Jack Kapino

Graph Research Project

CS 253 Data & File Structures

Dr. Neli Zlatareva

Kruskal's Algorithm- Muddy City Application

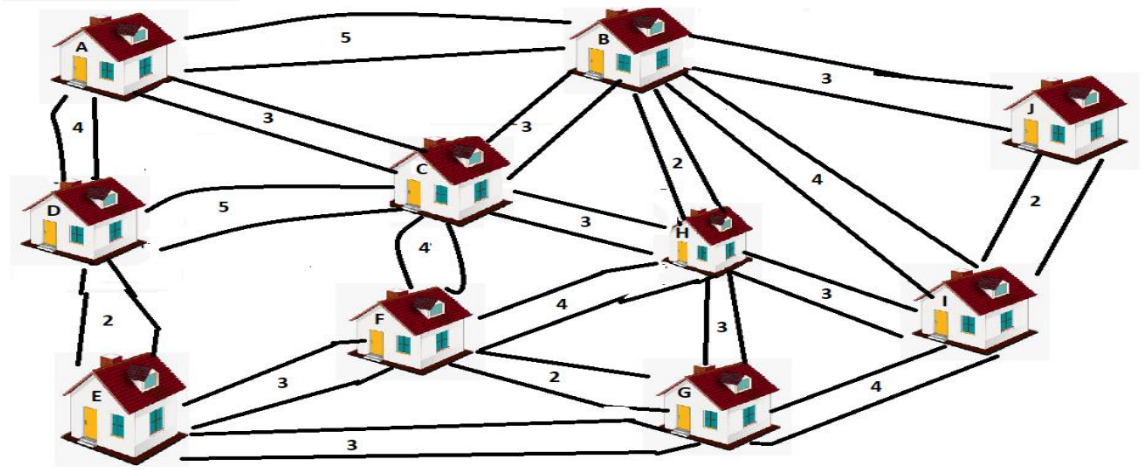
Muddy City Problem: Intro

The muddy city problem is fictional scenario that I came across when researching graph algorithms for this project. This application of Kruskal's algorithm could also be potentially applied real world situations such as construction and pavement of new city roads at minimum cost and maximum use/efficiency. A minimum spanning tree algorithm can also be applicable to telecommunications companies providing cable and telephone wires for a newly developed neighborhood or community. In my example of the muddy city there exists a newly developed city that currently has no paved roads, and due to rain and unfavorable weather conditions the unpaved pathways have become muddy or undesirable to travel on for cars, horses, people. Town officials decide that some of the roads must be paved but not all roads. The idea is to only spend the amount of money necessary for paving the roads. The town officials responsible specify that the paving of roads must follow 2 conditions.

Condition 1: Enough of the streets must be paved such that every citizen can access their own home and every other house through a paved road.

Condition 2: The cost of paving should be a minimal as possible, as long as condition 1 is satisfied for the specified graph given.

A graph layout is given below, the number of paving stones between each house can represent a cost of paving the road(weight of Edge). The task at hand is to find the best route that connects all homes while at the same time using as little (paving stones/ cost) of paving as possible



Nodes/Vertices can represented by houses labeled (A-J) or (A=1 , B=2,…. J=10).

The roads between the houses represent weighted edges, where the weights are the cost of paving that specific road. All houses on the graph must be reachable by a paved road and the cost of paving that road is equivalent to the weight.

Kruskal's Algorithm:

Kruskal's Algorithm takes a greedy approach to solving the minimum spanning tree problem. The algorithm treats nodes as independent trees or disjoint sets that are later connected/merged together by adding the least costly weight to minimum spanning tree until all nodes are included, and no cycles are present, along with (n-1) edges used to connect all nodes where n= number of vertices in graph. The Greedy approach is referred to as selecting the locally optimal choice in hopes that it leads to the globally optimal solution. This paradigm builds a solution one piece at a time always choosing the next option that gives the most obvious and immediate benefit.

A Spanning Tree is a special subgraph of a given graph that contains all the vertices and is maximally acyclic. The Spanning tree of a given graph (G) is the subset of the edges of graph G (E) , that forms a tree and connects to all vertices of (G). Spanning trees have (n-1) edges, n= number of nodes. The total number of spanning trees in a graph can be found in linear O(V+E) time by preforming Breadth-First Search or Depth-First Search.

A naïve approach to solving a minimum spanning tree problem is to compute all spanning trees calculate each of their total weight and choose the minimal one. This approach leads to problems when a graph has an exponential number of spanning trees such as the case with complete bipartite graphs. In Order to improve efficiency of calculating minimum spanning trees algorithms such as Prim's and Kruskal's can be used to show a much more effective way to compute MST's and Minimal cost spanning trees.

Kruskal's algorithm works by building up clusters of disjoint sets, at the beginning each node is its own separate set. The algorithm takes the lightest weighted edge and tests whether the two endpoints lie

within the same connected component/forest. Disjoint Set structures are used to determine if a cycle is created by including the edge, if a cycle is created the edge is discarded and the next edge is removed from the priority queue. This process continues merging sets together until a sing tree path spans over the graph.

Kruskal's Algorithm Time Complexity:

The overall time complexity of Kruskal's algorithm using a min heap as a priority queue and disjoint sets data structure for managing clusters can all be implemented in O(E log E), where E = number of Edges. This is also equivalent to O(E log V) , V= number of nodes, this is because E is at most (V^2) and each node is a separate component of the MST, if you ignore isolated nodes (log V) can be thought of as (log V) = O(log E). Ordering the weighted edges can be done in O (N log N) using a priority queue. If queue is initialized using a min-heap bottom up approach, repeated insertions can be done in O(N) time. The remaining calls of remove element will be done in O(log n) time. Kruskal's Algorithm using the disjoint set data structures must find clusters of nodes with (u,v) as endpoints of an edge E. If the two clusters are different/disjoint then they are merged together to form one cluster which doesn't contain a cycle. In this case Kruskal's will preforms at most (2M) find operations. O(m log n) time needed for ordering the edges dominates that of union find cluster operations so total running time for Kruskal's algorithm is O(E log E).

Pseudo Code taken from textbook po(page 667)

Algorithm Kruskal(G):
Input: A simple connected weighted graph G with n vertices and m edges
Output: A minimum spanning tree T for G
for each vertex v in G **do**
Define an elementary cluster $C(v) = \{v\}$.
Initialize a priority queue Q to contain all edges in G, using the weights as keys.
$T = \emptyset$ {T will ultimately contain the edges of an MST}
while T has fewer than $n - 1$ edges **do**
(u, v) = value returned by Q.removeMin()
Let $C(u)$ be the cluster containing u, and let $C(v)$ be the cluster containing v.
if $C(u) \neq C(v)$ **then**
Add edge (u, v) to T.
Merge $C(u)$ and $C(v)$ into one cluster.
return tree T

Code Fragment 14.16: Kruskal's algorithm for the MST problem.

Min-Heap/Priority Queue:

A priority queue extends the basic queue class by adding properties such as every element has a priority associated with it. The element with the highest priority is removed before elements with lower priority. I choose to implement the priority queue using a min-heap due to better performance in comparison to array or LinkedList implementations of a priority queue Most efficient strategy for implementing Kruskal's minimum spanning tree is to implement a Priority Queue using a min-heap data structure. Ordering the edges by weight using a min-heap is faster than just sorting the edges because you will

only need to visit a small fraction of all edges in a graph before completing the MCST. With the help of min heap priority queue the next edge can be obtained by the remove() operation in O(log E)

Adjacency List:

For the project I choose to implement Kruskal's algorithm using an adjacency list because the graph was sparse. An adjacency list is an array that stores nodes/Edges where the array size is equal to the number of nodes. Adjacency List uses O(n+m) space for a graph with n vertices and M edges.

Method	Running Time
numVertices( ), numEdges( )	$O(1)$
vertices( )	$O(n)$
edges( )	$O(m)$
getEdge(u, v)	$O(\min(\deg(u), \deg(v)))$
outDegree(v), inDegree(v)	$O(1)$
outgoingEdges(v), incomingEdges(v)	$O(\deg(v))$
insertVertex(x), insertEdge(u, v, x)	$O(1)$
removeEdge(e)	$O(1)$
removeVertex(v)	$O(\deg(v))$

Time Complexity of Adjacency List methods: From textbook page 623.

Union Find Data Structure:

Sometimes referred to as the union-find data structure. The purpose is to keep track of a set of elements that have been partitioned into non-overlapping subsets. This data structure usually comes with two methods find and union. The algorithm used in the project uses a union find data structure to determine if the graph contains a cycle or not. Implementation of find() and Union() methods are done in a naïve manner, the worst case runtime for union-find/disjoint set would be O(n) due to the representative tree structure becoming skewed and leading to traversal time similar to that of a linked list. This could be improved by techniques such as Union by rank which uses the size of a tree to determine its rank, this would lead to a worst case runtime of O(n log n). Also using path compression find() can improve the naïve implementation of union-find algorithm. The idea is the find method is called for a search node X, when X is found the find() operation must traverse from the element x up to the root, but with path compression you wouldn't have to traverse the entire list of nodes in order to find the root.

Methods commonly found in Disjoint Set Data Structure:

1. Find(x)- which determines the subset of a specified node, can also be used to determine if two nodes belong to the same subset.

2. Union(Set A, Set B): merges 2 subsets together.

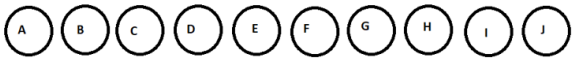
3. makeSet(): used to initialize a set.

An example trace of how the disjoint set works is given below, first all elements are treated as separate components. An edge is removed from the min heap, the 2 endpoints of the edge use find( ) operation to determine which set they belong to. If no cycle forms, then the union method is called, and the two disjoint sets are merged. This repeats till all nodes are connected and the Minimum spanning tree has (n-1) edges.

Example Trace of Union-Find/Disjoint Sets

Disjoint Set Trace:

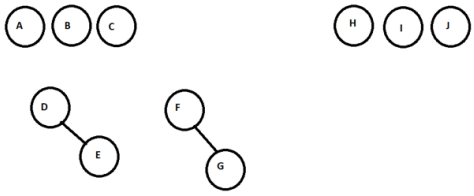
Starting universal set of all nodes.



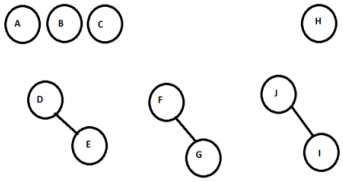
After Removing 1st Edge from Priority Queue:

Edge(D,E)

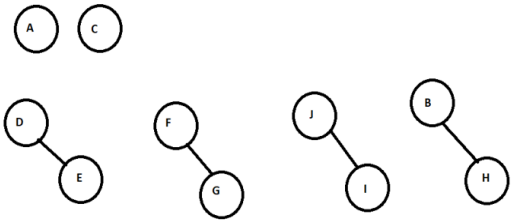
Edge (F,E)



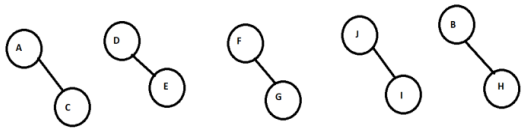
3rd Edge:



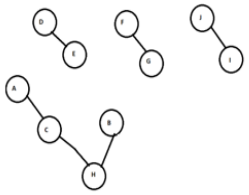
4th Edge:



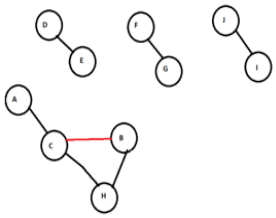
5th Edge:



6th Edge:

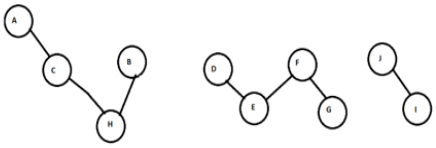


7th Edge: **Discard due to cycle.** Edge=(b,c)

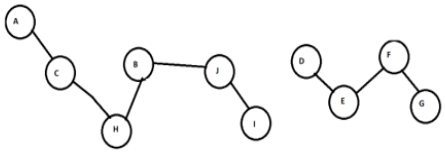


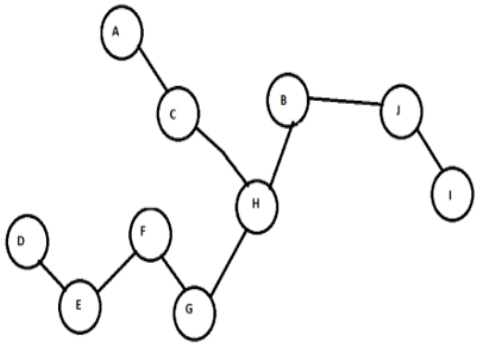
9th Edge (I,G) Final MST

7th Edge(E,F):



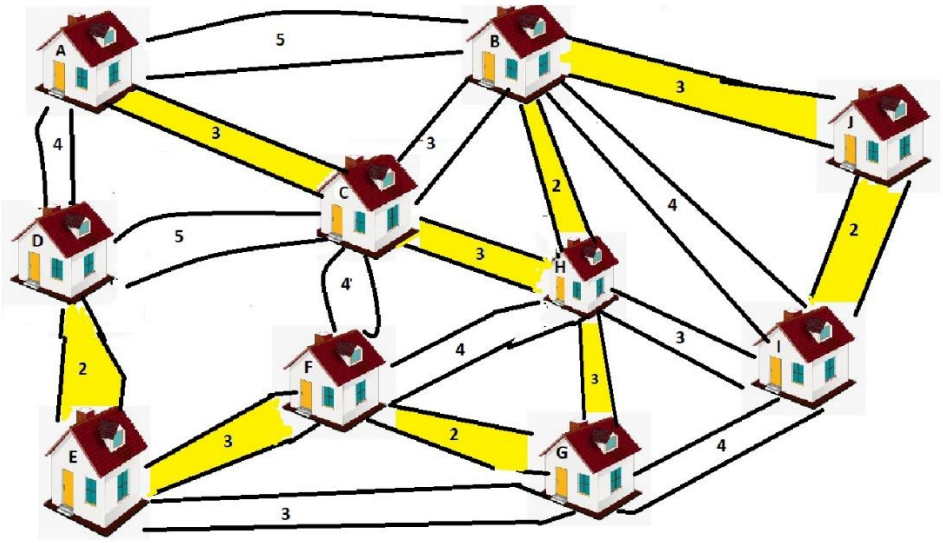
8th Edge(B,J):



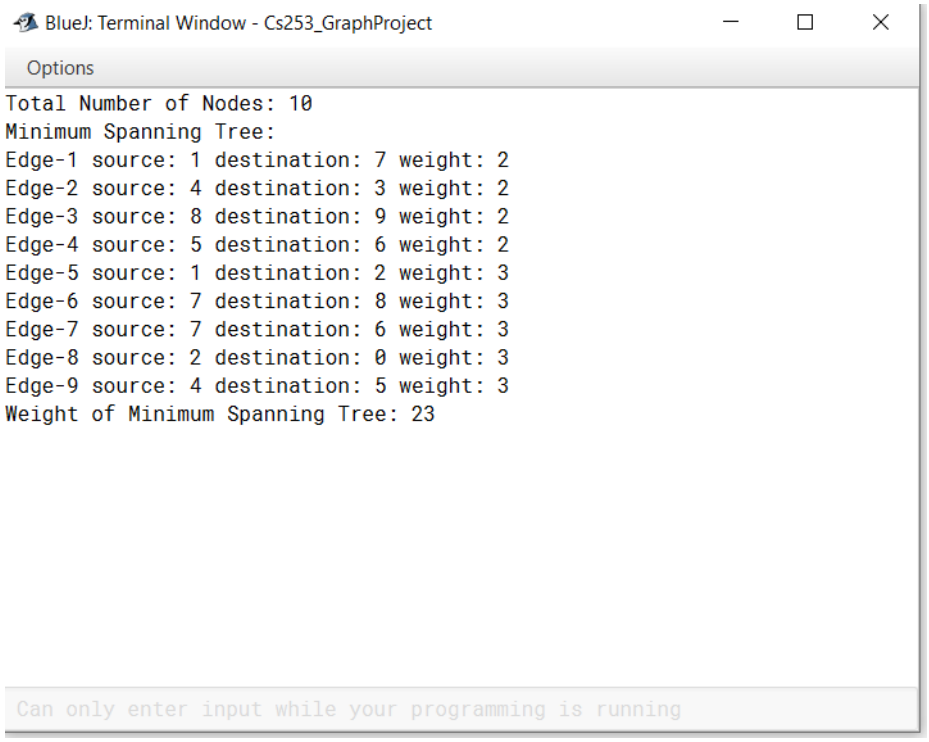


Kruskal's Minimum Cost Spanning Tree: Muddy City Solution

Total weight = **23 units**.



Program output:



Citations:

1. Goodrich, M., & Tamassia, R. (2014). *Data Structures and Algorithms in Java, 6th Edition*. John Wiley & Sons.
2. Sefidgar, R. (n.d.). The Minimum Spanning Tree Algorithm. Retrieved from https://www-m9.ma.tum.de/graph-algorithms/mst-kruskal/index_en.html.
3. CS3 Data Structures & Algorithms. (n.d.). Retrieved from https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/Kruskal.html
4. Pearson (n.d.). Greedy Algorithms. Retrieved from https://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures/04GreedyAlgorithmsII.pdf