to the graph with any weight.

II. Data Structures and Complexity

2.1 Data Structures Overview and Implementation

There are many data structures that can be utilized in programming languages to successfully write an algorithm, however choice of data structure can impact the time performance of the program. For the scope of this project, the team implemented both a linked-list and min-heap data structure into the two different algorithms to determine the effect each had on the performance of the overall runtime.

A linked-list is a data structure where the elements are stored with the addition of pointers that point to the next node or element in the list. In a linked-list data structure, there needs to be a node object that contains some field for the data value as well as a field for the reference to the next node in the list. In this data structure, it is only possible to navigate forward in the list by calling .next() or returning the front node of the list. For this project, the team implemented a custom linked-list class that has the ability to push the node into the list, pop a node off of the list, peek at the list which will return the front node, and traverse the list which will print out the entire data structure.

A min-heap is a data structure where the element at the root is the smallest of all the proceeding, or child, elements. This must be true recursively for all nodes in the min-heap as each child node becomes the parent of other children nodes. In order to implement a min-heap data structure into this project, the team wrote a custom class with various functions including simple returns for various locations in the heap like parent, right child, left child, and also a return to determine if that node is a leaf node. A leaf node being a node where no children exist, it is essentially the very bottom of that branch. This min-heap class also includes the ability to swap nodes if need be to put them into the correct sequence based on the value of those nodes. It is important to note that we must be passing around an object as we have to have an identifier field as well as a value field to know exactly what element is going where for our purposes of integrating this to our algorithms later on. Much like linked-list, min-heap also has functions to push a node onto the heap and pop a node off of the heap and these are intentionally named the

same thing so that they can be interchangeable in our overall algorithms when the user chooses which data structure to implement. The most important function here is the heapify function which will put the nodes into the correct order. This function works by saying if the node is not a leaf node and if the value of the node is greater than the left or right child node then it can pass into lower detail and see which decides which child to swap with. The function will choose either the right or left child to swap with and then recursively call the heapify function. Finally, there are functions in this class to define if the structure is empty and a function that will min-heapify the whole thing by recursively calling the heapify function on each node.

2.2 Linked List Complexity

Average-Case

Access	Search	Insertion	Deletion
BigΘ(n)	BigΘ(n)	BigΘ(1)	BigΘ(1)

Worst-Case

Access	Search	Insertion	Deletion
BigO(n)	BigO(n)	BigO(1)	BigO(1)

*where n is the number of elements in the data structure

A linked-list data structure is known to have a time complexity of O(n). O(n) time complexity means that the structure takes linear time. Moreover, this also means the run time will increase linearly at most with the size of the nodes. The two data structures will be compared below, however we can see this linear time occur in Figure 8. The main advantage of a linked list is upon insertion or deletion of an element at the current position since it can be accomplished in constant time. This is opposed to a normal array, it would need to be done in linear time because the elements have to be shifted. The space complexity of a linked-list is O(1) or constant because there is no need for more space outside the fixed number of nodes or elements that are present.

2.3 Min-Heap Complexity

Average Case

Access	Search	Insertion	Deletion
BigΘ(log(n))	BigΘ(log(n))	BigΘ(log(n))	BigΘ(log(n))

Worst-Case

Access	Search	Insertion	Deletion
BigO(log(n))	BigO(log(n))	BigO(log(n))	BigO(log(n))

*where n is the number of elements in the data structure

A min-heap data structure is known to have a time complexity of O(log(n)). O(log(n)) time complexity means that the structure takes logarithmic time. Moreover, this also means the run time will increase slowly as the number of nodes increases. The main advantage of a min-heap is that adding more elements to the binary heap after making it is very efficient. The space complexity of a min-heap is also O(log(n)) with the idea that the min heap will always be balanced.

III. Prim's Algorithm

3.1 Algorithm Concept and Implementation

The main objective of Prim's algorithm is to calculate the minimum spanning tree given a set of nodes and edges. A minimum spanning tree is a tree, or subset of all of the edges, that touches all nodes of any given connected graph without creating cycles and finds the minimum sum of edge weights possible. The main approach taken for Prim's algorithm is to maintain two different sets of nodes, the first of which being nodes that have not been visited by the algorithm. These nodes would not be in the minimum spanning tree while they are in that list, and the second tracking item would be nodes that have been previously visited. These nodes are already part of the minimum spanning tree so we know we may not go back to visiting those specific nodes. Each step of the algorithm will analyze the edge

weights and choose the lowest one that connects to another node that is not in the minimum spanning tree.

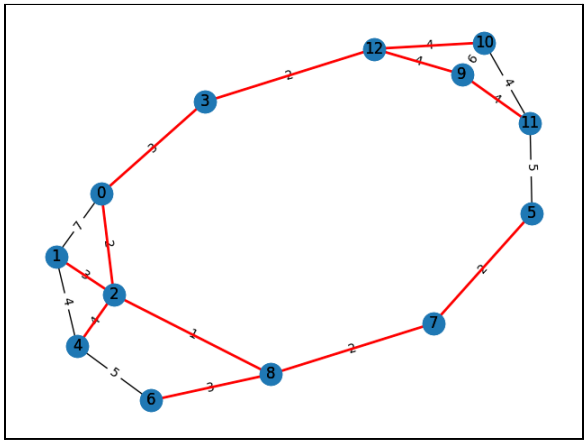

Figure 1: Prim's Minimum Spanning Tree

Figure 1, shown above, displays the output from the team's program on the calculation of Prim's minimum spanning tree. This specific output was calculated starting from node 0. It is also possible to add an edge between nodes and re-calculate Prim's. This can be seen in Figure 2 when an edge was added between nodes 8 and 12 with a weight of 4.

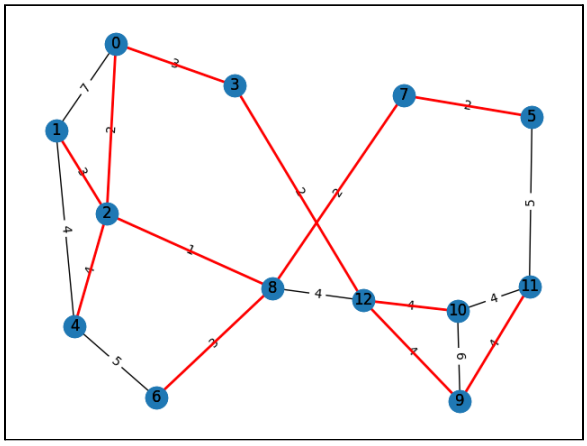

Figure 2: Prim's with Edge Addition

The team took an approach that allowed our code to be adaptable to both structure implementations so that the program could run the same function and by simply passing a boolean from the graphical user interface it would decide whether to use a min-heap or linked-list structure for the calculation. Step one in Prim's is defining an empty list that will end up storing the results of the algorithm. Then the program calls either the custom min-heap class or linked-list class depending on the boolean being true or false, which is determined by the selection of the radio buttons in the user interface.

Once the data structure being used is decided upon, the algorithm then pushes 0 as the starting node to the selected data structure. A few more variables are set including a counter to track how many nodes the algorithm has visited, a dictionary of nodes that have already been visited, a dictionary to keep track of which node the path came from, and finally a dictionary for map weights. While the data structure is not empty the program initializes the neighbor count to 0 and pops off the current node from the data structure and increments the neighbor count variable to track the number of nodes removed from the data structure. If the node's id is in the visited list then it will move on to the next node by calling the pop() function. The program appends the set of the node and the node it comes from to track the path of the minimum spanning tree. If the weight for the neighbors of the current node being analyzed is greater than 0, meaning that the path does exist, and the weight is not in the dictionary or the weight is less than the current weight recorded in the dictionary then the algorithm updates everything respectively to include that edge in the minimum spanning tree. The results are returned at the end to be fed into the graph to the user.

3.2 Prim's Complexity

The time complexity of the Prim's Algorithm is O((V+E)logV). If an adjacency list is used to represent the graph, then using breadth first search, all the vertices can be traversed in O(V+E) time. We traverse all the vertices of a graph using breadth first search and use a min-heap for sorting the vertices not in the minimum spanning tree. To get the minimum weight edge, we use min-heap as a priority queue. Min-heap operations, like extracting the minimum element, and decreasing a key value takes O(log(V)) time.

So, overall time complexity

= O(E + V) x O(logV)

= O((E + V)logV)

= O(ElogV)

This time complexity can be improved and reduced to O(E + VlogV) using a Fibonacci heap because each vertex is inserted in the priority queue only once and insertion in the priority queue takes logarithmic time.

IV. A* Heuristic Algorithm

4.1 Algorithm Overview

The A* heuristic algorithm, as mentioned before, is an algorithm that will determine the fastest path from some point or node, S, to a final node, E. This algorithm can be used in graph traversal in order to tell the user the best path from their starting point to their final destination. The A* heuristic algorithm is very similar to Dijkstra's shortest path algorithm in that it favors nodes that are close to the starting point, however it adds another concept on top of this that takes into consideration the nodes that are close to the destination or goal. Euclidean distance is the heuristic implemented in this program.

$$d(p,q) = \sqrt{\Sigma(q-p)^2}$$

Where p and q are two points in Euclidean space

There are 3 main aspects to the A* heuristic:

1. g-score: A cost associated with traveling along an edge from one node to another individually. This ends up being the summation of all the nodes that the algorithm has decided to visit from the starting node.

2. h-score: An estimated cost associated with traveling from the starting node to the final destination. This is an estimation because the true cost cannot be calculated until the algorithm has reached the final destination.

3. f-score: The summation of the g score and the h score.

The concept of this extra layer to Dijkstra's where the A* algorithm takes into consideration the nodes closer to the final destination is evident here with the addition of an h-score and f-score. In each step, the algorithm will choose the next route with the smallest f-score value because the f-score takes into consideration both g-score and h-score. Taking the smallest of the f-score is the best bet on making sure we are taking the best path we know how considering proximity from both the starting node and destination. It is very important that we do not *overestimate* the h-score, or heuristic value, because overestimating this can cause the algorithm to make a decision that is not the best decision for the overall path. If we overestimate h-score, we will be overestimating f-score which is not ideal since the goal is to choose the minimum f-score to determine the next movement.

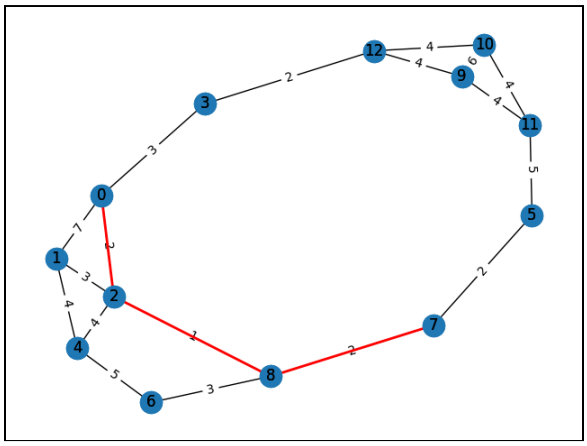

Figure 3: A Heuristic Algorithm Graph*

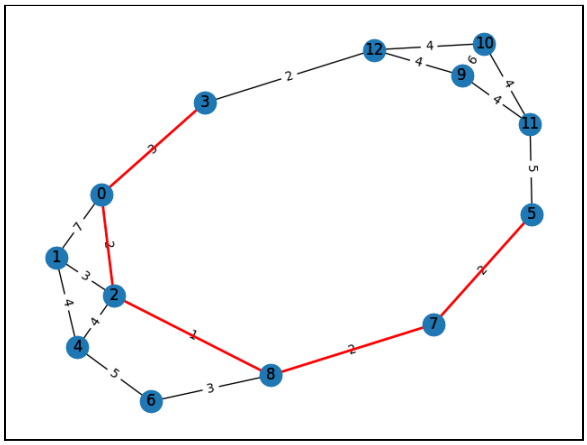

Figure 4: A Heuristic Algorithm Graph with Different Start/Stop Nodes*

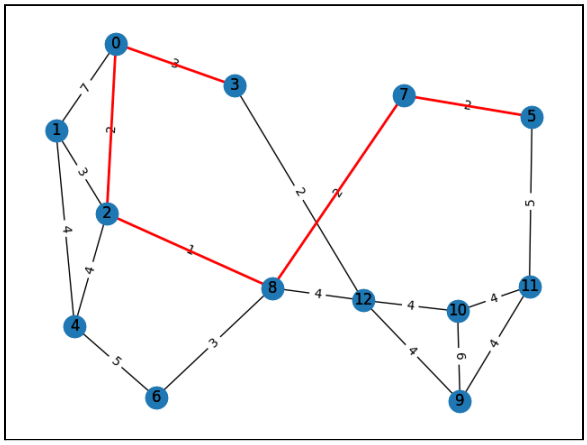

Figure 5: A Heuristic Algorithm Graph with Edge Addition*

3.2 A Algorithm Complexity*

The A* algorithm time complexity is not very clear cut or straight-forward because it is dependent on the heuristic used. In general, the time complexity of Dijkstra's Algorithm is O(V^2) but with a min-heap priority queue it improves worst case performance to O(V+ElogV). This same complexity should hold true for implementation of A* as it takes Dijkstra's and implements a heuristic to improve performance slightly. What sets A* apart from a greedy best-first search algorithm is that it takes the cost/distance of nodes already traveled through into account. This allows the algorithm to backtrack to a node as it comes up in the priority queue. If it would come up it would mean that a better F score is found and thus a better path is available.

V. Graphical User Interface Implementation

5.1 GUI Functionality Objective

The main objective of the graphical user interface was to provide a means of interacting with the developed algorithms that are sitting in the backend of the interface. The team aimed to let the user be able to choose specific start and end nodes so that the difference between these choices could be seen on the impact of the path being output by the program. In addition, the team set out to allow the user to incrementally add edges to the graph and see those added interactively. While the nodes do get placed in different arrangements on the graph when a new edge is made, the graph still holds all of its previous facts true in that all the same nodes are still connected in the exact same manner with the same exact weights as well.

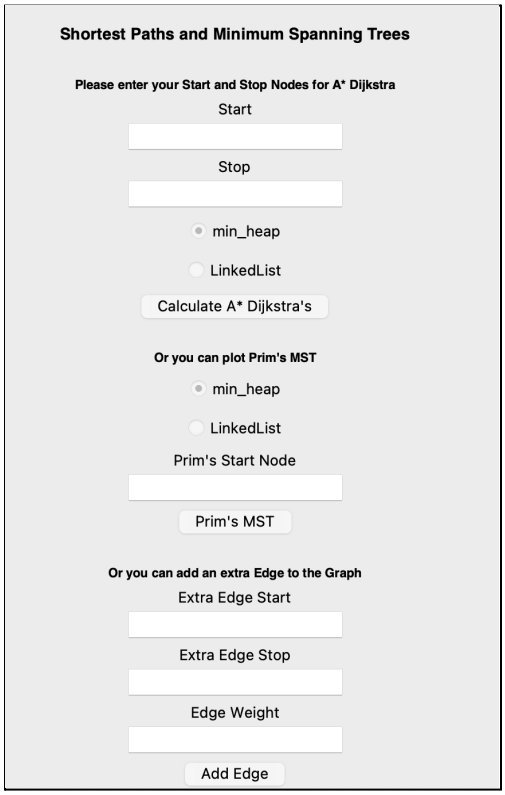

Figure 6: Graphical User Interface

5.2 Implementation

The graphical user interface was built using the tkinter package in Python, while the graph portion of the program to show the results was built using the networkx package. The whole program is initialized by the run of the GUI which has both algorithms and their respective data structure implementations tied to it in the backend so that it can be determined which algorithm to run with what data structure through the GUI functionality. When the GUI is run, it automatically creates an adjacency matrix of size 13. The window is then declared, named root, declared by root = tk.Tk() where tk is the tkinter package import alias. The GUI title and instructions are first set as window labels. The

main functionality of the GUI is set next with various buttons defined that are tied to different commands to run the different algorithms. All of the entry boxes such as start, stop, prim's start, extra edge start/stop, and edge weight are then defined and packed. The GUI must pack the objects so that the interface appears in a logical manner, when pack() is called it will pack the object relative to the last object packed. A StringVar() is used to initialize the variable that will be set to the radio buttons allowing the user to choose linked-list or min-heap. This variable is initially set to '1' so that min-heap is currently selected. The program then defines a list of min-heap being 1 and linked-list being 2 and for the items in that dictionary the program creates the radio button with the StringVar variable defined earlier. To end the build of the GUI, everything is packed according to how the team wanted to visualize the interface and root.mainloop() is called so that actions can be called while the program is running.

5.3 Challenges

For the most part, we didn't encounter any huge problems or obstacles during this project. One problem worth mentioning would be ensuring that the Python files and modules were imported and connected properly. When we had completed the programming of the project, we found there were redundant dependencies in the import of files and modules. This caused VSCode to run the same code multiple times during program execution.

The only other challenge we faced was ensuring that the program was compliant with MAC computers. We solved this with a simple import statement.

VI. Results and Discussions

6.1 Prim's Algorithm

In order for the team to verify that our results are outputting as expected, there were a number of unit tests applied. However, to visualize that our overall algorithm is outputting what is expected the manual calculation that the team knows to be true was completed to compare to the program output result. Figure 7 shows the initial graph exactly like the one in the program with all the same nodes and edge weights.

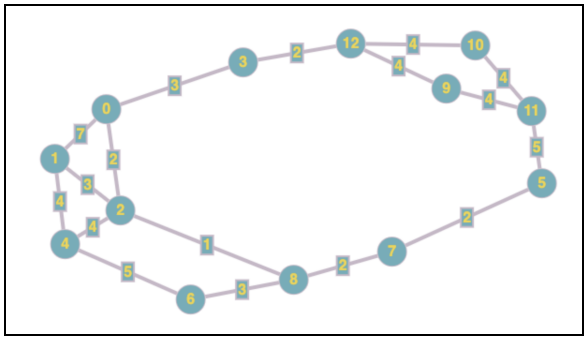

Figure 7: Initial Graph - No Edges Selected

The first step to completing Prim's algorithm is to choose a starting node to base the minimum spanning tree off of. For this calculation purpose, the starting node will be 0. Looking at all the edges connected to 0, the minimum is to be chosen so according to the algorithm at this step we would add the edge connection nodes 0 and 2 that have an edge weight of 2. Node 2 is then added to the minimum spanning tree, demonstrated by Figure 8.

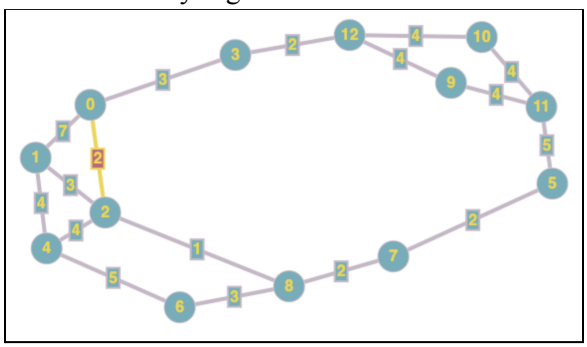

Figure 8: Prim's Algorithm - First Edge Selected

Continuing this pattern, the algorithm then would look at all edges connected to any node in the minimum spanning tree and choose the lowest edge weight given that it does not create a cycle within the graph. At this step the algorithm will choose the edge connecting node 2 to node 8 with a weight of 1. Figure 9 demonstrates this addition to the minimum spanning tree.

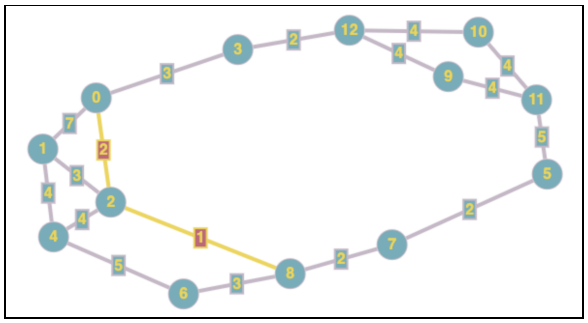

Figure 9: Prim's Algorithm - Second Step

The next couple of iterations follow the same pattern and the steps would be as follows:

- Step 3:
  - Nodes in the MST: {0, 2, 8}
  - Edge Chosen: [8, 7] Weight = 2
  - Node Added to MST: 7
- Step 4
  - Nodes in the MST: {0, 2, 8, 7}
  - Edge Chosen: [7, 5] Weight = 2
  - Node Added to MST: 5
- Step 5
  - Nodes in the MST: {0, 2, 8, 7, 5}
  - Edge Chosen: [8, 6] Weight = 3
  - Node Added to MST: 6
- Step 6
  - Nodes in the MST: {0, 2, 8, 7, 5, 6}
  - Edge Chosen: [2, 1] Weight = 3
  - Node Added to MST: 1
- Step 7
  - Nodes in the MST: {0, 2, 8, 7, 5, 6, 1}
  - Edge Chosen: [0, 3] Weight = 3
  - Node Added to MST: 3
- Step 8
  - Nodes in the MST: {0, 2, 8, 7, 5, 6, 1, 3}
  - Edge Chosen: [3, 12] Weight = 2
  - Node Added to MST: 12
- Step 9
  - Nodes in the MST: {0, 2, 8, 7, 5, 6, 1, 3, 12}
  - Edge Chosen: [12, 10] Weight = 4
  - Node Added to MST: 10
- Step 10
  - Nodes in the MST: {0, 2, 8, 7, 5, 6, 1, 3, 12, 10}
  - Edge Chosen: [12, 9] Weight = 4
  - Node Added to MST: 9
- Step 11
  - Nodes in the MST: {0, 2, 8, 7, 5, 6, 1, 3, 12, 10, 9}
  - Edge Chosen: [9, 11] Weight = 4
  - Node Added to MST: 11
- Step 12
  - Nodes in the MST: {0, 2, 8, 7, 5, 6, 1, 3, 12, 10, 9, 11}
  - Edge Chosen: [2, 4] Weight = 4
  - Node Added to MST: 4
- Step 13
  - Nodes in the MST: {0, 2, 8, 7, 5, 6, 1, 3, 12, 10, 9, 11, 4}
  - All Nodes have been added by this step

The steps above outline the complete calculation of Prim's minimum spanning tree algorithm which results in the selections in Figure 10.

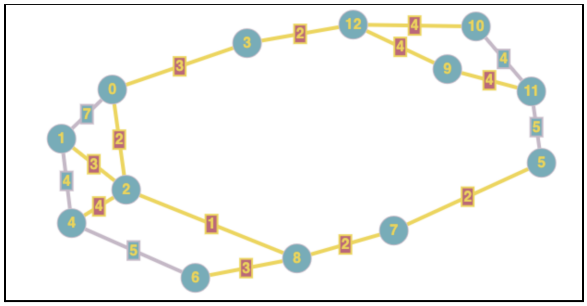

Figure 10: Final MST Edge Selections

It is evident that the edges selected in Figure 10 match those selected in Figure 3, proving that our algorithm did function correctly. It is important to note that the algorithm had the opportunity to make a decision of many equal decisions at some point in this algorithm where edge weights of the same magnitude exist. There could be multiple correct answers, but by breaking these ties we do get the program output as one of the valid correct outputs of Prim's algorithm proving to the team that we were successful in the implementation.

6.2 A Algorithm*

To test the results of the Dijkstra's A* Algorithm shown in Figure 4, the path was calculated manually to verify the heuristic and then used as a truth to compare to the output of the program.

Given that the start node is chosen to be 0 and the ending node is chosen to be 7. Since the A* heuristic uses a euclidean distance the manual graph was drawn to scale and a ruler was used to calculate the h-scores (distances between given node and the end node).

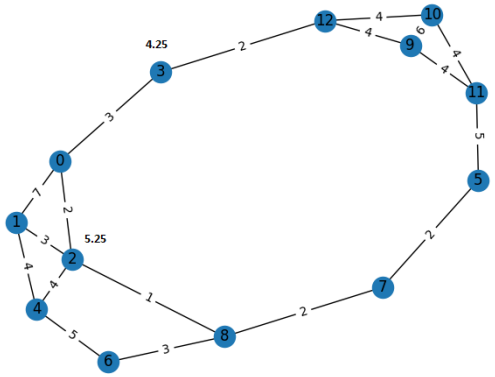

Figure 11. A Initial Graph - No edges selected*

The algorithm tells us to calculate the following first:

- Edge: [0,3]
  - G-score: 3
  - H-score: 4.25
  - F-score: 7.25
- Edge: [0,2]
  - G-score: 2
  - H-score: 5.25
  - F-score: 7.25
- Edge: [0,1]
  - G-score: 7
  - H-score: 6.5
  - F-score: 13.5

Since the g-score for the [0,2] edge is lower than that of the [0,3] edge or the [0,1] edge the algorithm adds the edge [0,2] to the path.

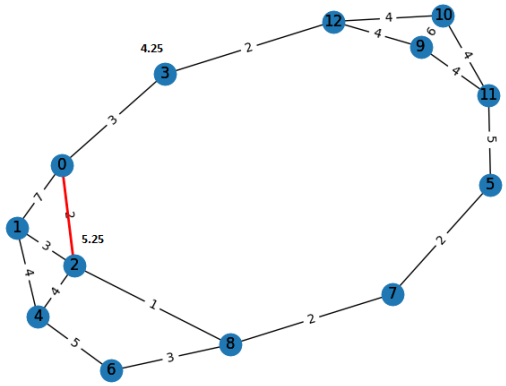

Figure 12. A - First Edge Selected*

Similar to what was done in step one the algorithm then calculates the g-score, h-score, and f-score for the edges [2,1], [2,4] and [2,8]. Edge [2,8] is chosen because it has the lowest g-score of 3 plus 2.25 for a total f-score of 5.25. This time the g-score is calculated to be the weight of the previous edges in the path plus the edge that is being proposed, so the addition of edge [2,8] has a weight of 3 because g-score of [0,2] is 2 and the g-score of edge [2,8] is 1.

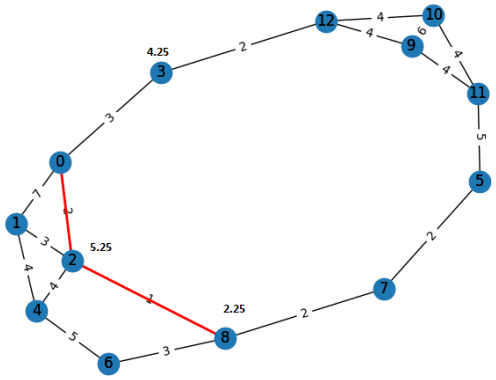

Figure 13. A - Second Edge Selected*

The third and final step the algorithm tests edge [8-6] and [8,7], since 7 is the end node the algorithm is complete and the cost of the path is the summation of the edges taken to get to the end node, it this case it is 2+2+2 for a sum of 6.

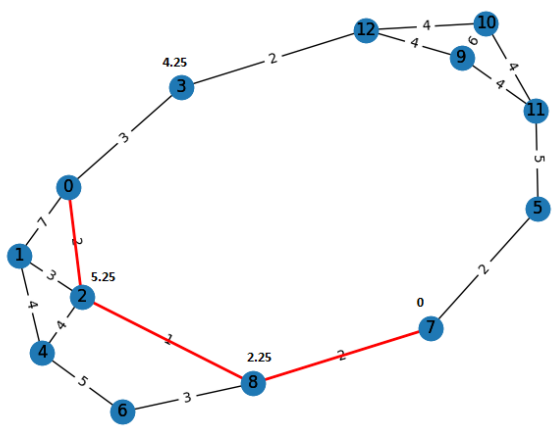

Figure 14. A - 3rd and Final Edge Selected*

The results of the manual calculation shown in Figure 14. When compared to the graph shown in Figure 4, they are identical thereby serving as reasonable proof that the algorithm implemented in the code works as expected.

6.3 Data Structure Performance

In order to see how each data structure stacks up against each other in this program, the team tested the Prim's algorithm time for an increasing number of nodes for both linked-list and min-heap. A random adjacency matrix was created for node sizes of 15, 50, 100, 500, 1000, 1500, 3000, 5000, and 8000. The timer was started after the matrix was made, and right before the Prim's algorithm was called. The timer was ended after the Prim's function was completed and the time was printed to the console in seconds. The code for testing this performance is shown in Figure 15.

```
from Prim import PrimMST
from adj_matrix import *
from random import randint
import time


g = Adj_Matrix(10000)

i = 0
for row in range(10000):
    for column in range(8000):
        weight = randint(0,6)
        g.add_edge(row,column,randint(0,6))

start = time.time()
PrimMST(g, 2, False)
end = time.time()
print(f'LL Time: {end-start:5.3f}s', end='  ')

start = time.time()
PrimMST(g, 2, True)
end = time.time()
print(f'MH Time: {end-start:5.3f}s', end='  ')
```

Figure 15: Performance Testing Code

The code in Figure 15 was run over the iterations of node sizes mentioned earlier, and the times were recorded for both min-heap and linked-list. The team then graphed these times to create a visual of the scalability each data structure has. This visual can be seen in Figure 16. It is clear from this graph that min-heap is much more scalable than linked-list as it stays more consistent in time performance in comparison to linked-list. This matches with what the team expected to see since we know it to be true that the big O time complexity of a linked-list data structure is O(n) while the min-heap data structure is O(log(n)). Seeing this result further proved to the team that the data structures were acting correctly within the respective algorithms.

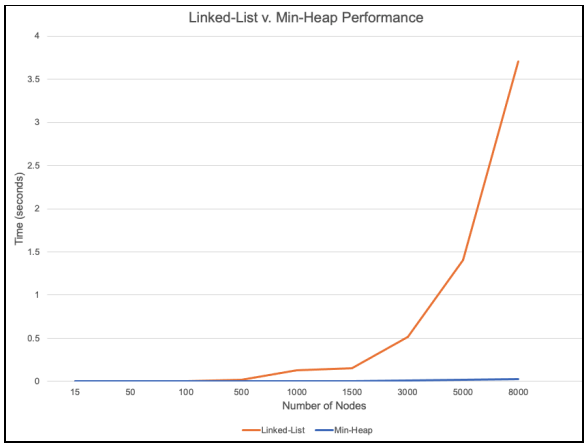

Figure 16: Linked-List Versus Min-Heap Performance

VII. Ideas for Future Research

Given that the project would go on for longer and the team had more time to make enhancements to the code base, there are a few ideas the team would have liked to explore. The team was able to get a graph that shows the result in the graph highlighted in red and also allow the user to add edges that it populates live, but the difficulty of incrementally showing the steps of choosing every edge within the algorithm was unable to be accomplished. The team could have explored ways to break up the algorithm to return intermittently to populate the graph, or maybe a better package could have been implemented for this aspect of the project. The team would have also explored different IDEs to use over VSCode that allowed the team to optimize team meetings and work offline in our own time, in unison. The utilization of Jupyter Notebook could have been beneficial in sharing a notebook much like a google doc so all code was always live, but the team was not sure what version control that would have and worried about overwriting working code with that approach.

VIII. References

1. A* search algorithm. Wikipedia. 2018. http://www.ccpo.odu.edu/~klinck/Reprints/PDF/wikipediaNav2018.pdf
2. Common Data Structure Operations. Big-O Algorithm Complexity Cheat Sheet. 2013. https://www.bigocheatsheet.com/
3. Prim's Algorithm. GateVidyalay. 2020. https://www.gatevidyalay.com/tag/prims-algorithm-time-complexity/