Shortest Path and Minimum Spanning Trees

Matteo Bassani - Dean Golden - Rocio Krebs - Nathan Rayon

Term Project [Spring, 2023]

The University of West Florida

Apr 15, 2023

COP 6416 Advanced Algorithms

Dr. Katie Broadhead

Table of Contents

1 Executive Summary

In this project we implemented the A* heuristic variation of Dijastra's shortest path algorithm and Prim's minimum spanning tree algorithm. These two algorithms utilize a Priority Queue which a user can specify the Linked List version or the Binary Min Heap version. For the programming language, we went with Python to develop this project. With Python, we have a wide range of libraries necessary for this project network visualization, network management, and testing.

2 Requirements

- pytest
    - Testing framework to ensure classes/functions worked as expected
- matplotlib
    - Used for visualization/animation of Romanian network and used to show benchmarking results
- tkinter
    - Used for GUI to get user input for run configuration
- networkx
    - Used to generate graphs for benchmarking and managing the graph used for visualization
- etc
    - Various other libraries are listed in requirements however they are there as the dependencies for the previously mentioned libraries

3 Priority Queues

In a priority queue, elements are stored based on their priority. The element with the highest priority is stored at the front of the queue using the function enqueue, and the function dequeue will deliver the element with the highest priority.
There were two implementations: Binary Min-Heap and Linked List.

3.1 Binary Min-heap

A Binary Min-Heap ensures that the min value is always the root of the tree. This is achieved with the enqueue and dequeue functions. The enqueue function (O(logn)) inserts an element into the bottom of the tree (represented by an array) and then looks at its parent and if it's lower than the parent swaps positions and it repeats this process until it is larger than the parent or the root. The dequeue function (O(logn)) pops the tree's root off, takes the last element in the tree, and puts it in its place. Then it looks at both its children and swaps with the smallest one it repeats this process as far down the tree as it needs to. These two functions ensure that the root of the tree is always the smallest element.

3.2 Linked List

A linked list can implement a priority queue. Each node in the linked list contains an element and its priority. The linked list is ordered based on the priority of the elements, with the highest priority element at the front of the list.

The class Node represents a single node in the linked list in the project. Each node has a value attribute that stores the element's value, in our case, a city name and a priority attribute that stores its priority. The PriorityQueueLL class is the main class that holds the linked list with all the functionalities.

The enqueue function adds elements to the list sorted by priority, it runs O(1) if the list is empty, or the new head has higher priority than the head node. In the worst case O(n) if the new node has lower priority than all existing nodes in the linked list, and we need to traverse the entire linked list to find the correct position to insert the new node.

The dequeue function removes and returns the element with the highest priority from a priority queue. The runtime of the dequeue method in a priority queue implemented as a linked list is O(1) in the best and average cases, and O(n) in the worst case, where n is the number of nodes in the linked list.

4 A* Shortest Path & Prim's MST (Minimum Spanning Tree)

4.1 A* algorithm

This is a modified version of the original Dijkstra algorithm incorporating a heuristic function to guide the search for the shortest path. This algorithm combines the advantages of Dijkstra's algorithm (finding the shortest path between two nodes in a graph with non-negative edge weights) and the Greedy Best-First-Search algorithm (using a heuristic function to prioritize exploration toward the goal node).

The heuristic function in our project contains the straight-line distance from a given node to "Bucharest."

But let's get deeper into our implementation of A*:

- The **add_weight** function initializes the shortest path to a newly reached node and the route to get there.
- The **update_open_weight** function checks if a better path is found to get to a node that has been reached before but never visited. If a better path is found, the best path is updated.
- The **update_closed_weight** function checks if a better path is found to get to a node that has already been visited. If true, all the best paths to its neighbors must be updated.

- Finally, the **AStar** implements the A* algorithm. The function initializes the start node and enqueues it. Then, it loops through the nodes in the priority queue, dequeues them, and adds them to the visited list. If the current node is the end node, the function returns the shortest path to the end node, its length, and the ordered list of all the nodes visited during the search. If the current node has not been visited, the function adds neighboring nodes to the queue and updates their paths. If the current node has been visited, the function recursively updates its neighbors' paths.

Since the heuristic never overestimated the shortest distance to the goal node, it is considered admissible. This means that our implementation of A* algorithm is guaranteed to find the optimal solution.

The time complexity of A* depends on the heuristic. In the worst case of an unbounded search space, the number of nodes expanded is exponential in the depth of the solution *d*: O(*bd*), where *b* is the branching factor.

However, the heuristic function significantly affects the functional performance of A* search since a good heuristic allows A* to prune away many of the *bd* nodes that an uninformed search would expand. Therefore, good heuristics are those with a low effective branching factor (the optimal being b* = 1).

4.2 Prim’s minimum spanning tree algorithm

Prim’s algorithm is a greedy algorithm used to find a minimum spanning tree of a particular weighted undirected graph. The minimum spanning tree must include all the vertexes of the original graph, but the sum of the edges must be minimized.

Our implementation of Prim’s algorithm utilizes an adjacency matrix and either a priority queue or a binary min-heap.

As for functions, the **Prims** function is the main function used to compute the minimum spanning tree. This function takes in a graph data structure as mentioned above to handle which nodes have been visited and to add to the chosen queue. The start variable and queue are also passed into this function with the start being the starting city or node the algorithm should start on and the queue being the chosen queue implementation.

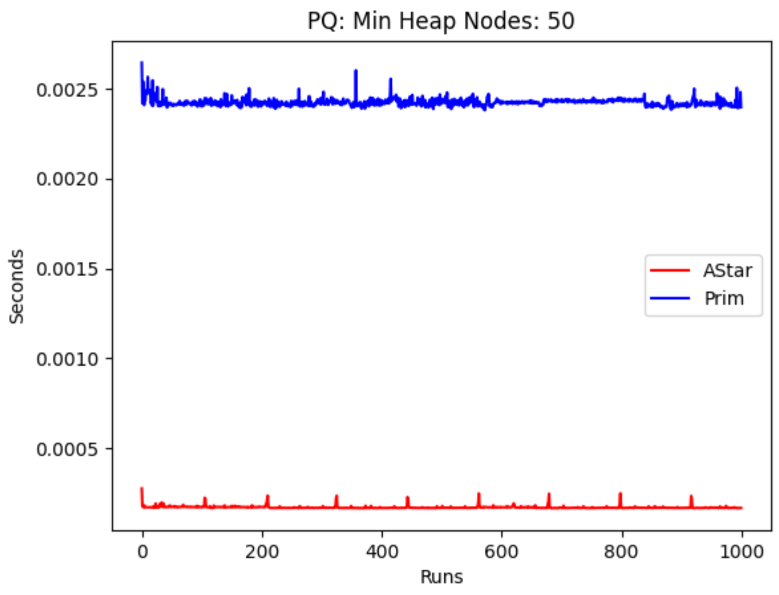
At the start of our implementation of Prim’s, the starting node is inserted into the chosen queue and we will create an empty minimum spanning tree. Next, we will enter the main loop of the algorithm which will terminate if the queue is empty. In this loop, we will pop the lowest weight node in the queue and add this to the minimum spanning tree, then we will loop through all of the chosen nodes and add them to the queue. Since our queue has the nodes with the lowest weight at the beginning, each time we pop from the queue we will ensure that tree we are building is minimal.

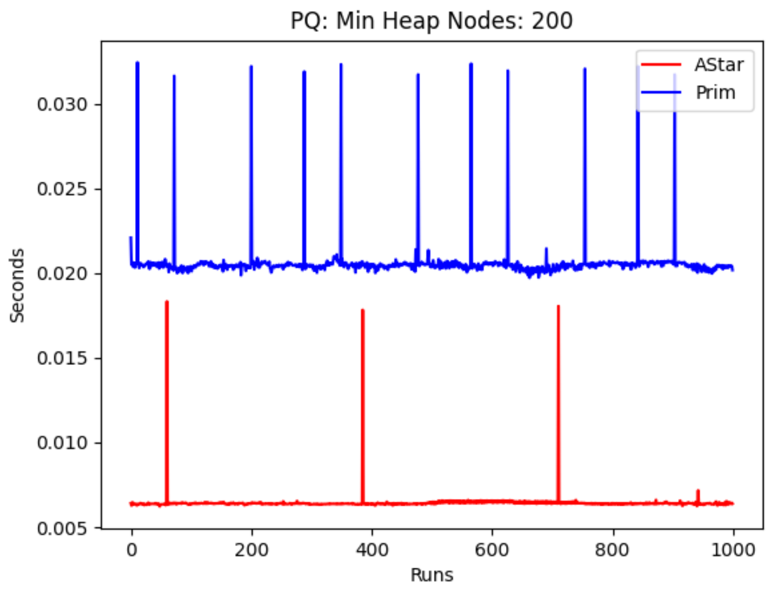
The time complexity of Prim’s algorithm is heavily dependent on the data structures used to store the graphs and edges. Since our implementation uses a priority queue to store the weights of each node, we come up with O(ELog(V)) with E being the number of edges and V being the number of vertices.

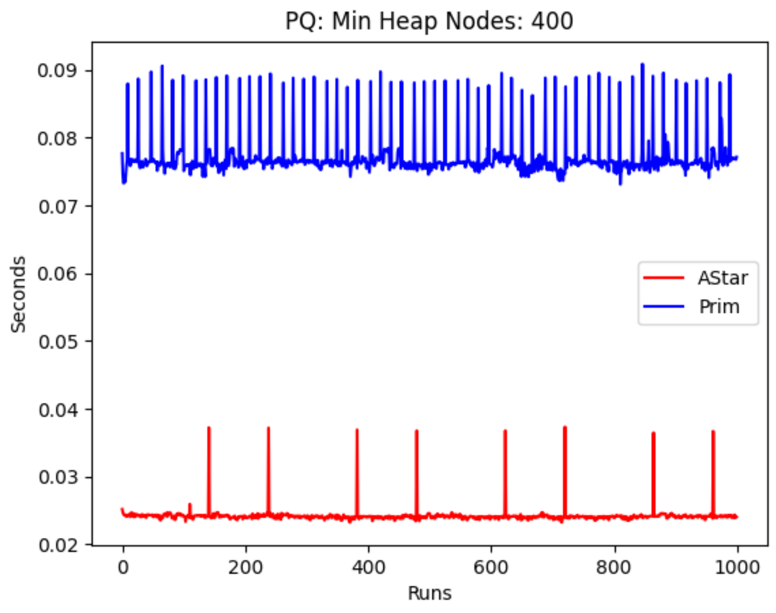
5 Benchmark

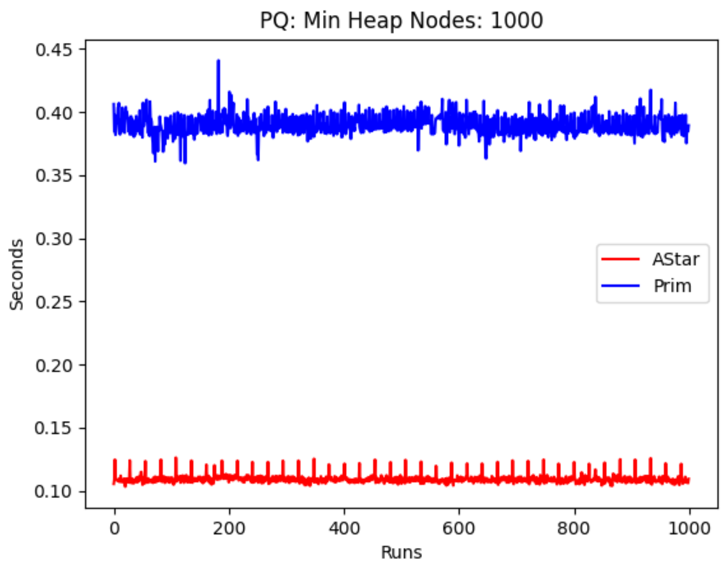
These benchmarks are achieved by generating a random graph via networkx then adding weight and heuristic for each node/edge. After that, we then run the algorithms on the graph. This is done as many times as the user defines; however, the higher, the better, as it shows clear trends and makes outliers noticeable. The user also determines the size of the graph, and we show different sizes to show differences in the algorithm's performance.

5.1 Binary Min-heap

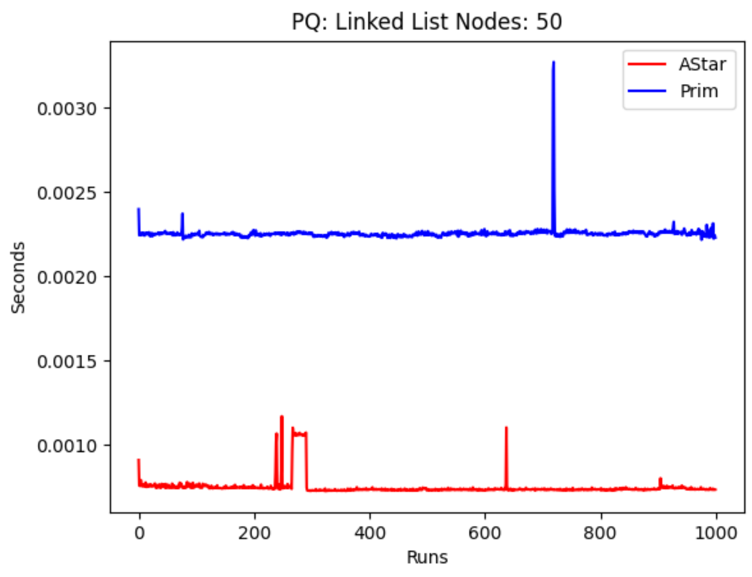


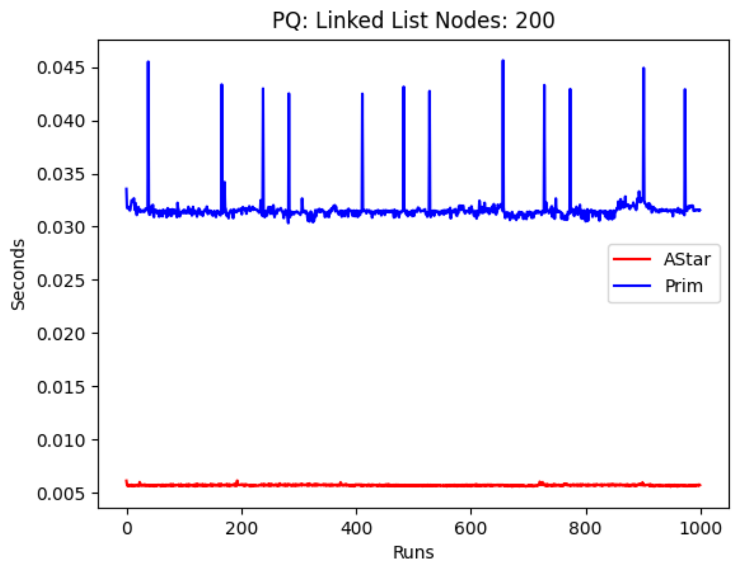


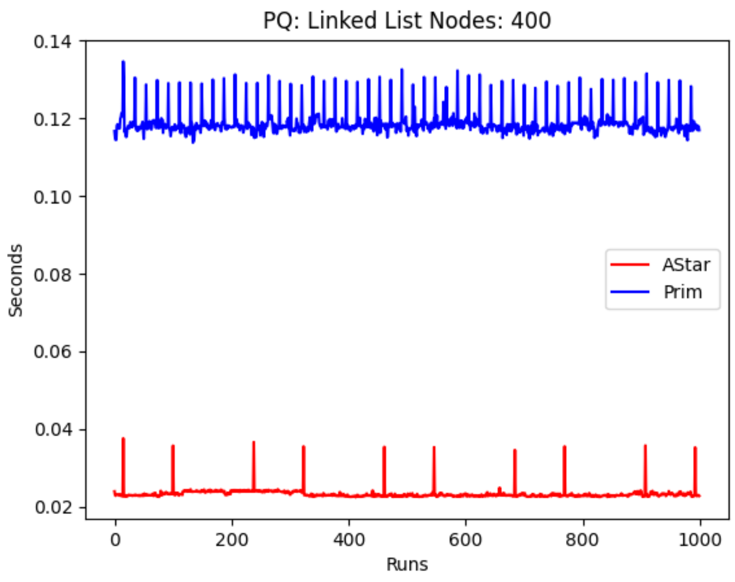

PQ: Min Heap Nodes: 400
Seconds
0.09
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0
200
400
600
800
1000
Runs
AStar
Prim

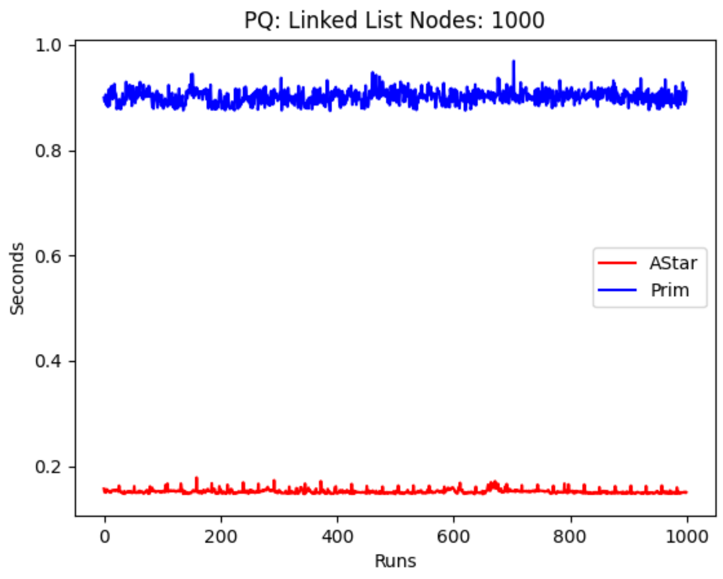

PQ: Min Heap Nodes: 1000
Seconds
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0
200
400
600
800
1000
Runs
AStar
Prim

5.2 Linked List









5.3 Conclusion

From these graphs we can come to a few conclusions. The bigger the graph the longer it takes the algorithms to find the shortest path/MST. A* consistently beats Prims. The time it takes Prim's algorithm to find the MST grows at a much faster rate than A* which is just finding the shortest path from one node to another. However, if we needed to keep finding the shortest path from different nodes within the same graph Prims would have an advantage as we have already generated a MST. Another thing is that the difference between the Priority Queues are not as noticeable at first with the small graphs with the large ones you can tell that Binary Min-Heap takes the edge due to its enqueue and dequeue time being faster than the Linked List implementation.