CSE 5311 – Project

MST and TSP for Metric Graphs with MST Heuristic

By

Sabarish Raghu

Changanti Sambasiva Tej

Koushik Modayur Chandramouleeswaran

Table of Contents

Introduction:-

The goal of this project is to implement the Kruskal’s MST algorithm, Prim’s MST algorithm using various data structures. The various MST algorithm implementations are compared with each other based on the size of the input and performance (runtime). The most efficient MST algorithm is chosen from the variations implemented in this project and it is used to solve the Travelling Salesman problem with 2-Approximation. The results of this approximation algorithm is compared with the optimal solution.

System Design:-

Modules:-

Module Name	**Description**
Node.java	This class defines the attributes and methods associated with Nodes.
NaiveKruskal.java	This class contains the Naïve Kruskal’s algorithm implementation.
KruskalUnion.java	This class contains the Kruskal’s algorithm using Union Find data structure implementation.
KruskalUnionPath.java	This class contains the Kruskal’s algorithm using Union Find data structure with path compression implementation.
PrimsUnsorted.java	This class contains the implementation of the Prim’s algorithm using unsorted array.
PrimsSorted.java	This class contains the implementation of the Prim’s algorithm using sorted array.
PrimsHeap.java	This class contains the implementation of the Prim’s using Heap.

TspApprox.java	This class contains the 2-Approx algorithm for travelling sales man using Prim's.
SortedEdges.java	This class defines the attributes and methods for Candidate Edges.

Implementation:-

Kruskal's Algorithm:-

In this project three variations of Kruskal's algorithm have been implemented. They are listed below.

Naïve Kruskal's Algorithm: - In the Naïve Kruskal's algorithm the nodes are created based on the input file. Then the Euclidean distance between the nodes are computed and stored in an adjacency matrix. The edges with the least distance are traversed and in order and the nodes are marked as visited, if some edges form a cycle then they are skipped, this traversal is done until all the nodes have been visited. The existence of a cycle is checked using Depth First Search and it takes O (E) time. Total time complexity is O(E.V).

Methods:

- **findMSTKruskal :-** This method sorts the edges, calculates the distance between the nodes and adds it to the adjacency matrix.
- **checkDfsCycle :-** This method is used to check if the set of edges that are visited form a cycle or not using depth-first search traversal.

Data structures Used:-

- **Tree Set: -** Tree set is used to store the edges, these edges are sorted using a comparator.
- **Array List: -** Array list is used to store the distances/weights between the various nodes in the form of adjacency matrix.

Kruskal's Union: - In Kruskal's union the nodes are created based on the input file and a hash set containing the nodes as individual sets are created. Then the Euclidean distance between the nodes are computed and stored in an adjacency matrix. The edges with the least distance are traversed and in order and the nodes are marked as visited and the union is done if it does not form a cycle. The existence of a cycle is done by checking if the two nodes are in the same set. If they are not in the same set then the union is performed. The time taken to detect a cycle is O (n). Total time complexity is O(E.Log(V)).

Methods:

- **findMSTKruskalUnion: -** This method sorts the edges, calculates the distance between the nodes and adds it to the adjacency matrix. The edges with the least distance are traversed and in order and the nodes are marked as visited. Separate sets are created for each node by calling the makeset method and cycle checks are made by calling the find method. Finally the union operation is performed.
- **makeset:-** This method is used to create individual sets for each node.
- **find:-** This method returns the name of the set associated with a node.
- **union :-** union method performs the union of two sets by rank .

Data Structures Used: -

- **Tree Set: -** Tree set is used to store the edges, these edges are sorted using a comparator.
- **Array List: -** Array list is used to store the distances/weights between the various nodes in the form of adjacency matrix.
- **Hash Table: -** Hash table is used to create individual sets for the nodes that are present in the graph.

Kruskal's Union with path compression: - In the Kruskal's union with path compression the nodes are created based on the input file and a hash set containing the nodes as individual sets are created. Then the Euclidean distance between the nodes are computed and stored in an adjacency matrix. The edges with the least distance are traversed and in order and the nodes are marked as visited and the union is done if it does not form a cycle. When a union is performed path compression is done by setting the parent of the new node as the name of the set. The time taken to detect a cycle is O(1). Total time complexity is O(E.Log(V)).

Methods:-

- **findMSTKruskalUnion: -** This method sorts the edges, calculates the distance between the nodes and adds it to the adjacency matrix. The edges with the least distance are traversed and in order and the nodes are marked as visited. Separate sets are created for each node by calling the makeset method and cycle checks are made by calling the find method. Finally the union operation is performed.
- **makeset:-** This method is used to create individual sets for each node. Takes O(1) time.
- **find:-** This method returns the name of the set associated with a node. If the parent node of a node is not the name of the current set then this function is recursively called until it finds the node which is also the name of the set and updates that as the name of the set for which find was called and all the nodes that are between the new node and the node that is the set name. By doing this we perform path compression and this reduces the time complexity of find operation to O (1).
- **union :-** union method performs the union of two sets by rank.

Data Structures Used: -

- **Tree Set: -** Tree set is used to store the edges, these edges are sorted using a comparator.
- **Array List: -** Array list is used to store the distances/weights between the various nodes in the form of adjacency matrix.
- **Hash Table: -** Hash table is used to create individual sets for the nodes that are present in the graph.

Prim's Algorithm: - In this project three variations of Prim's algorithm have been implemented. They are listed below.

Prim's Algorithm with Unsorted Array: - In the Prim's algorithm with unsorted array the nodes are created based on the input file, the first node is taken as source node and the adjacent edges are considered as the candidate edges and stored in a Hash Set. The edge having the minimum edge is found by iterating through all the edges in the candidate edge set. Find the node to which this edge reaches and recursively find the minimum edges until all the nodes are visited. Check that there is no cycle by checking if both the nodes on the chosen edge are not available in the covered nodes list. Checking the cycle takes. Total run time is O(E.V).

Methods:-

- **findMSTPrim :-** In this method the first node of the graph is set as source and candidate edges for it are found by calling the method getRecursiveMST.
- **getRecursiveMST:-** In this method we find the adjacent edges to the node and add them to the candidate set which is a tree set. Find the first candidate edge. Then get id of the node to which the candidate edge reaches and recursively find the adjacent edges of the nodes and add them to the candidate edge set and traverse to the first edge in the again.

Performing this process recursively gives us the Minimum spanning tree when all the nodes have been visited. The edges that have been traversed are removed from the candidate edges set.

Data Structure Used:-

- **Hash Set: -** The nodes that are covered through the traversal are maintained in a Hash Set.
- **Array List: -** The graph is stored as an array list of nodes. The candidate edges are also stored as array list.

Prim's Algorithm with Sorted Array: - In the Prim's algorithm with sorted array the nodes are created based on the input file, the first node is taken as source node and the adjacent edges are considered as the candidate edges and stored in a Tree Set. The edge with the minimum weight is traverse and the node reached through that edge is marked as visited. The outgoing edges from that is added to the candidate edges and the process of finding the minimum edge is done again , this process continues until all the nodes are visited. Check that there is no cycle by checking if both the nodes on the chosen edge are not available in the covered nodes list. Total runtime is O(E.log(V))

Methods:-

- **getMSTSorted:-** In this method the first node of the graph is set as source and candidate edges for it are found by calling the method getRecursiveMST.

- **getRecursiveMST:-** In this method we find the adjacent edges to the node and add them to the candidate set which is a tree set. Find the first candidate edge. Then get id of the node to which the candidate edge reaches and recursively find the adjacent edges of the nodes and add them to the candidate edge set and traverse to the first edge in the again. Performing this process recursively gives us the Minimum spanning tree when all the nodes have been visited. The edges that have been traversed are removed from the candidate edges set.

Data Structures Used: -

- **Hash Set: -** The nodes that are covered through the traversal are maintained in a Hash Set.
- **Array List: -** The graph is stored as an array list of nodes.
- **Tree Set: -** The candidate edges are stored in a Tree Set. The Tree Set stores the edges in sorted order.

Prim's Algorithm with Heap: - In this algorithm we use the heap data structure to store the candidate edges and using heap enables faster retrieval of the edge with the minimum weight. The insertion and deletion operations take the majority of the time, the find min from the queue takes only constant time. Check that there is no cycle by checking if both the nodes on the chosen edge are not available in the covered nodes list. Total runtime is O(E.(log(V))

Methods:-

- **getMSTSorted :-** In this method the first node of the graph is set as source and candidate edges for it are found by calling the method getRecursiveMST.

- **getRecursiveMST:-** In this method we find the adjacent edges to the node and add them to the candidate set which is a heap. Find the first candidate edge. Then get id of the node to which the candidate edge reaches and recursively find the adjacent edges of the nodes and add them to the candidate edge set and traverse to the first edge in the again. Performing this process recursively gives us the Minimum spanning tree when all the nodes have been visited. The edges that have been traversed are removed from the candidate edges set. Since we use a heap this method takes less time when compared to using unsorted or sorted array.

Data Structures Used:-

- **Array List: -** The graph is stored as an array list of nodes.
- **Binary Heap: -** Binary heap is used to store the candidate edges.

TSP 2-Approximation Algorithm: - The approximation algorithm that is proposed here is a 2-approzimation algorithm. The minimum spanning tree of the graph is first found using Prim's algorithm with heaps. A pre-order traversal of the minimum spanning tree is performed and the last node is connected back to the staring node, this gives us the solution to the TSP with an upper bound of 2 * Optimal solutions. Total time complexity is O(E.log(V)).

Methods:-

- findTSPApprox **: -** This method calls the getMSTSorted function which uses Prim's algorithm implemented using heap to find the minimum spanning tree. The sorted edges are stored in an array list, visit preorder method is called with the first node in the MST as the starting point.

- visitpreorder :- This method returns the list of nodes in the pre-order traversal of the minimum spanning tree . This gives us the Hamiltonian cycle of the given MST and this is the solution to the TSP problem.

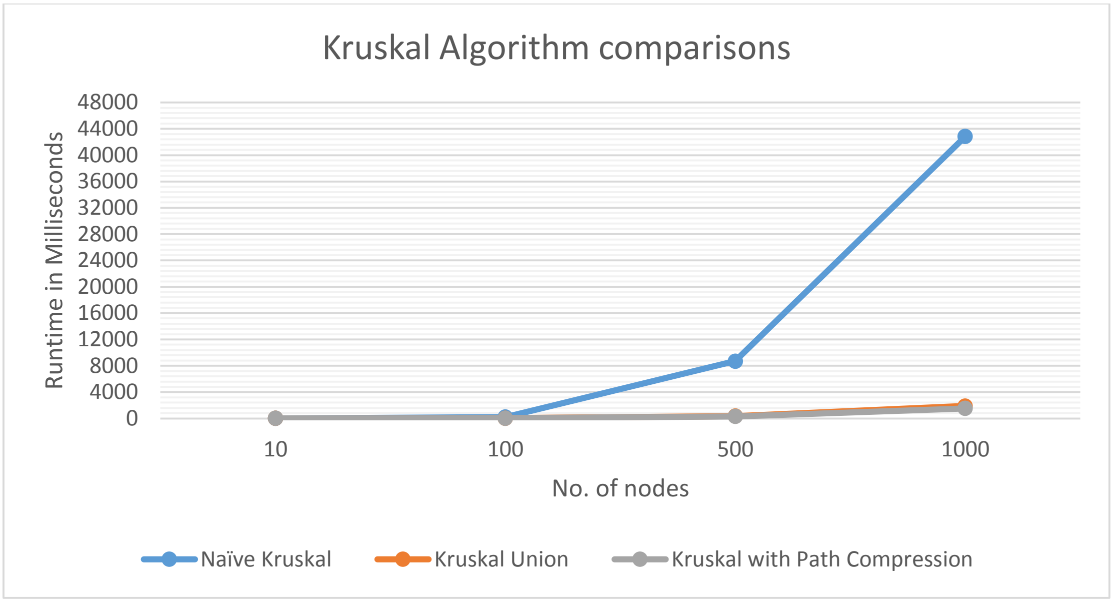
Data Structures Used:-

- **Array List: -** The graph is stored as an array list of nodes.
- **Binary Heap: -** Binary heap is used to store the candidate edges.
- **Linked Hash Set**:- Linked Hash set is used to store the nodes visited during pre-order traversal of the MST.

Experimental Analysis:-

1. The runtimes of the different variations of the Kruskal's Algorithm were compared for a set of inputs and the results are shown below.

No. of Nodes	Naïve Kruskal's Runtime(millisecs)	Kruskal's with Union Runtime(millisecs)	Kruskal's with Path Compression Runtime(millisecs)
10	10	10	10
100	210	40	40
500	8714	350	320
1000	42833	1892	1518

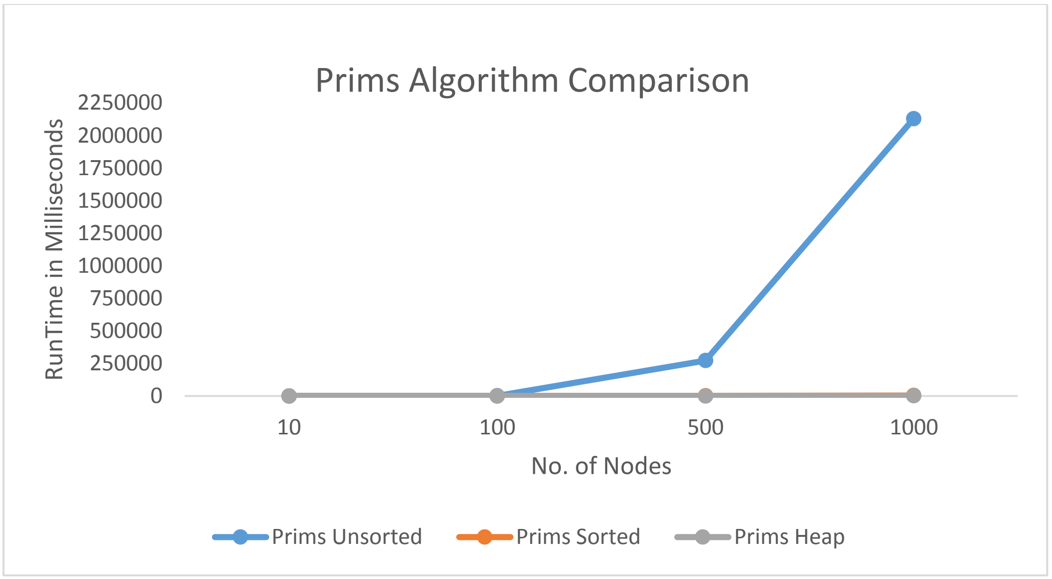


The following observations were made about the performance of Kruskal’s algorithm for different inputs.

- For smaller inputs like 10 nodes the performance of Naïve Kruskal’s, Kruskal’s Union and Kruskal’s with path compression was similar.
- As the size of the input increased Kruskal’s algorithm using Union and Kruskal’s with path compression started performing better than the Naïve Kruskal’s algorithm.
- For large sized inputs Kruskal’s with path compression was 30 times faster than Naïve Kruskal’s algorithm.

2. The runtimes of the different variations of the Prim's Algorithm were compared for a set of inputs and the results are shown below.

No. of Nodes	Prim's Unsorted Runtime(millisecs)	Prim's Sorted Runtime(millisecs)	Prim's Heap Implementation Runtime(millisecs)
10	10	5	9
100	1112	40	99
500	271605	695	499
1000	2129620	3181	999



The following observations were made about the performance of Kruskal's algorithm for different inputs.

- For smaller inputs like 10 nodes the performance of Prim's Unsorted, Prim's Unsorted and Prim's Heap implementation was similar.

- As the size of the input increased, Prim's Heap implementation and Prim's sorted implementation was faster than the Prim's unsorted implementations. This is because the input graph is a complete graph and finding the minimum candidate edge from a large number of edges takes more time, but in case of sorted and heap implementations the time taken to find the minimum weight edge from candidate edges is faster.
- For large sized inputs Prim's Heap implementation was 2000 times better than Prim's unsorted implementation. The primary reason for this behavior is because in a Heap implementation the maximum time is spent only when building the heap. The methods to find the minimum candidate edge takes only constant amount of time i.e. O (1).

3. Travelling salesman problem approximation algorithm runtimes are shown in the table below.

No. of Nodes	TSP 2 –Approximation Run Time (milli seconds)
10	30
100	102
500	1631
1000	2833

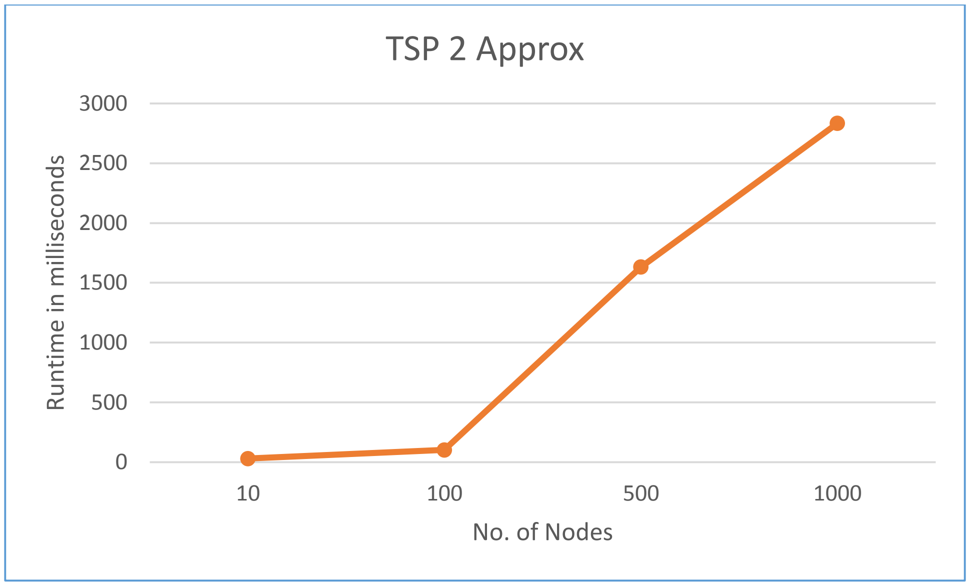
Approximation Algorithm Vs Optimal algorithm comparison

The cost of the overall tour of nodes in the approximation algorithm for 10 nodes was 41.76. The cost for tour in the optimal algorithm was 31.93.

Therefore the Approximation factor = cost of tour in approximation algorithm/cost of tour in optimal algorithm

Approximation factor = 41.76/31.93

We can see that the approximation factor is less than twice the optimal cost.


TSP 2 Approx
Runtime in milliseconds
No. of Nodes