CS110 Final Project

A comprehensive exploration of Minimum Spanning Trees

Rhali Attar

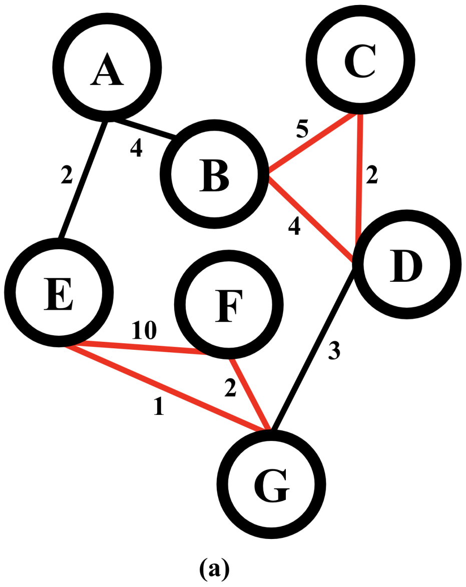
April 21 2022

Contents

Networks are everywhere. Whether we are looking at particles at the micro level, society at a meso level, or galaxies at the macro level, everything around us is so intrinsically connected that we could model it using Graph Theory. The question is, what do we do with this network once we have it? Fortunately mathematicians and computer scientists have been researching this subfield since Euler solved the Königsberg bridge problem in 1735 and we can now analyze networks in great depth. This paper will investigate one of their most important properties and the backbone of many applications of Graph Theory: the Minimum Spanning Tree (MST).

Background

A network is a set of nodes (or vertices) connected by edges (or arcs) that carry a weight. Edges can be directed which would mean that we can go from a node to another but not the other way around. However, we do not need to worry about directions for the purpose of this paper. A simple example is a network of cities as nodes, connected by their roads which have a length. The standard notation is $G = (V, E)$ where V denotes the set of vertices and E, the set of edges. A spanning tree is a subnetwork that spans G, which means that all its edges must also belong to G and it must include every node in G. However, due to its classification as a tree it has the added constraint that it must not have any cycles which occur when we are able to start at a node and come back to it when walking along edges. A network has many spanning trees, but the ones of most interest are the ones that minimize the sum of the weights of the edges: the MSTs. There are two algorithms that allow us to find these: Kruskal's and Prim's.



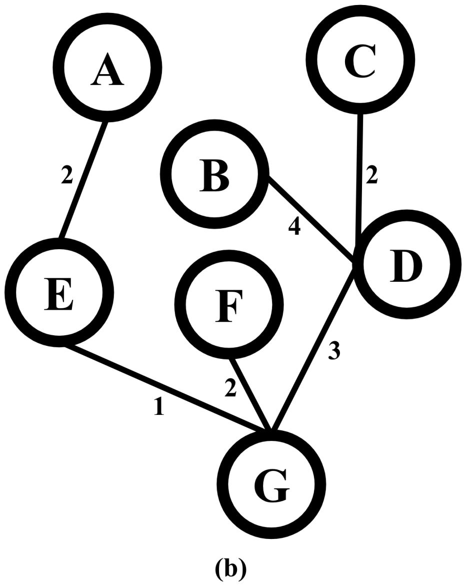


Figure 1: An example of a network (a) and one of its MSTs (b) (note that we could have included edge AB instead of edge BD and gotten another valid MST). Cycles in the network are highlighted in red.

Before getting into them though, let's go through how we can represent a network numerically. There are two main ways to do this, we can either use an adjacency list or an adjacency Matrix. An adjacency list is a list that contains lists of all the nodes that each node is connected to and the weight of the edge that connects them. For example, the graph in Figure 1 would have the following adjacency list:

$$A : [(B, 4),\ (E, 2)]$$

$$B : [(A, 4),\ (C, 5),\ (D, 4)]$$

$$C : [(B, 5),\ (D, 2)]$$

$$D : [(B, 4),\ (C, 2),\ (G, 3)]$$

$$E : [(A, 2),\ (F, 10),\ (G, 1)]$$

$$F : [(E, 10),\ (G, 2)]$$

$$G : [(D, 3),\ (E, 1),\ (F, 2)]$$

On the other hand, an adjacency matrix of a graph with n nodes is an $n \times n$ matrix where each entry represents an edge. To create such a matrix, we need to number the nodes in an arbitrary way and if there is an edge that goes from node i to node j, we store its weight in row i, column j. If there is no edge from node i to node j though, we fill row i, column j with a 0 instead. So, the graph in Figure 1 would have the following adjacency matrix:

$$\begin{bmatrix} 0 & 4 & 0 & 0 & 2 & 0 & 0 \\ 4 & 0 & 5 & 4 & 0 & 0 & 0 \\ 0 & 5 & 0 & 2 & 0 & 0 & 0 \\ 0 & 4 & 2 & 0 & 0 & 0 & 3 \\ 2 & 0 & 0 & 0 & 0 & 10 & 1 \\ 0 & 0 & 0 & 0 & 10 & 0 & 2 \\ 0 & 0 & 0 & 3 & 1 & 2 & 0 \end{bmatrix}$$

When considering optimizing space complexity, it is clear using adjacency lists would be more appropriate because it only stores connections while the adjacency matrix also stores a 0 when there isn't a connection. However, since it is much easier to access edge weights in an adjacency matrix (to find the edge going from node i to node j we can just call AdjacencyMatrix[i][j] but to find that same edge in a adjacency list we need to scan through everything in AdjacencyList[i]), it will be used more in this paper.

1 Kruskal's Algorithm

1.1 Algorithmic strategy

Despite its slightly intimidating name, Kruskal's algorithm is really straightforward! When given a graph G, Kruskal's starts by creating a list with all the edges in the G and an empty adjacency matrix that will represent the MST. It then removes the smallest edge in the list and checks what would happen if we add it to the MST.

If it would create a loop, the algorithm discards it but if it does not, it stores it in the MST's adjacency matrix. It then keeps repeating this process until there are no more edges left in the edge list. The flowchart in Figure 2 below visualizes this step by step process.

This seems too easy though, right? Well, that's thanks to the magic of greedy algorithms. Indeed, Kruskal's holds the greedy property because it always reaches the globally optimal solution (the MST) but at each step, it only optimizes for the current step (by adding the next smallest weighted edge that does not create a cycle) without considering how that will affect the next ones.

1.2 Python Implementation and Data Structures

Now that we know how the algorithm works, we can code it! To do so, I decided to use Object Oriented Programming with a Node and a Graph class. This was especially useful as it allowed me to easily store and update the attributes of these classes (e.g. accessing and adding to the adjacency matrix when necessary or getting a node's arbitrarily assigned number). It also just makes the code more readable and reusable.

Let's discuss the choices I made regarding data structures. For this code, I had choices to make about how I could store two essential variables: the list of edges, and the adjacency list. We know that we need to somehow be able to access the smallest element in the edge list, which means that we can either implement it a sorted array or a minimum priority queue. While both options would affect the time complexity of Kruskal's by the same amount (because sorting a list of size n or pushing n elements

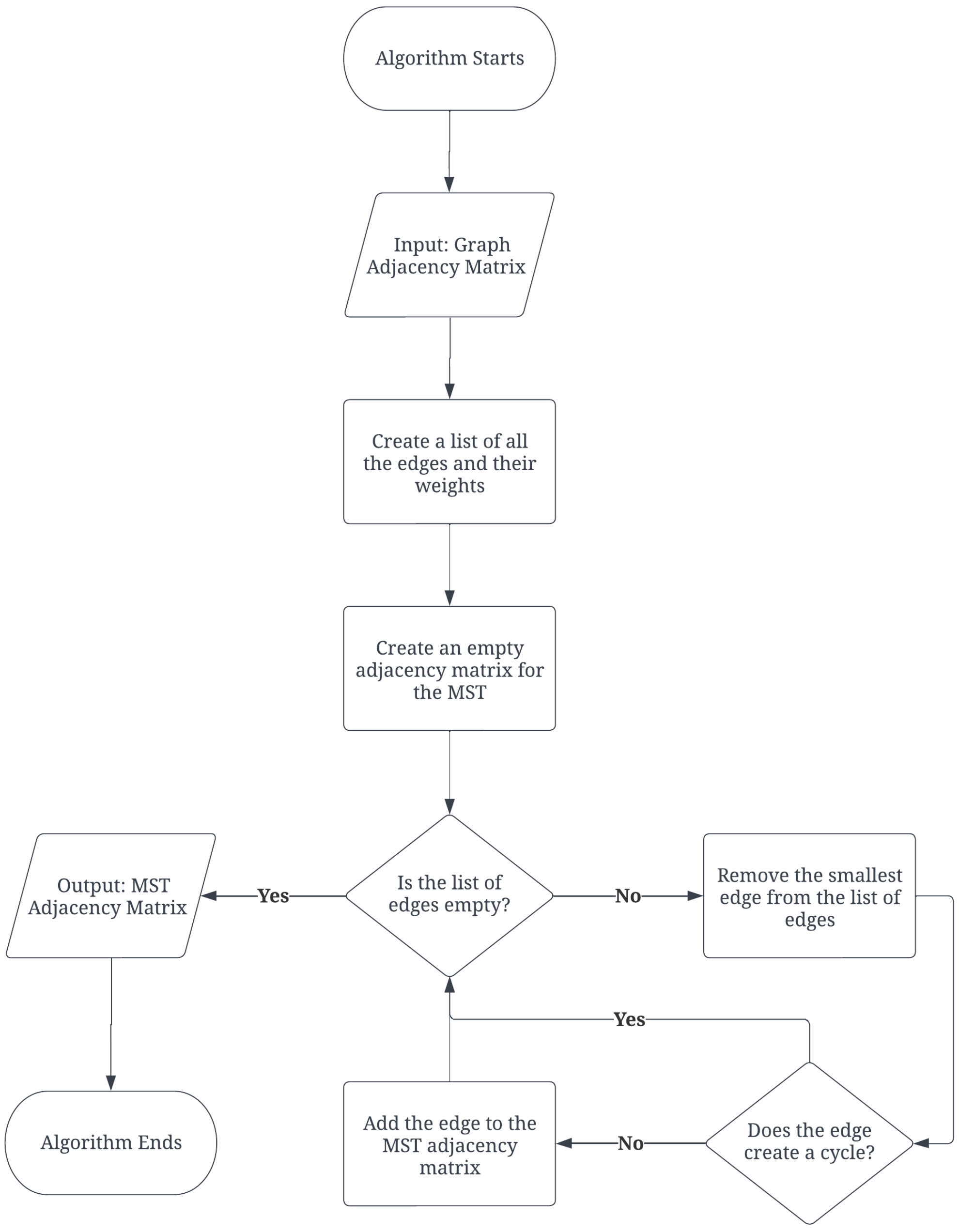

Algorithm Starts
Input: Graph Adjacency Matrix
Create a list of all the edges and their weights
Create an empty adjacency matrix for the MST
Is the list of edges empty?
Yes
Output: MST Adjacency Matrix
Algorithm Ends
No
Remove the smallest edge from the list of edges
Does the edge create a cycle?
Yes
No
Add the edge to the MST adjacency matrix

Figure 2: Flowchart for Kruskal's Algorithm

into a heap both scale with $O(nlog(n)))$, using a minimum priority queue is a more dynamic approach. This is due to the fact that if we decided to add an edge to our graph we can just push it into the queue but if we had used a list, we would have had to sort it all over again. Moreover, the adjacency list could have been coded using either a dictionary or a simple array. What's useful about dictionaries in this context is that we can very easily find and access a node's adjacencies. However, while this might provide a more sophisticated solution, since every node has a number associated with it, we can simply just store the adjacency list as an array of arrays where array$[i]$ represents node i's adjacencies. This solves the problem while keeping the implementation more simple and therefore more accessible.

Testing

I came up with four types of test cases that will inform me on whether this algorithm works properly or not. Firstly, I will try it on a random graph I created to make sure that it works for average cases. I will then see what happens when inputting a graph with no edges and a graph with no nodes. In these situations, we should not get any errors but should instead get an empty adjacency list as an output. I will then test it on a complete network (one where every node is connected to each other) as not only will this test that the algorithm works on a very dense network, but it will also really put my CycleFinder method to the test. Lastly, I will test it on a disconnected graph because it is an edge case that Kruskal's should be able to find a solution for. Indeed, Kruskal's is able to find a forest of disconnected MSTs because the edge found at a specific step does not have to be connected to the edge that will be found at the next step.

1.3 Complexity Analysis

1.3.1 Theoretical

Let's analyze my implementation of the algorithm to try to figure out how it should scale with input size. Looking at the loop in the flowchart, we know that the algorithm goes through every single edge, which means that the time complexity must be proportional to the number of edges. Inside the loop however, we need to check for cycles which will also affect the time complexity. My CycleChecker method starts by creating two adjacency sublists which should scale with the total number of vertices ($|V|$) because the number of adjacent nodes $leq|V|$. It then goes through one of the lists for every element of the other list which should therefore scale with $|V| \cdot |V|$ since the size of each list is proportional $|V|$. So putting this all together, we get a time complexity of $O(E(V + V^2)) = O(EV^2)$.

1.3.2 Practical

For the practical analysis, I created 3 simulations. The first one keeps the number of nodes constant and increases the number of edges and vice versa for the second one. Therefore, theoretically, the first one should have a time complexity of $O(E)$ and the second should have a time complexity of $O(V^2)$. The third one increases both the number of nodes and the number of edges by carrying out the algorithm on complete graphs. In the third case, $|E| = \frac{|V|(|V|-1)}{2}$, so theoretically, it should have a time complexity of $O(V^4)$.

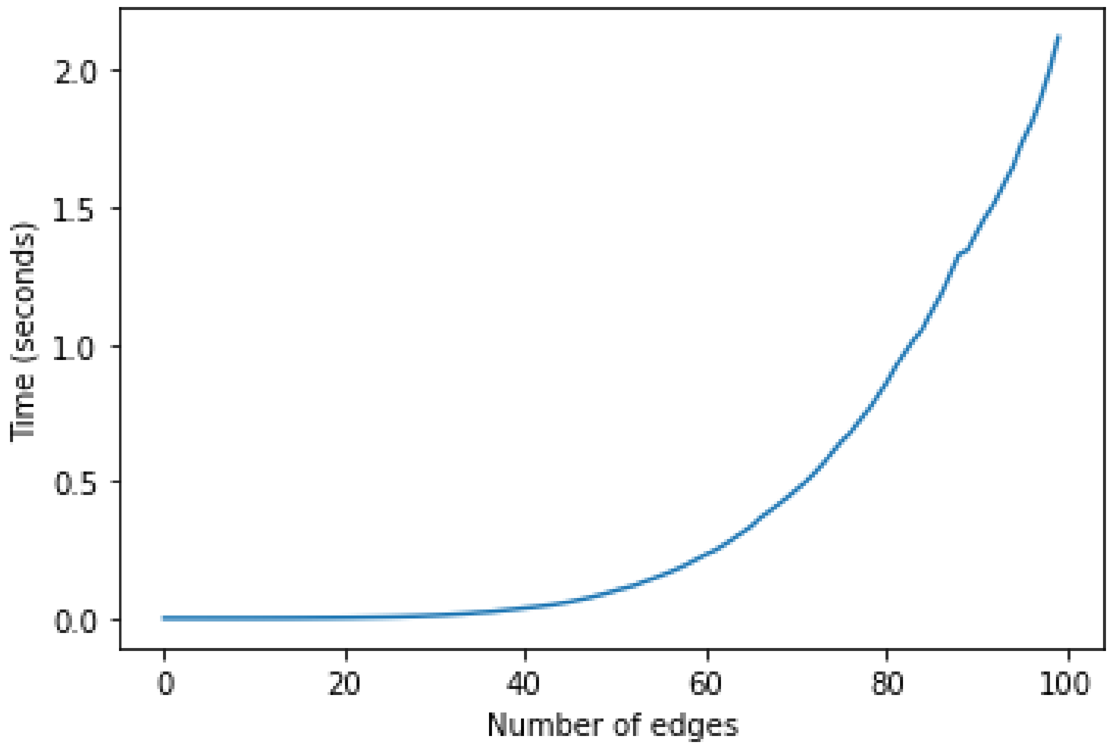


Figure 3: Graph of the scaling of Kruskal's when keeping the number of nodes constant

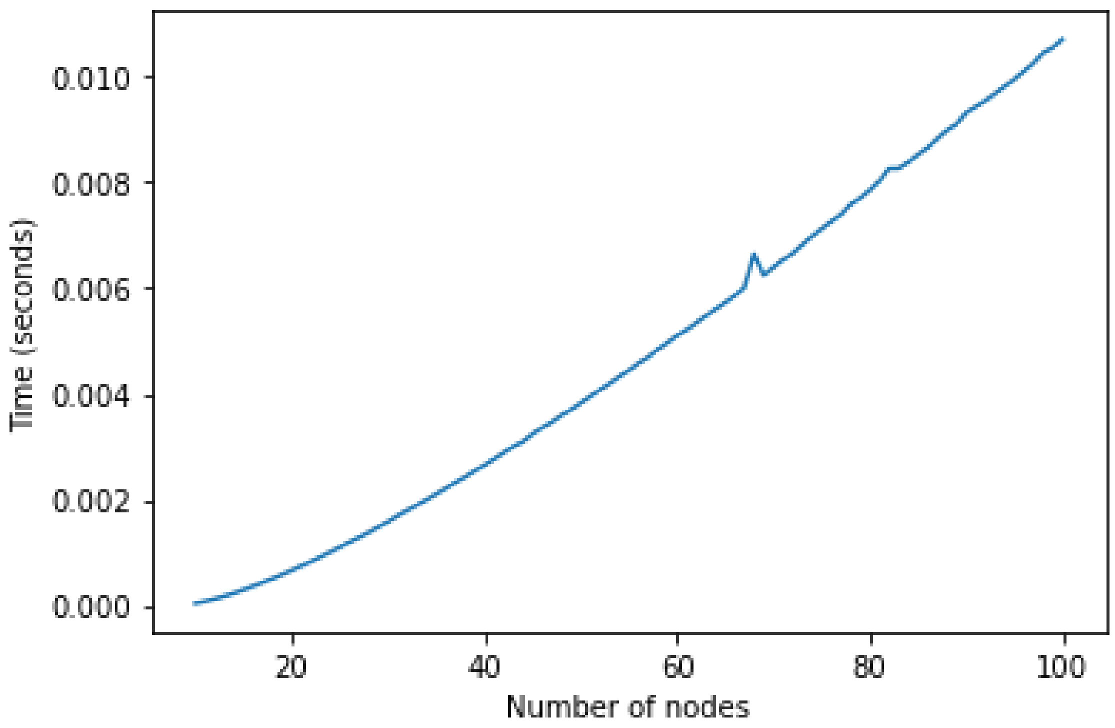


Figure 4: Graph of the scaling of Kruskal's when keeping the number of edges constant

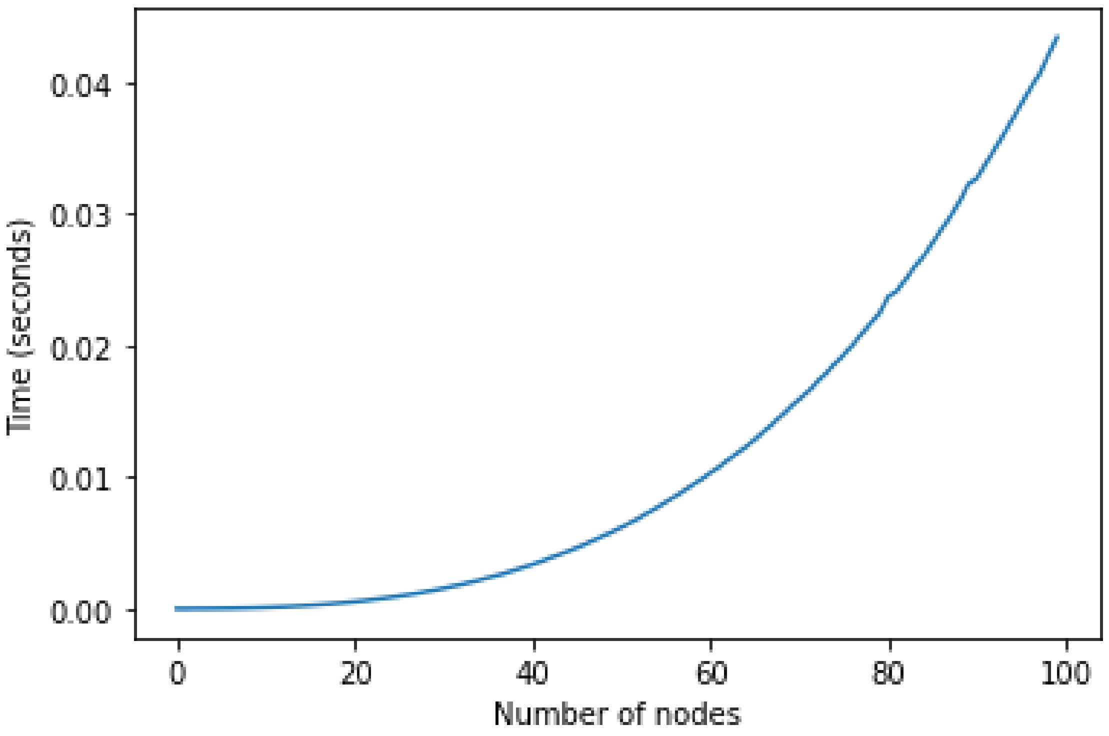


Figure 5: Graph of the scaling of Kruskal's when running it on complete graphs

1.4 Critique

Let's discuss the elephant in the room: the time complexity. According to our Introduction to Algorithm textbook (page 633), Kruskal's algorithm should have a time complexity of $O(ElogE)$ or $O(ElogV)$, which is significantly better than my implementation [Cor+09]. The reason why that is the case is due to my CycleChecker method as it has a time complexity of $O(V^2)$. Moreover, this method has some additional issues that must be discussed. Indeed, it is only able to detect direct loops such as $A \to B \to C \to A$, but not ones with more nodes in between A. While this is an edge case and won't cause issues in most situations, it should still be improved. One method could be to use depth first search to ensure that we can find every type of loop. This would also improve the overall time complexity to $O(E(V+E))$ as it has a time complexity of $\Theta(V+E)$. While this is still not fully optimal, it would already improve it a lot.

Moreover, I have to mention the space complexity issues with this implementation. Firstly, the Class has both an adjacency list and an adjacency matrix as attributes, even though they share the same information which is a waste of space. Also, regardless of whether a Node instance has already been created for a specific node, anytime an edge that includes this node is created it will require us to make a duplicate. Lastly, the Kruskal's implementation requires many list/priority queue duplicates which take up additional space.

Another minor critique of this implementation is that nodes have to be labeled with a single character with the first one being an A which could be limiting for data analysis.

2 Prim's Algorithm

2.1 Algorithmic strategy

Now let's explore this easier to pronounce algorithm. When given a graph G, it starts by creating two lists, one for nodes that have been visited, and one for nodes that have not been visited. Initially, we haven't visited any nodes so the visited nodes list is empty and the non-visited nodes list contains all the nodes in the graph. The algorithm then picks a random node (the one it picks isn't important since all nodes will end up in the MST anyway), removes it from the non-visited list and adds it to the visited list.

It then looks through all the edges that connect visited nodes to non visited nodes, finds the smallest one and records it in the adjacency matrix. We have now visited one new node, so it removes it from the non-visited list and adds it to the visited list. It then keeps repeating this process of connecting non-visited nodes to visited nodes until all nodes have been visited.

Once again, this seems to good to be true, and once again, it's thanks to the greedy property. Just like Kruskal's, Prim's chooses the locally optimal solution at each step (finds the smallest connection between visited and non-visited nodes) without considering the impact this could have on the next steps, and still reaches the globally optimal solution (the MST).

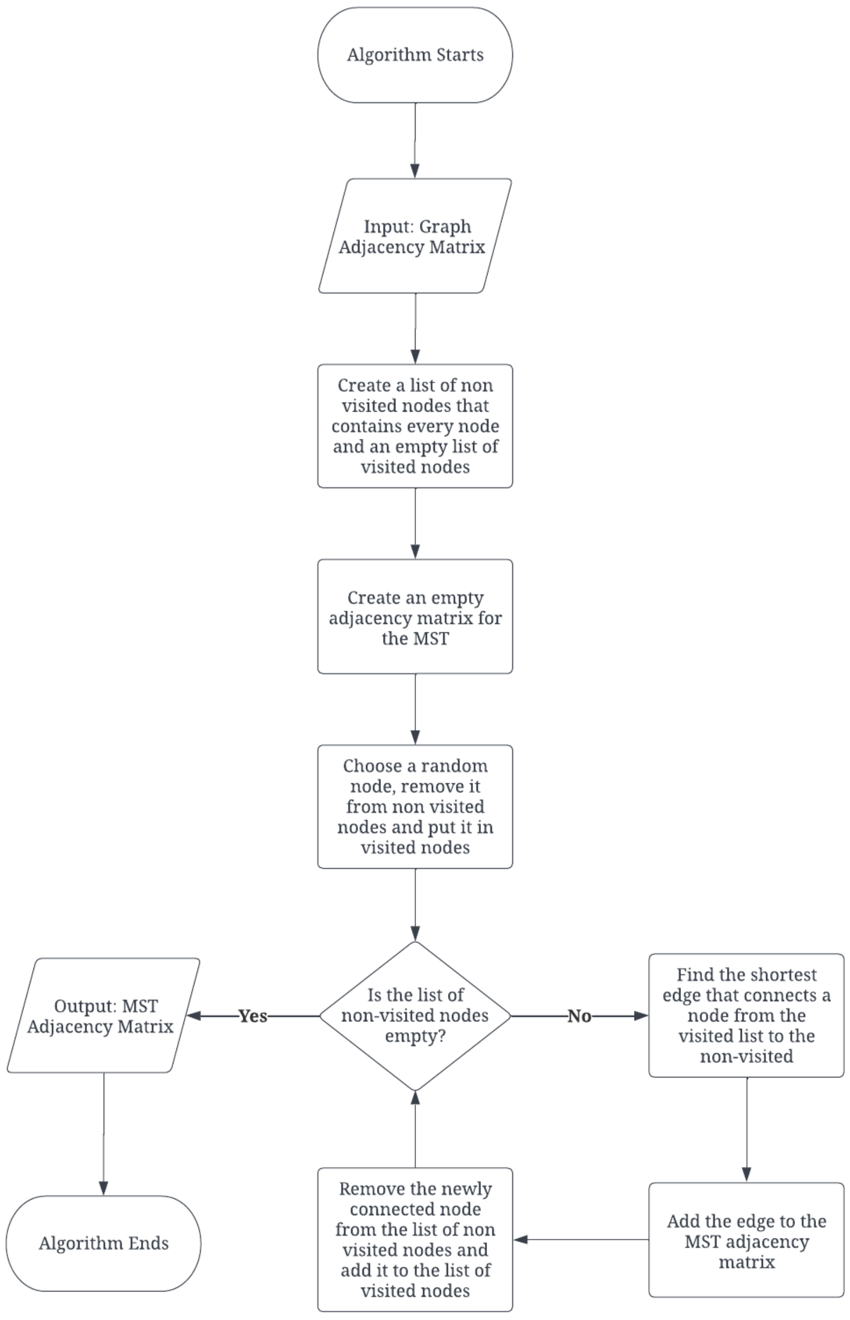

Algorithm Starts
Input: Graph Adjacency Matrix
Create a list of non visited nodes that contains every node and an empty list of visited nodes
Create an empty adjacency matrix for the MST
Choose a random node, remove it from non visited nodes and put it in visited nodes
Is the list of non-visited nodes empty?
Yes
No
Output: MST Adjacency Matrix
Algorithm Ends
Find the shortest edge that connects a node from the visited list to the non-visited
Add the edge to the MST adjacency matrix
Remove the newly connected node from the list of non visited nodes and add it to the list of visited nodes

Figure 6: Flowchart for Prim's Algorithm

2.2 Python Implementation

Remember how I mentioned that using Object Oriented Programming increased code reusability? Well, all I needed to do to code this algorithm was to add it as a method into the Graph class and I could reuse any relevant attributes and methods.

Testing

The tests I used for Kruskal's are also applicable here since they should both have the same outputs so I reused them. The only difference is that if the input graph is disconnected, Prim's will only end up finding the MST of one of the connected subgraphs instead of being able to find a forest of MSTs like Kruskal's. This is because, as Prim's is in the process of building it, the MST is always connected, so if all the nodes in one subgraph have been visited, there will be no way to connect them to the other nodes and the algorithm will promptly end. I still decided to test Prim's on a disconnected graph, though, to see if my code would break.

2.3 Complexity Analysis

2.3.1 Theoretical

Looking at the flow chart, we know that the loop will repeat itself $|V|$ times because we remove one node from the non-visited nodes list at each step until there are none left. Within that loop, we also find the shortest edge that connects the two lists. To do this, we go through the relevant row of the adjacency matrix (which has $|V|$ elements) of each node in the visited nodes list. So, the time complexity of each step of the loop is found by multiplying the current number of nodes in the visited nodes list (which increments by one at every step) by $|V|$. To find the overall time complexity, we just need to sum up all of these individual time complexities. Therefore, the time complexity of this Prim's algorithm implementation should be:

$$O\left(\sum_{i=1}^{V} Vi\right) = O\left(V\sum_{i=1}^{V} i\right) = O(V\frac{V(V+1)}{2}) = O(V^3)$$

2.3.2 Practical

I used the same simulations as the ones in the Kruskal's algorithm section, and here are the results:

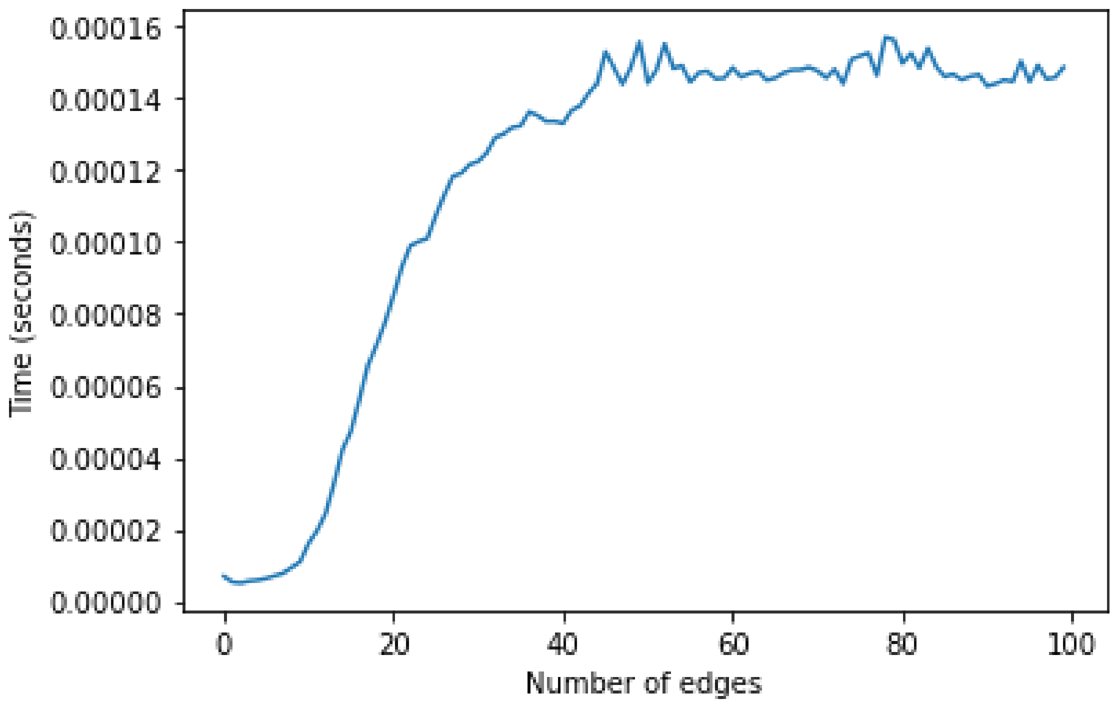


Figure 7: Graph of the scaling of Prim's when keeping the number of nodes constant

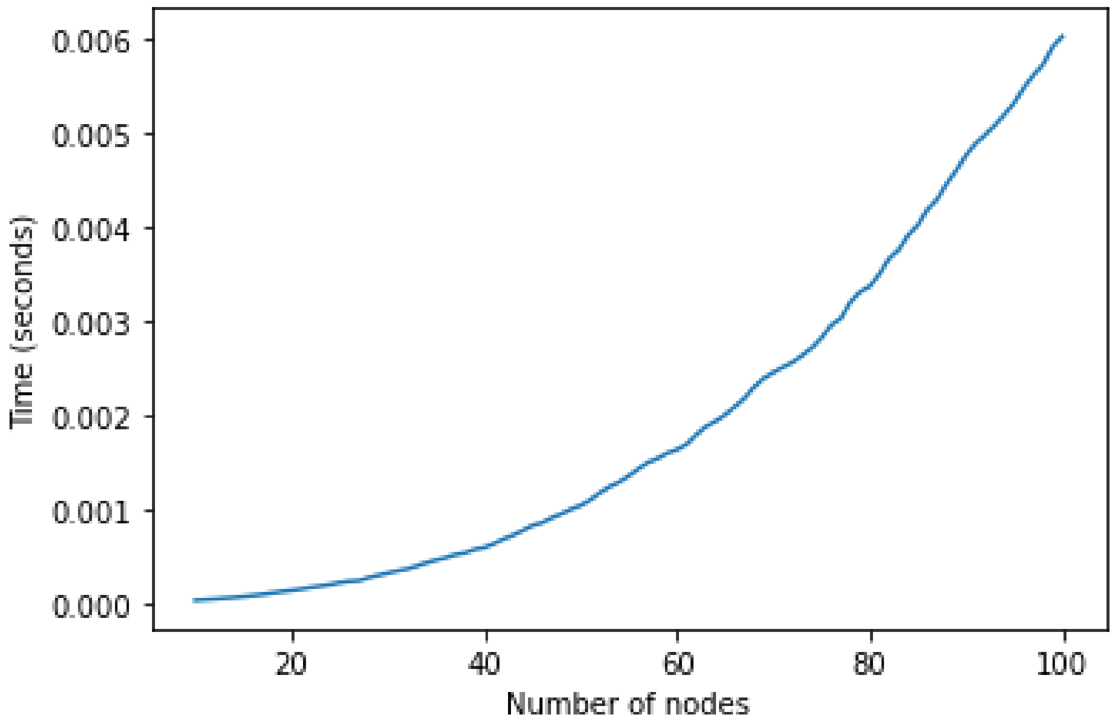


Figure 8: Graph of the scaling of Prim's when keeping the number of edges constant

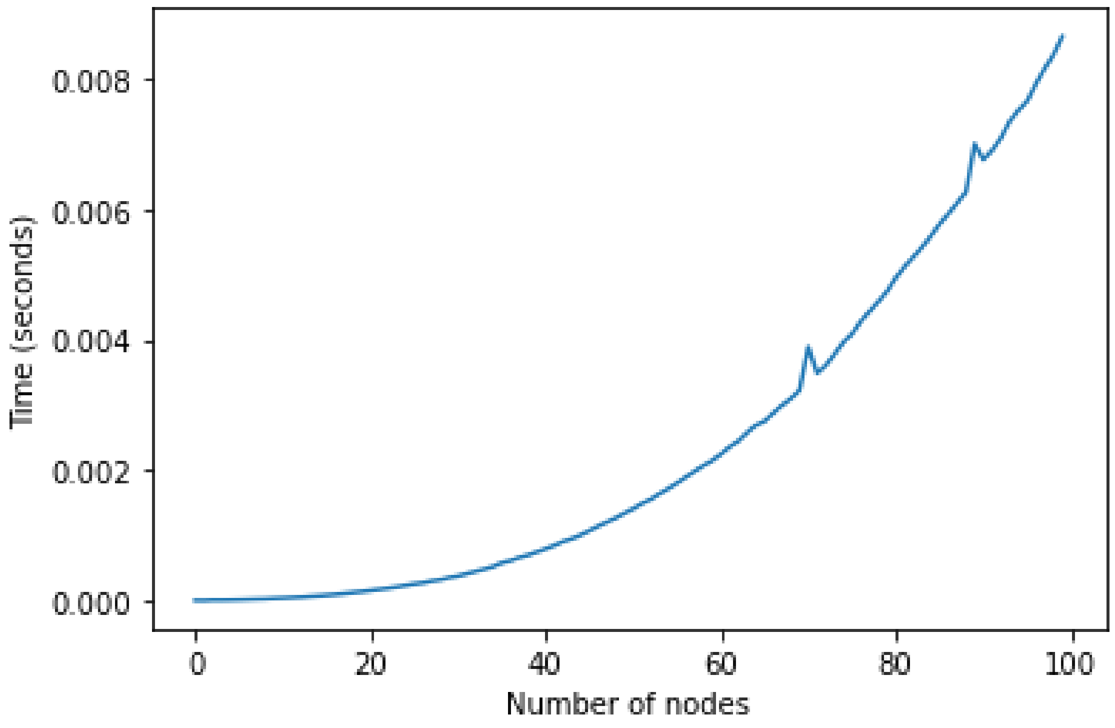


Figure 9: Graph of the scaling of Prim's when running it on complete graphs

2.4 Critique

Once again, the time complexity could (and should!) definitely be improved. Indeed, according to page 636 Introduction to Algorithms, Prim's algorithm can have a time complexity of $O(ElogV)$ [Cor+09]. The reason why my time complexity is so high is definitely due to the method I used to find the shortest edge that connects the two lists. Indeed, as we saw in the theoretical time complexity calculation, this part of the algorithm contributes quadratically to the time complexity.

Moreover, since this is within the same Class as the Kruskal's implementations, the more general critiques also apply here.

3 Kruskal's or Prim's?

3.1 According to this implementation

We can generate the following graphs based on the simulations I created to better compare the simulations:

So, we can clearly see that the Prim's implementation performs much better than the Kruskal's implementation, both practically and theoretically. Additionally, no CycleChecker method is required for Prim's, making it more easily readable. Therefore, we can reasonable conclude that Prim's wins the fight against Kruskal's within the context of these implementations.

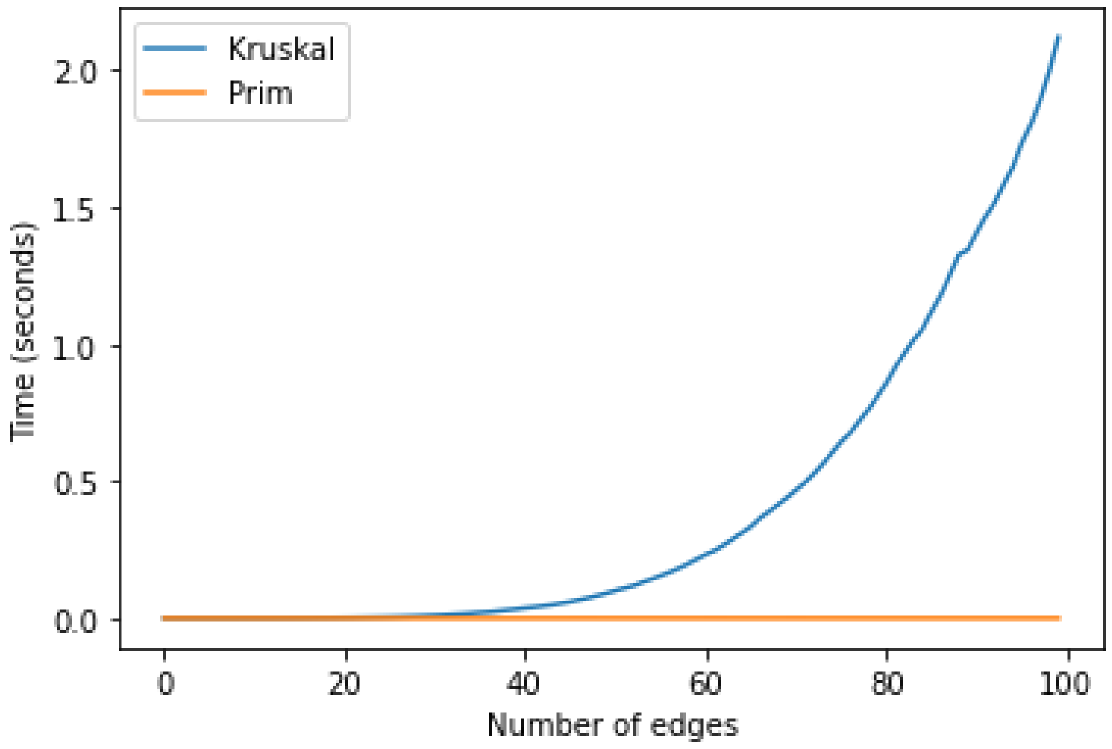


Figure 10: Graph of the scaling of both algorithms when keeping the number of nodes constant

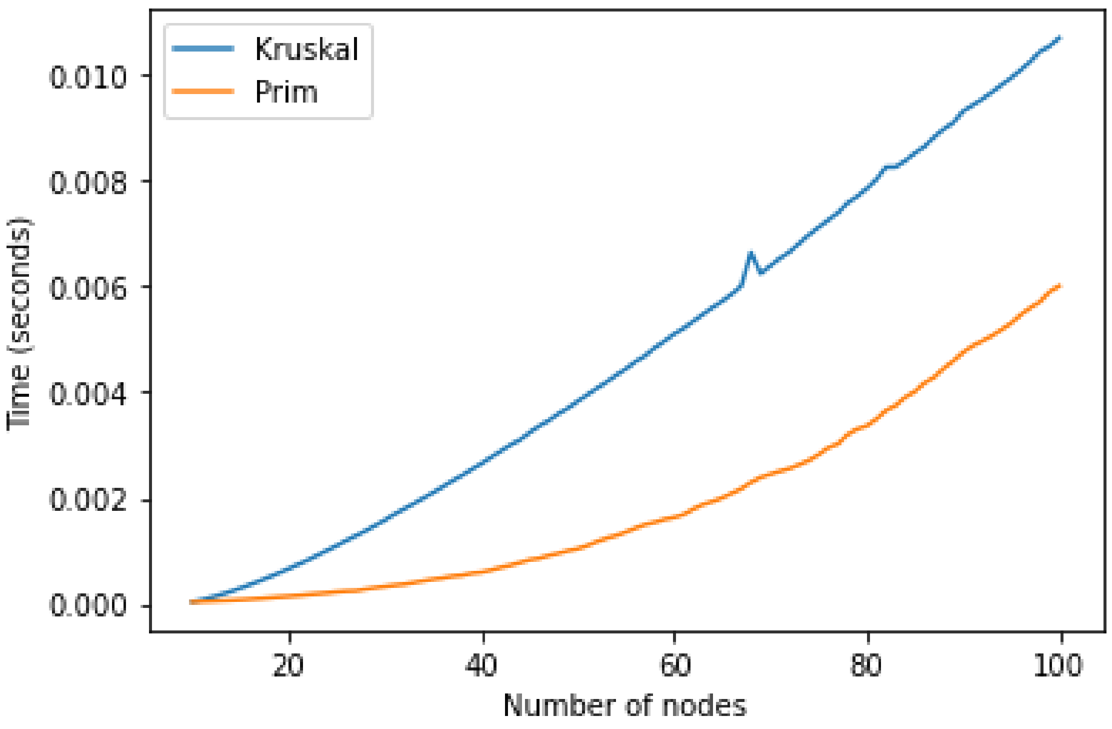


Figure 11: Graph of the scaling of algorithms when keeping the number of edges constant

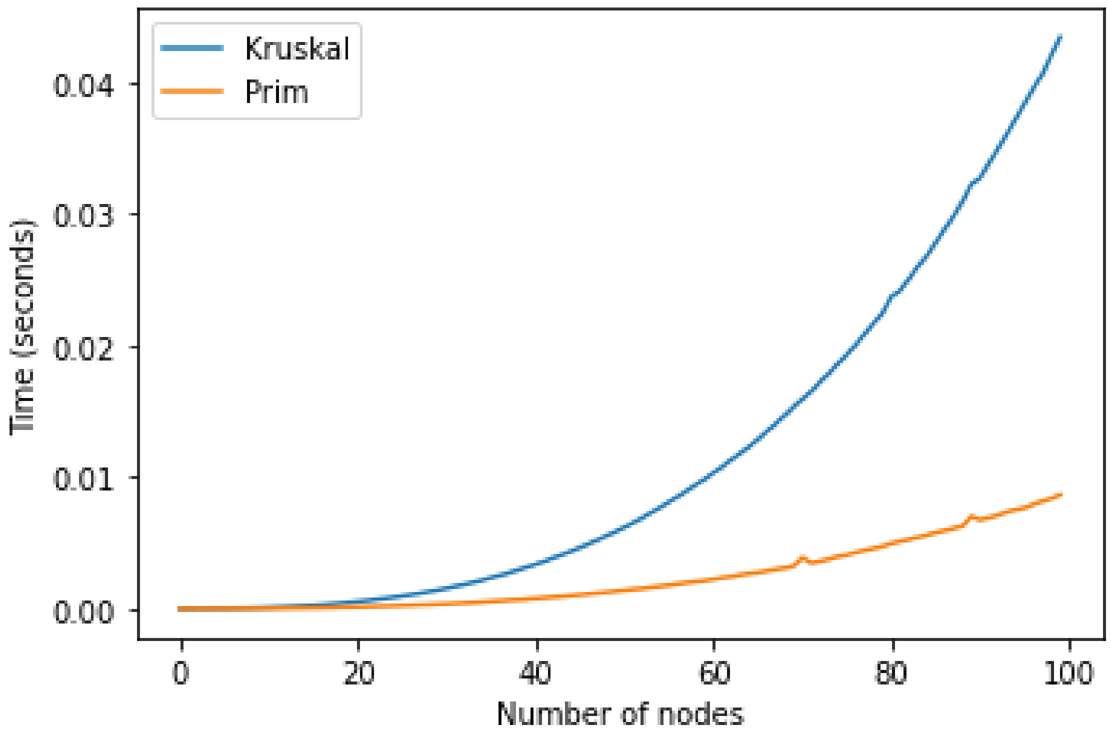


Figure 12: Graph of the scaling of algorithms when running them on complete graphs

3.2 More generally

More generally though, we can't necessarily make this easy of a conclusion. As mentioned previously, the actual time complexities of both of these algorithms are meant to be $O(ElogV)$, so their performance is actually the same. We therefore need to dive a little deeper in order to find the one other significant difference between them. This one is the fact that Kruskal's works on disconnected graphs while Prim's doesn't, as explained above. This gives Kruskal an important edge (no pun intended) because we do not necessarily know what the inputted graph looks like, especially if there are a large number of nodes, and therefore, we can't always know whether it is connected or not. Therefore, Kruskal wins in a more general context.

A Prim's and Kruskal's implementations code

```python
import heapq
import copy

class Node:
    """
    A class that implements relevant properties of nodes

    Attributes
    ----------
    name : str
        A one letter string representing a unique node

    number: int
        An integer representing the ASCII code - 65 (so that A = 0)
        of the name of the letter. Used as the index of the adjacency
        list/array that represents the node.
      """

    def __init__(self, name):
        self.name = name
        self.number = ord(name) - 65

    def __lt__(self, other):
        """
        Overloaded operator so that we can use the heapq module with instances
        of this class.

        Parameters
        ----------
            other: Node
                The node we want to compare the current node to
        Returns
        ----------
            Bool
                The truth value of whether the other node is larger than the current
        """
        return (self.number < other.number)

class Graph:
    """
    A class that implements an undirected network

    Attributes
    ----------
    n : int
        The number of nodes in the graph
    """
    def __init__(self, n):
        self.n = n
        self.adjacency_matrix = [[float('inf') for _ in range(n)] for _ in range(n)]
        self.adjacency_list = [[] for _ in range(n)]
        self.edges = []
```

```
def add_edge(self, node1, node2, weight):
    """
    Add a weighted undirected edge between two nodes to the Graph
    Parameters
    ----------
        node1: Node
            the first node

        node2: Node
            the second node

        weight: float
            the weight of the edge
    Returns
    ----------
        None
    """
    #add to the adjacency matrix
    self.adjacency_matrix[node1.number][node2.number] = weight
    self.adjacency_matrix[node2.number][node1.number] = weight

    #add to the adjacency list
    self.adjacency_list[node1.number].append(node2)
    self.adjacency_list[node2.number].append(node1)

    #heaappush into the priority queue of edges
    heapq.heappush(self.edges, (weight, node1, node2))

def add_edge_list(self, edge_list):
    """
    Add a list of weighted undirected edges to the Graph

    Parameters
    ----------
        edge_list: lst
            list of 3-tuples each representing an edge in the form
            (node1, node2, weight)
    Returns
    ----------
        None
    """
    #go through each edge
    for edge in edge_list:

        #use the add_edge method to add the current edge to the Graph
        self.add_edge(edge[0], edge[1], edge[2])
```

```python
def CycleCheck(self, node1, node2, adjacency_list):
    """
    Checks if adding an edge between node1 and node 2 creates a cycle

    Parameters
    ----------
        node1: Node
            the first node

        node2: Node
            the second node

        adjacency_list: lst
            the adjacency list of the graph we want to check
    Returns
    ----------
        Bool
            True if the edge creates a cycle, false otherwise

    """
    #get node 1 and node 2's adjacencies
    node1_connections = adjacency_list[node1.number]
    node2_connections = adjacency_list[node2.number]

    #initialize lists where we can fill up the
    #names of the adjacent nodes for simpler comparisons
    node1_connections_easy = []
    node2_connections_easy = []

    #fill up node1_connections_easy
    for connection in node1_connections:
        node1_connections_easy.append(connection.name)

    #fill up node2_connections_easy
    for connection in node2_connections:
        node2_connections_easy.append(connection.name)

    #check all of node 1's adjacencies
    for connection in node1_connections_easy:

            #if it is also adjacent to node 2, there is
            #a cycle
            if connection in node2_connections_easy:
                return True

    return False
```

```python
def Prim(self):
    """
    Implements Prim's algorithm on the Graph to find
    an MST

    Parameters
    ----------
        None

    Returns
    ----------
        lst
            The adjacency list of the Graph's MST

    """
    #if there are only 0 or 1 nodes, there are no edges so
    #return the empty adjacency list
    if self.n < 2:
        return self.adjacency_list

    #create an empty adjacency list for the mst
    mst_adjacency_list = [[] for _ in range(self.n)]

    #create a copy of the Graph's adjacency matrix
    adjacency_matrix_copy = copy.copy(self.adjacency_matrix)

    #initialize the non_visited_nodes list by filling it
    #up with the nodes' numbers
    non_visited_nodes = [i for i in range(self.n)]

    #initialize the visited_nodes list by putting the first element
    #of the non_visited_nodes list
    visited_nodes = [non_visited_nodes.pop(0)]

    #while there are still some unvisited nodes
    while non_visited_nodes:

        #initialize a smallest edge
        smallest_edge = (float('inf'), None, None)

        #check every visited node
        for node in visited_nodes:

            #get the current node's adjacencies
            adjacencies = adjacency_matrix_copy[node]

            #find the smallest edge connecting the current visited node
            #and the
            closest = min(adjacencies)

            #if this edge is smaller than the smallest edge found and the other node it is
            #connected to has not been visited
            if closest < smallest_edge[0] and adjacencies.index(closest) in non_visited_nodes:

                #save it as the smallest edge found
                smallest_edge = (min(adjacencies), node, adjacencies.index(closest))

        #if the smallest edge was not changed, we are done so return
        #the adjacency list of the mst
        if smallest_edge == (float('inf'), None, None):
            return mst_adjacency_list

        #remove the adjacency between the two nodes so that it does not get considered
        #another time
        adjacency_matrix_copy[smallest_edge[1]][smallest_edge[2]] = float('inf')
        adjacency_matrix_copy[smallest_edge[2]][smallest_edge[1]] = float('inf')

        #add the smallest edge to the mst's adjacency matrix
        mst_adjacency_list[smallest_edge[1]].append(chr(65 + smallest_edge[2]))
        mst_adjacency_list[smallest_edge[2]].append(chr(65 + smallest_edge[1]))

        #remove the newly visited node from the list of unvisited nodes
        #and add it to the list of visited nodes
        non_visited_nodes.remove(smallest_edge[2])
        visited_nodes.append(smallest_edge[2])

    return mst_adjacency_list
```

B Tests and simulations

Tests

```python
#general graph
G1 = Graph(5)
G1.add_edge(Node('A'), Node('B'), 2)
G1.add_edge(Node('A'), Node('C'), 1)
G1.add_edge(Node('C'), Node('B'), 3)
G1.add_edge(Node('C'), Node('D'), 1)
G1.add_edge(Node('D'), Node('B'), 4)
G1.add_edge(Node('C'), Node('E'), 6)
G1.add_edge(Node('E'), Node('A'), 5)

assert G1.Kruskal() == [['C', 'B', 'E'], ['A'], ['A', 'D'], ['C'], ['A']]
assert G1.Prim() == [['C', 'B', 'E'], ['A'], ['A', 'D'], ['C'], ['A']]
```

```python
#disconnected graphs
G2 = Graph(4)
G2.add_edge(Node('A'), Node('B'), 2)
G2.add_edge(Node('C'), Node('D'), 1)

assert G2.Kruskal() == [['B'], ['A'], ['D'], ['C']]
assert G2.Prim() == [['B'], ['A'], [], []]

G3 = Graph(7)
G3.add_edge(Node('A'), Node('B'), 2)
G3.add_edge(Node('A'), Node('C'), 1)
G3.add_edge(Node('C'), Node('B'), 3)
G3.add_edge(Node('C'), Node('D'), 1)
G3.add_edge(Node('E'), Node('F'), 1)
G3.add_edge(Node('G'), Node('F'), 10)
assert G3.Kruskal() == [['C', 'B'], ['A'], ['A', 'D'], ['C'], ['F'], ['E', 'G'], ['F']]
assert G3.Prim() == [['C', 'B'], ['A'], ['A', 'D'], ['C'], [], [], []]
```

```python
#graph with no nodes and no edges
G4 = Graph(0)

assert G4.Kruskal() == []
assert G4.Prim() == []
```

```python
#complete graph
import random

G5 = Graph(5)

all_nodes = []
all_edges = []

for i in range(5):
    new_node = Node(chr(65+i))

    for node in all_nodes:
        if i == 4:
            all_edges.append((new_node, node, 1))

        else:
            all_edges.append((new_node, node, 2))

    all_nodes.append(new_node)

G5.add_edge_list(all_edges)

assert G5.Kruskal() == [['E'], ['E'], ['E'], ['E'], ['A', 'B', 'C', 'D']]
assert G5.Prim() == [['E'], ['E'], ['E'], ['E'], ['A', 'B', 'C', 'D']]
```

Simulations

```python
import random
import matplotlib.pyplot as plt
import time

def simulation_complete(trials, method):

    all_edges = []
    all_nodes = []
    number_of_nodes = []
    times = []

    for i in range(trials):

        number_of_nodes.append(i)

        G = Graph(i)
        new_node = Node(chr(64+i))

        for node in all_nodes:
            all_edges.append((new_node, node, random.random()))

        all_nodes.append(new_node)
        G.add_edge_list(all_edges)

        start = time.process_time()

        if method == 'Kruskal':
            mst = G.Kruskal()
        else:
            G.Prim()

        end = time.process_time()

        times.append(end - start)

    return times, number_of_nodes

def create_graph_complete(method, max_size):

    compiled_results = []
    results = []

    for i in range(50):
        results.append(simulation_complete(max_size, method)[0])

    input_size = simulation_complete(max_size, method)[1]

    for i in range(max_size):

        result = 0

        for j in range(50):

            result += results[j][i]

        compiled_results.append(result/50)

    return input_size, compiled_results

prim_complete = create_graph_complete('Prim', 100)
kruskal_complete = create_graph_complete('Kruskal', 100)
```

```python
import random
import matplotlib.pyplot as plt
import time

def simulation_constant_nodes(trials, method):

    all_edges = []
    number_of_edges = []
    times = []

    G = Graph(20)

    nodes = [Node(chr(65+i)) for i in range(20)]

    for i in range(trials):

        number_of_edges.append(i)
        sample = random.sample(nodes, 2)
        all_edges.append((sample[0], sample[1], random.random()))

        G.add_edge_list(all_edges)

        start = time.process_time()

        if method == 'Kruskal':
            mst = G.Kruskal()

        else:
            mst = G.Prim()

        end = time.process_time()

        times.append(end-start)

    return times, number_of_edges

def create_graph_nodes(method, max_size):

    compiled_results = []
    results = []

    for i in range(10):
        print(i)
        results.append(simulation_constant_nodes(max_size, method)[0])

    input_size = simulation_constant_nodes(max_size, method)[1]

    for i in range(max_size):
        result = 0

        for j in range(10):
            result += results[j][i]

        compiled_results.append(result/10)

    return input_size, compiled_results

prim_nodes = create_graph_nodes('Prim', 100)
kruskal_nodes = create_graph_nodes('Kruskal', 100)
```

```python
import random
import matplotlib.pyplot as plt
import time

def simulation_constant_edges(trials, method):

    all_edges = []
    all_nodes = []
    number_of_nodes = []
    times = []

    edges = 20
    for i in range(10):
        all_nodes.append(Node(chr(64+i)))

    for i in range(10, trials+1):

        G = Graph(i)

        all_nodes.append(Node(chr(64+i)))
        number_of_nodes.append(i)

        for i in range(20):
            sample = random.sample(all_nodes, 2)
            all_edges.append((sample[0], sample[1], random.random()))

        G.add_edge_list(all_edges)

        start = time.process_time()

        if method == 'Kruskal':
            mst = G.Kruskal()
        else:
            mst = G.Prim()

        end = time.process_time()

        times.append(end - start)

    return times, number_of_nodes

def create_graph_edges(method, max_size):

    compiled_results = []
    results = []

    for i in range(50):
        results.append(simulation_constant_edges(max_size, method)[0])

    input_size = simulation_constant_edges(max_size, method)[1]

    for i in range(max_size-9):

        result = 0

        for j in range(50):

            result += results[j][i]

        compiled_results.append(result/50)

    return input_size, compiled_results

prim_edges = create_graph_edges('Prim', 100)
kruskal_edges = create_graph_edges('Kruskal', 100)
```

C LO and HC tags

#ComplexityAnalysis

I carried out very thorough analyses of the time complexities of both Kruskal's and Prim's algorithms. Indeed, I first started by deriving them theoretically by analyzing the flow charts and the codes, for both of them. I then created three different simulations that would allow me to test these theoretical results. Lastly, I also made sure to discuss space complexities.

#ComputationalCritique

I applied this LO at three separate occasions in this assignment. The first two times where when I critically assessed my implementations of each algorithm separately by comparing them to the textbook, and by pointing out other flaws or issues. Additionally, I provided explanations as to why there are disparities in time complexities and provided a potential improvement for Kruskal's algorithm. Moreover, the third time I applied this LO was in the "Kruskal's or Prim's?" section where I compared both algorithms within the context of my implementation using graphs and within a more general context.

#AlgorithmicStrategies

Before discussing algorithmic strategies, I provided a background section to introduce the relevant aspects of networks and MSTs. This therefore provided beginner readers enough knowledge to be able to easily follow the strategies of both algorithms.

I then went through how these algorithms work in a simple and straightforward matter, without the use of any jargon (except graph theory jargon which was previouly introduced) and with the aid of descriptive flowcharts.

#DataStructures

The main data structure that was used in this assignment was networks which I provided an in depth explanation of in the background section. However, when I had the choice between two data structures, I made sure to explain why both options would work and then justified why I picked a specific one.

#CodeReadability

Throughout my code, I used very clear variable names, as well as including docstrings for any new Class or method created. I also heavily, yet appropriately, commented my code. Lastly, I also chose to use Object Oriented Programming to further maximize code readability.

#PythonProgramming

While my implementations were not necessarily the best, I always made sure to explain the flaw within them and provided a potential improvement for Kruskal's. Moreover, I also carried out multiple tests that explored various edge cases and explained why these specific ones were important to verify. Lastly, I made sure to briefly discuss the benefits of using Object Oriented Programming.

#scienceoflearning

Throughout this assignment I applied multiple principles of Science of Learning to extend and deepen my knowledge. Firstly, I decided to work on this topic because I had already studied graph theory and MSTs extensively, but only within a Mathematics context. Therefore, by exploring it within the field of computer science, not only did I use the building on priors and foundational learning principle to increase my knowledge, but I also used spaced practice to refresh my prior knowledge. This is an integral example of a learning phenomenon known as Quinian bootstrapping. It is the idea that you categorize something (like MSTs) and then start to build a network of understanding around it. The more you transfer it to different disciplines, the better you become better at understanding it. Moreover, I structured my assignment by following the associative chaining principle so that the information would be stored in my brain like a story, making it much easier to remember.

#plausibility

This HC was very useful when looking at time complexity and critiquing my algorithm. Indeed, I had a hypothesis about the time complexities of these algorithms based on the textbook. However, when noticing that my time complexities were not plausible based on that hypothesis, I was able to take a deeper dive into my algorithms and, therefore, become able to point out the flaws and issues within them.

References

[Cor+09] Thomas Cormen et al. *Introduction to algorithms*. Mit Press, 2009.