Qualitatively understanding how mst/shortest path algorithms can be used in urban mass rapid transit

Krushna Goel: 2022MCB1269

Nimish Sehgal: 2022MCB1273

Shardul Kiran Deo: 2022CSB1122

Aniruddh Muley: 2022MCB1257

**ABSTRACT:**

This report acts as a brief look on how graph theory can be used to model urban mass rapid transit (MRT or metro in short) systems. There are brief descriptions on metro networks, graph algorithms as well as a small digression into heuristic and approximation algorithms. Then we take a very simplistic MRT example, where we randomly generate a K15 graph with each edge weight representing the construction costs of connecting the two stations and apply MST algorithms on this. The output is then analysed and compared with real-life metro networks, while there is also discussion on the different variables that can affect metro construction in general and could have been incorporated into the graph model. All in all, this is an intersection of civil engineering and mathematical modelling, with the modelling done using discrete mathematics (among other topics such as optimization), graph theory in particular.

**MASS RAPID TRANSIT SYSTEM (MRT):**

A mass rapid transit system (MRT or metro in short) is a mode of public transport found in urban areas, such as cities and urban agglomerations. It connects important areas in the city such as business centres, markets, train stations, airports, landmarks and monuments, important localities, etc. In many cities, metro networks are long enough to span the entire city. They transport people in high numbers on a daily basis without the problems of traffic congestion or delays, while having a smaller carbon footprint than many other modes of transport. Constructing a metro is no mean feat, and requires a combination of different approaches and considerations. It involves analysing factors such as population density, demand patterns, existing transportation infrastructure, land availability, and budgetary constraints. Comprehensive planning and modelling techniques, including network analysis, optimization algorithms, and simulation, are typically employed to design an effective and efficient rail transit system. One way of performing network analysis and optimization of cost and travel time is by modelling a metro network using graph theory. We can model it using a weighted graph, with the weight incorporating factors such as construction cost, travel time, etc.

**METRO NETWORK OPTIMIZATION USING GRAPH THEORY:**

If we are to model a metro rail system using graph theory, then there always arises the problem of optimization, be it in construction costs or in travelling time. Minimum Spanning Tree (MST), Shortest Path, and Travelling Salesman Problem (TSP) are three well-known problems in graph theory and optimization. Although they share some similarities, they are distinct problems with different objectives and approaches to a solution. Here we explore each problem individually, in the context of mass rapid transit:

* **Minimum Spanning Tree (MST):** The MST problem aims to find a tree that connects all the vertices of a weighted undirected graph while minimizing the total weight of the tree. The resulting tree should have no cycles and include all the vertices. MSTs have various applications, such as designing network infrastructure, constructing efficient electrical networks, and organizing hierarchical clustering. If each edge has a distinct weight, then there will be only one and a unique MST. Algorithms for finding the MST have existed before the advent of modern computers (Borůvka’s algorithm). Two well-known MST algorithms are the Prim’s and Kruskal’s algorithms. It is a suitable approach if the primary objective is to establish a connected rail network that reaches all the desired locations while minimizing the overall cost and/or distance, and using the minimum number of tracks and connections required.
* **Shortest Path:** The Shortest Path problem focuses on finding the most efficient path between two vertices in a graph, i.e., with the minimum total weight or cost. Shortest path algorithms, like Dijkstra's algorithm and the Bellman-Ford algorithm, are commonly used to solve this problem. Applications of the shortest path problem include route planning, navigation systems, and network routing. In the context of urban rail transit development, Dijkstra's algorithm can be employed to optimize route planning, particularly for identifying the most efficient paths between popular destinations, such as major employment centres, residential areas, and commercial hubs. This approach can help minimize travel time and congestion on the rail network, by choosing the right stations for the metro network from a list of options.
* **Travelling Salesman Problem (TSP):** The TSP is an optimization problem that seeks to find the shortest possible route that visits all given cities and returns to the starting point. The objective is to minimize the total distance or cost of the tour. It is a classic NP-hard problem, meaning that there is no known efficient algorithm that can solve it for large instances in polynomial time. Various approximate algorithms and heuristics, such as the 2-opt and 3-opt algorithms, are employed to find near-optimal solutions. This approach, however, is not directly applicable to urban rail transit development. However, some elements of TSP, such as heuristics and approximate algorithms used to solve TSP, can be applied to optimize specific aspects of urban transit in general, for example in bus timetables where the time taken in a round trip covering all stops is the variable being optimized. We shall describe heuristic and approximation algorithms in a bit more detail in the next section.

While the MST, TSP, and shortest path problems have their own applications, none of them individually provide a complete solution for urban rail transit development. It is necessary to combine elements from different approaches and employ comprehensive planning techniques to create an effective and efficient rail transit system.

**HEURISTIC AND APPROXIMATION ALGORITHMS:**

In computer science, the P set of problems (which can be solved in polynomial time complexity) are much easier to solve than the NP set of problems (which have non-polynomial time complexity and are very computationally expensive). The main aim of heuristic algorithms is to increase the speed of solving the problem, i.e., to solve NP problems in polynomial time, and are used when classical methods fail to solve NP problems efficiently. They may involve trial-and-error methods. For this time reduction, they use accuracy as a trade-off, as more often than not, the “good enough” solution we get from such algorithms is quite a reasonable approximation for the “perfect” algorithm. Approximation algorithms too are used where finding the optimal solution is infeasible. An approximation algorithm is guaranteed to be reasonably close to the optimal solution, while heuristic algorithms may or may not lead to an optimal solution. Applications in public transport in general include bus route planning, vehicle rescheduling, choosing metro stations in a list and departure control in metro rail stations.

**MST ALGORITHMS (AND GREEDY ALGORITHMS):**

We shall look at two MST algorithms in particular:

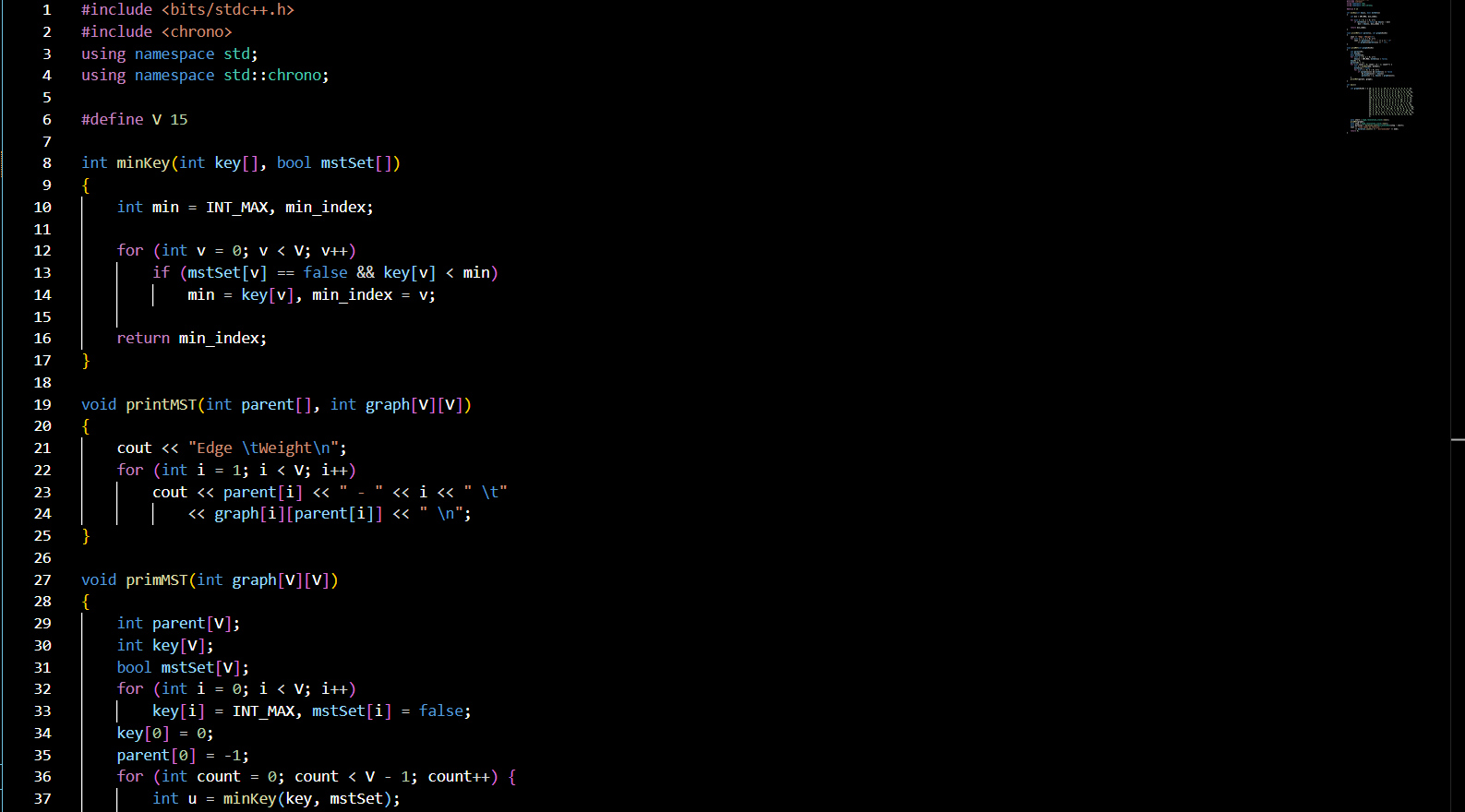
* **Kruskal’s algorithm:** This algorithm finds the minimum spanning tree of a graph by first sorting all the edges by their weight in ascending order. Then, it iteratively adds the edges to the minimum spanning tree, starting with the edge with the smallest weight, as long as adding the edge doesn’t create a cycle. To keep track of whether adding an edge creates a cycle, the algorithm uses a disjoint-set data structure. This algorithm needs more time because it needs to sort the edges, so its time complexity is O(E log E), where E is the number of edges in the graph. It also needs more space to store the edges and the disjoint-set data structure, so its space complexity is O(E + V), where V is the number of vertices in the graph.
* **Prim’s algorithm:** This algorithm finds the minimum spanning tree of a graph by starting with an arbitrary vertex and iteratively adding the edge with the smallest weight that connects a vertex in the tree to a vertex not in the tree. To keep track of which edges to add, the algorithm uses a priority queue. This algorithm needs more time because it needs to keep track of which edges to add using a priority queue, so its time complexity depends on the implementation of the priority queue. With a binary heap, the time complexity is O((E + V) log V), where E is the number of edges and V is the number of vertices in the graph. It also needs more space to store the edges and the priority queue, so its space complexity is O(E + V).

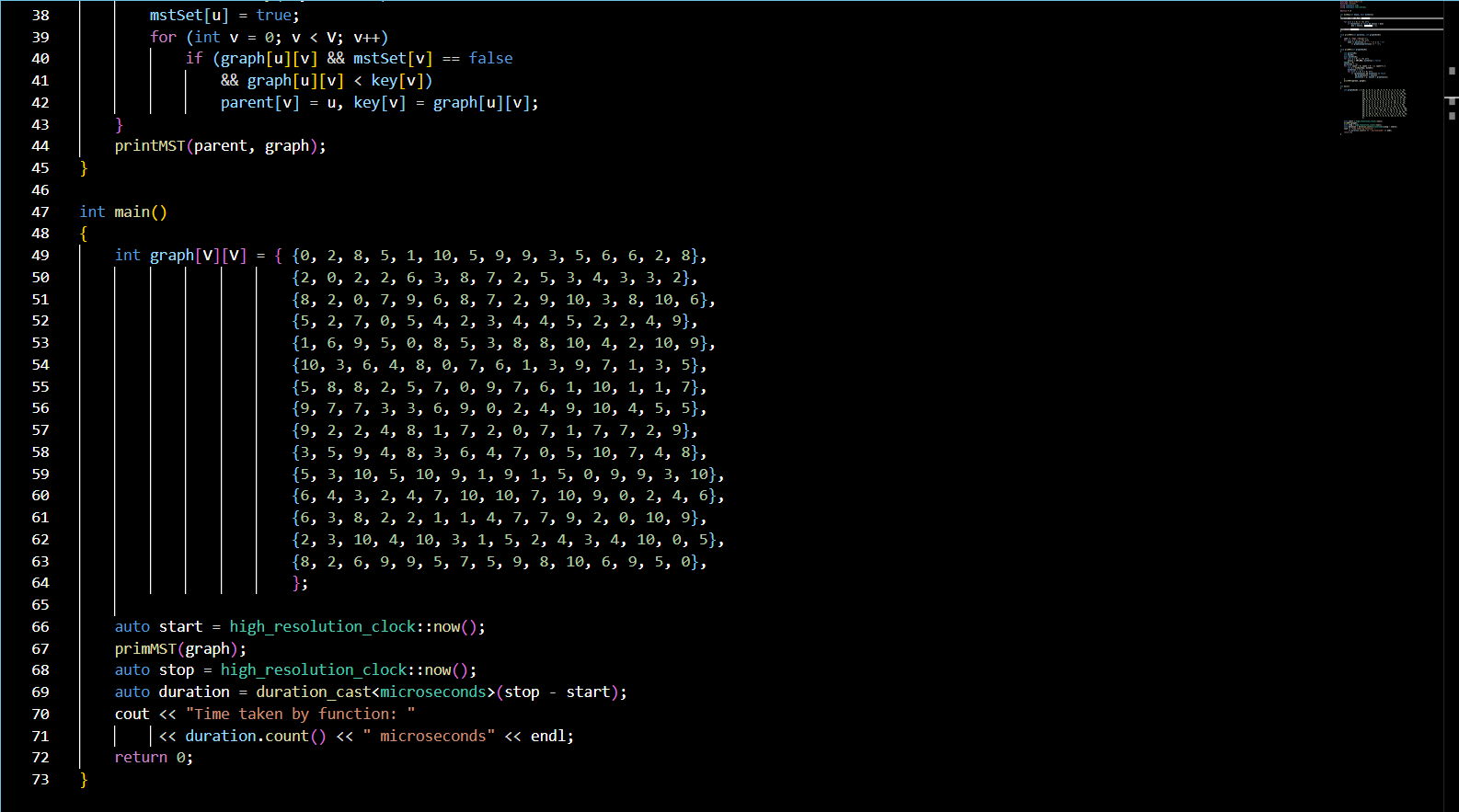
Both of these algorithms are greedy algorithms. Greedy algorithms choose the next step in a manner that seems optimal within a local scope. This heuristic approach of choosing the next most optimum step leads to a locally optimum solution, but not necessarily the global optimum. It can be computationally efficient and effective in situations where a good heuristic estimation is available. It can quickly find trees that are often reasonably good and close to optimal, especially in scenarios where the obstacles are graph is sparse or not too complex. In general, using greedy algorithms depends on how much of a trade-off is acceptable between speed and accuracy.

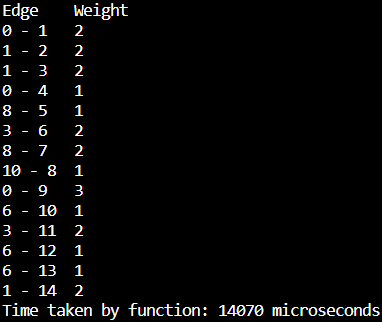
**MODELLING A METRO NETWORK USING A WEIGHTED GRAPH – A TOY EXAMPLE:**

For the purpose of illustration, let us imagine, a growing city. We wish to connect the main destinations of the city, such as the main software hub, main business district, main market, main tourist attraction, most populous locality/suburb, main railway station, the airport, among other important landmarks. We are imagining the metro layout to have 15 stations, with the metro lines being overground. This is modelled using a complete graph K15. The edge weights in this model represent (purely) the costs of constructing the lines between the two stations connected by an edge (we shall see how this affects the outcome, and what other variables come into play). The edge weights are non-negative integers less than or equal to 10 units. We wish to obtain the most cost-effective metro network using the given inputs; in other words, we wish to find the MST of this graph.

The weights of the graph for this example are generated using the rand() function, a pseudorandom number generator. We are using both Prim’s and Kruskal’s algorithms here to generate the MST. We compare their runtimes and their outputs in the subsequent section. We then analyse the outputs and compare them to real-life scenarios and variables which haven’t been accounted in this simplistic model.

C++ Code for Prim’s algorithm (source: GeeksforGeeks):

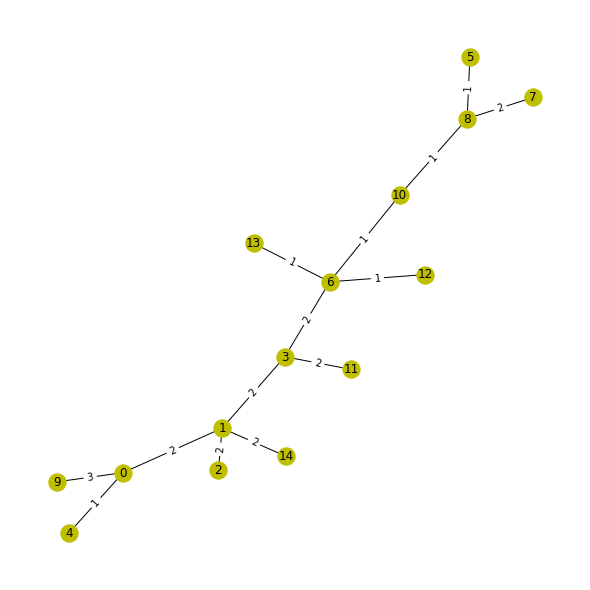




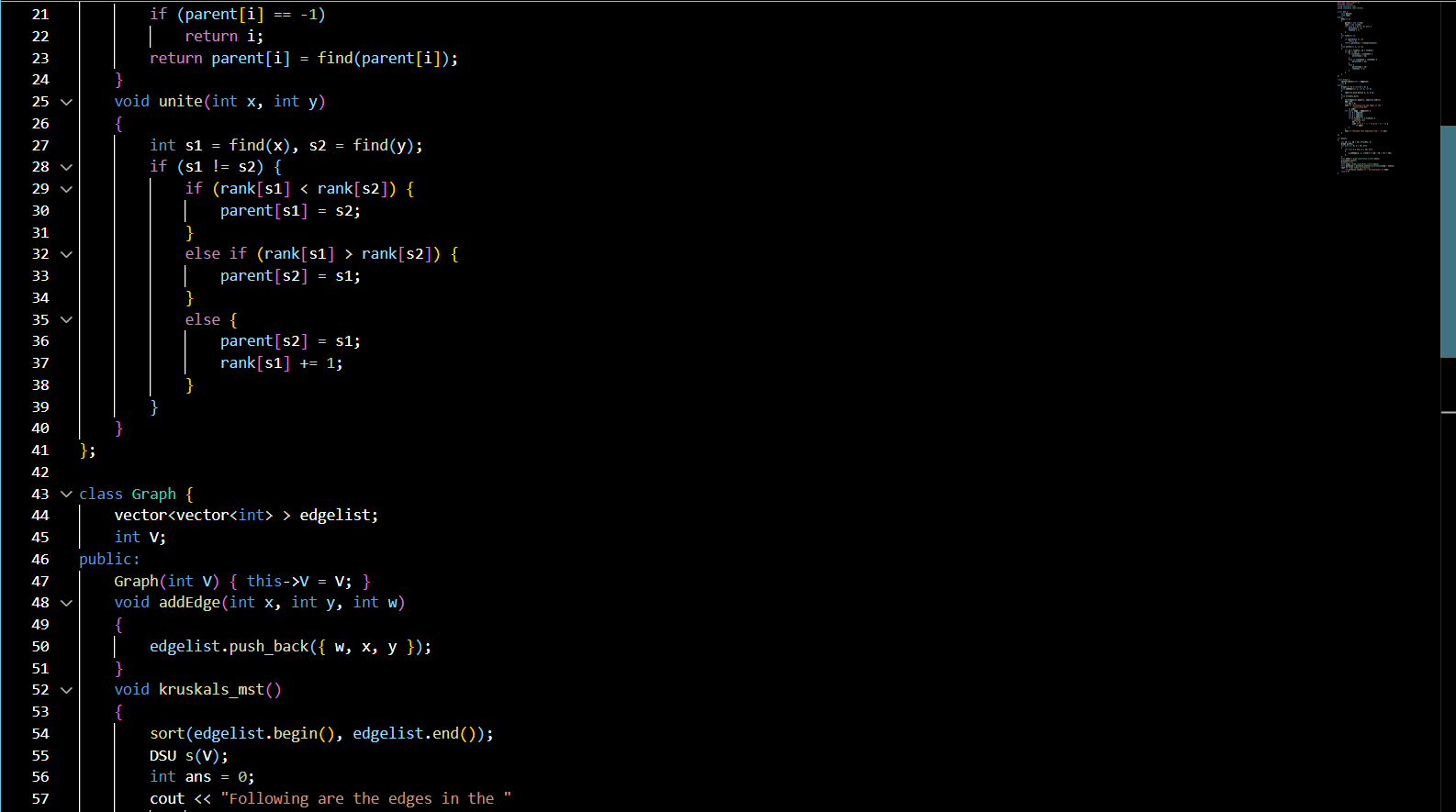
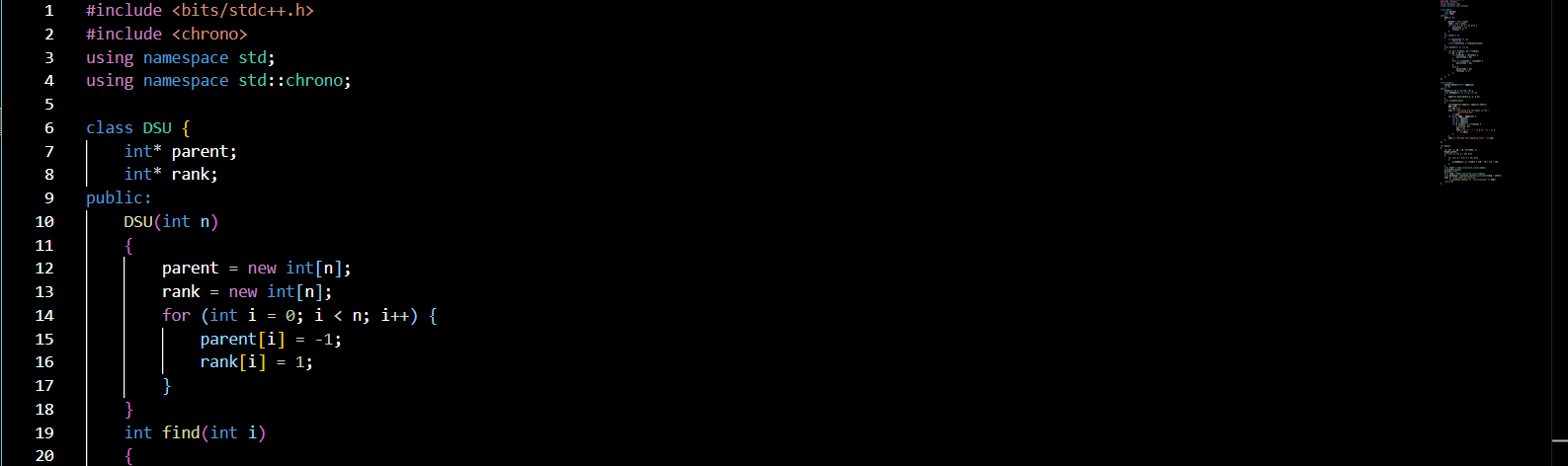
Output of the Prim’s algorithm code

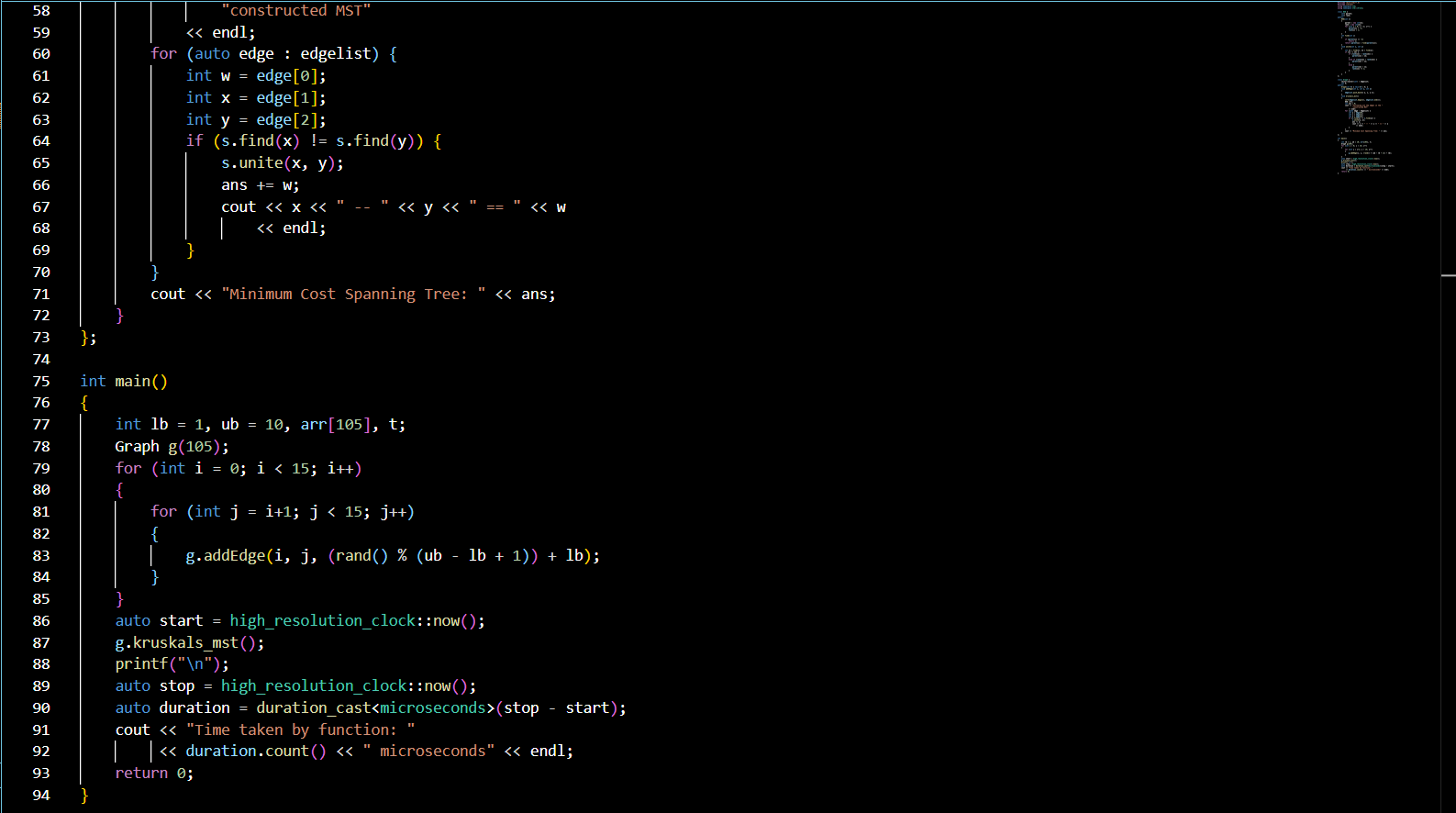
Python code for pictorially representing Prim’s algorithm

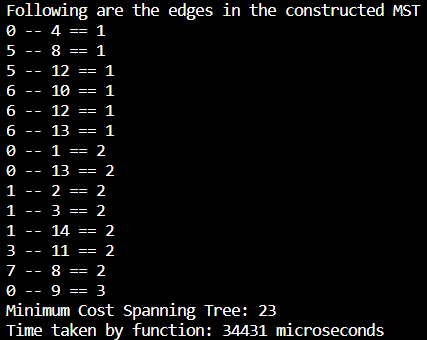




Prim’s algorithm: a pictorial representation

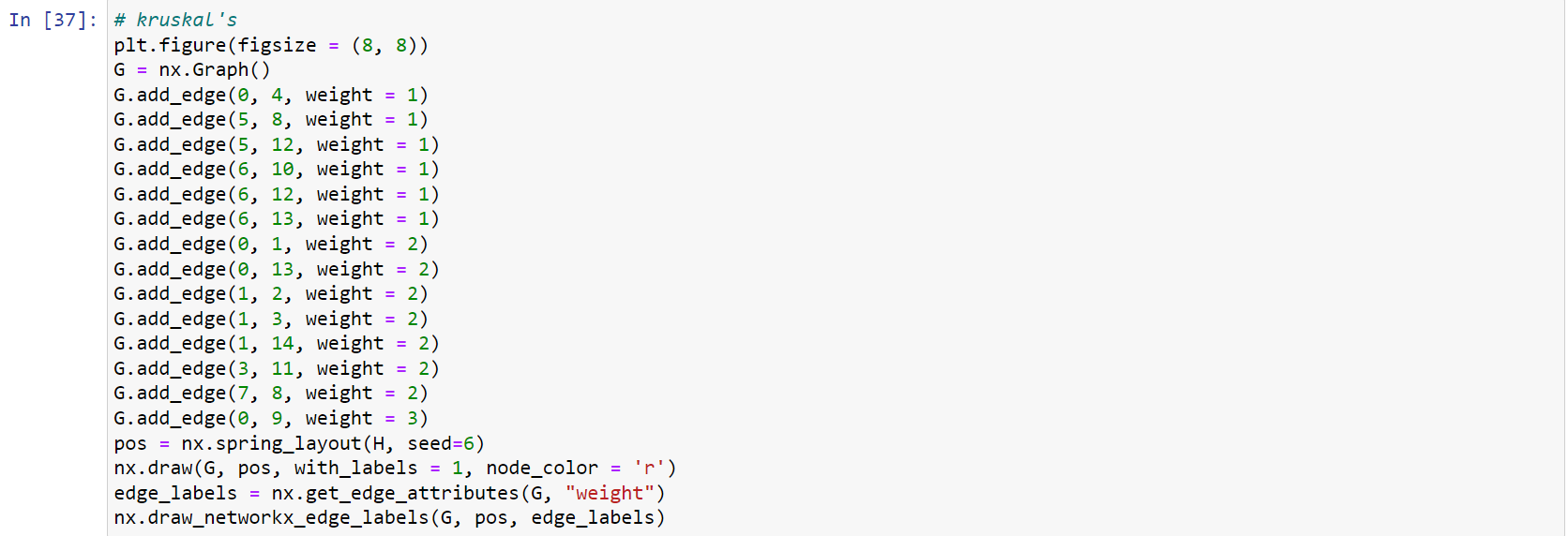
C++ Code for Kruskal’s algorithm (source: GeeksforGeeks):

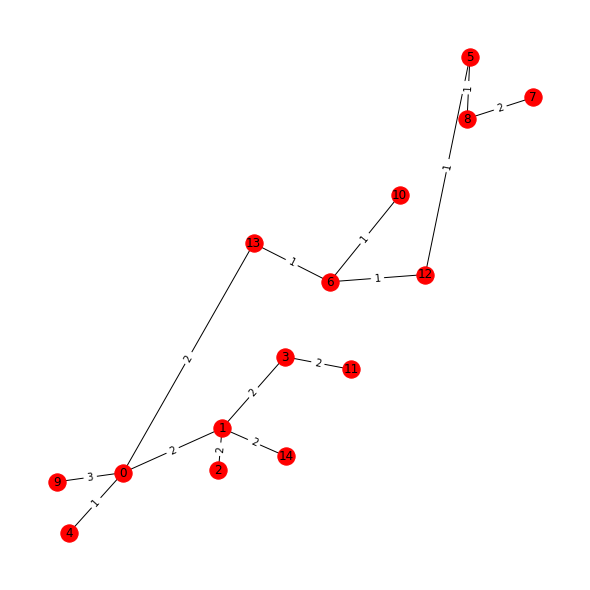




Output of the Kruskal’s algorithm code

Python code for pictorially representing Kruskal’s algorithm



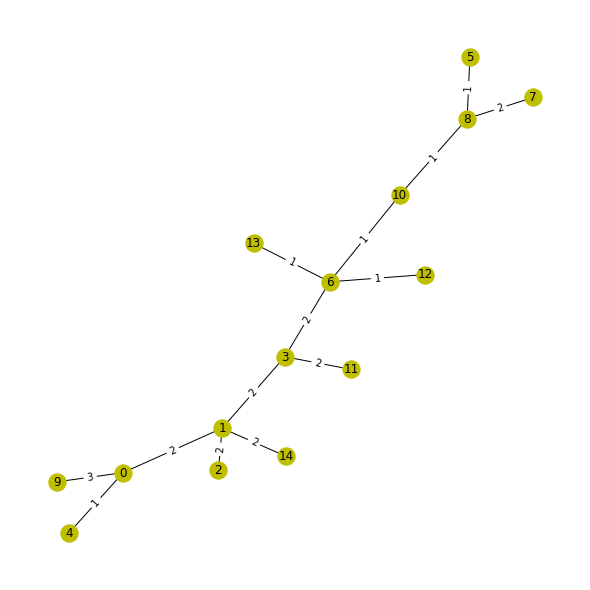
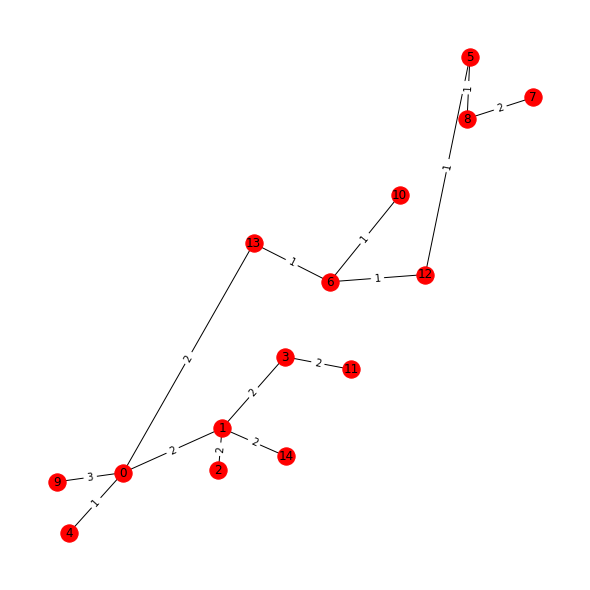


Kruskal’s algorithm: a pictorial representation

**ANALYSIS AND DISCUSSION:**

The two algorithms generate two minimum spanning trees with the cost of the tree being 23 units. However, the two graphs generated are distinct, indicating that different MST algorithms may converge at the same minimum cost, but not always at the same MST, as multiple such MSTs may have the same cost. The time taken to run Kruskal’s algorithm was 34431 microseconds, while the Prim’s algorithm implementation took 14070 microseconds. This particular implementation of Kruskal’s algorithm uses a disjoint-set data structure and creates a graph class, while the Prim’s algorithm code operates directly on an adjacency matrix. The runtime isn’t a major issue here, instead it’s the graphs generated which deserve a more detailed discussion.

The red graph shown below is how the Kruskal’s algorithm output MST looks like, while the yellow one is for the Prim’s algorithm output.

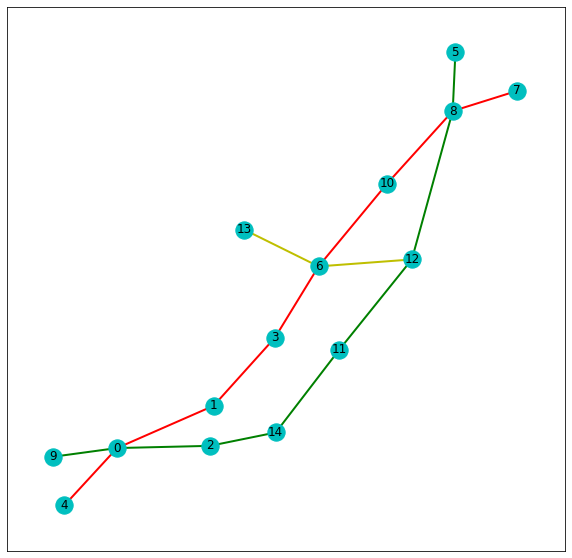
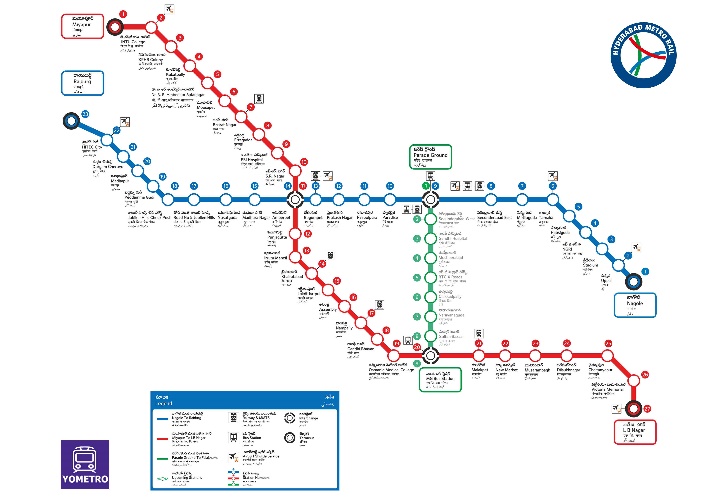


Output of Prim’s algorithm

Output of Kruskal’s algorithm

A “line” in a metro network refers to a particular route connecting two terminal stations, and having many intermediate stations. All metro networks have a finite number of lines and their combination and intersection points form an extensive metro network. Now this is a randomly generated graph for 15 nodes, and while these indeed are the MSTs, are these outputs (in our example metro network) actually practical?

There is a main line, yes, but then all these smaller branches, practically, will require a separate set of rolling stock to cover just the distance between two stations. The Hyderabad metro has 3 long lines in a secant topology, while the MST we generate above will have atleast 6-7 lines! The operational cost for this will far outweigh the establishment cost. It may be cheaper to construct the metro lines like this, but the cost in power consumption, rolling stock, logistics such as independent signal systems for each line, not to mention the cost of building multiple interchange stations (i.e. where different lines intersect, hence they have more than one level compared to normal stations and are more expensive to build; we have 5 of them in the yellow graph, namely: 0, 1, 3, 6, and 8) will make the cost balloon. A more realistic looking MRT system on this set of stations would be like this (discounting the weights here, as the weights are randomly generated):

Map of Hyderabad metro network

A more realistic looking metro network

Notice how station number 13 is once again forcing the creation of a new metro line? The location of stations could have been chosen in a more appropriate manner (only in this particular case, as the locations of the points on this graph were generated using a seed in the code).

The modelling should have included more factors beyond simply the cost of constructing a metro line based on, say, geographic location. It is easier to construct overground metro lines along a road, rather than across a major lake, river or a small cluster of hills. Also, some places may require construction of an underground metro system, which further increases costs.

Other factors such as passenger traffic and travel patterns should either be considered beforehand (as restrictions on edge locations, to further narrow down the possible graphs and get a better output) or should be incorporated into the edge weights of the model and accordingly influence the algorithm’s convergence to an MST. Certain arteries in the road system, and certain areas (IT hubs/tourist areas/airport/downtown/sports stadium) see regular and predictable flow of passenger traffic, and the road network congestion zones in such areas can be eased using MRT (we have, in fact, assumed that the metro in our example city connects such major activity centres/landmarks). We also need to consider the location of depots for housing and maintenance of the metro rolling stock.

It is also possible that the planned metro line or station may be near, say, a religious site (such as a temple) or an area of conservation (the Aarey colony situation in Mumbai immediately comes to mind), there is possibility of conflict between the metro authority and those being affected, such as temple-goers and conservationists. Such soft factors may force the metro authority to take longer routes or shift the metro station itself, which may increase costs, and have to be accounted for beforehand.

Let us imagine that we have converged at a satisfactory MST (as shown in the graph above). Now we need to construct the station at the chosen location and along the chosen route. Doing so has its own factors to take into account and its own costs, which cannot be incorporated into the edge weights of our graph model. Some examples of factors (posed in the form of questions) which can affect the cost are:

* Is it underground or overground?
* Does it serve a single line or is it an interchange station?
* If it is an interchange station, how do the passengers transfer from one line to another (or to another mode of transport such as bus or train)?
* Is the station accessible via roads, foot or some other mode of transport?
* Is it easy to access for all people, including the disabled/elderly?
* If it is near, say, a football stadium where regular matches are held, can it handle the passenger traffic?

This lengthy discussion on metro rails may seem to be more related to civil engineering and mathematical modelling. However, the application of graph theory is a form of mathematical modelling, and was used to qualitatively (and in a rather non-numerical way) illustrate how it can be used to generate a network for a metro system. Also, the subsequent analysis of the outcome and comparison with real-life metros, as well as discussion of the various real-life variables that could influence our model or affect metro construction in general, was to give a more well-rounded view of the topic of modelling an MRT system in general.

**SOURCES:**

<https://link.springer.com/chapter/10.1007/978-3-642-01129-0_91>

<https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/>

<https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/>

<https://www.geeksforgeeks.org/rand-and-srand-in-ccpp/>

<https://www.geeksforgeeks.org/measure-execution-time-function-cpp/>

<https://en.wikipedia.org/wiki/Rapid_transit>

<https://networkx.org/documentation/stable/auto_examples/drawing/plot_weighted_graph.html>

<https://www.ltmetro.com/metro-network-map/>

<https://en.wikipedia.org/wiki/Mathematical_model>

<https://warwick.ac.uk/fac/soc/wbs/subjects/orms/research/areas/approximation_and_heuristics>

<https://en.wikipedia.org/wiki/Minimum_spanning_tree>

<https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction>

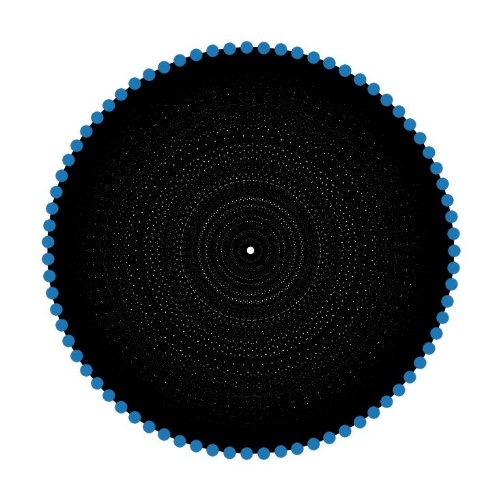
<https://en.wikipedia.org/wiki/Shortest_path_problem>

<https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm>

<https://en.wikipedia.org/wiki/Travelling_salesman_problem>

<https://www.geeksforgeeks.org/travelling-salesman-problem-using-dynamic-programming/>

<https://en.wikipedia.org/wiki/Greedy_algorithm>



A K75 graph using NetworkX, showing the beauty of graph theory