DAA Assignment: Project Report for Kruskal’s Algorithm

# Problem statement

The minimum spanning tree of a connected, weighted, undirected graph is a subgraph that connects all vertices without any cycles, i.e. a tree, with the minimum possible total edge weight. Minimum spanning trees are used in a variety of real-world problems, such as vast areas of network design and approximating solutions to the travelling salesman problem.

Kruskal’s algorithm is a commonly used algorithm to find a minimum spanning tree of a given graph. The aim of this project is to implement Kruskal’s algorithm using the union-set data structure, verify its correctness, analyse it in terms of time and space complexity, and compare to Prim’s algorithm, another algorithm used to find minimum spanning trees.

# Description of the approach and method used

Given a graph G = (V,E), Kruskal’s algorithm uses the following procedure to find a minimum spanning tree:

First an empty tree is initialised, and every edge in E is sorted by edge weight. The algorithm then loops through each edge in increasing order, checking if its inclusion in the tree creates any cycles—if not, then the edge is added.

In this project, the union-set data structure is used for the detection of cycles. Along with this data structure are a collection of disjoint-set operations. Every vertex is initially given a pointer to itself and a rank value of zero. This is the set initialisation procedure; every vertex is contained in its own set.

As edges are added, these sets are merged to denote the sub-trees formed—this is done by the union operation. Hence, to avoid cycles, vertices belonging in the same sub-tree should not have a new edge connecting them. To accomplish this, each vertex has pointers that lead up to a vertex called the root of the tree, whose parent pointer is a self-loop. Thus, the root element is a representative for the set.

Each node also has a rank that, which is interpreted as the height of the subtree hanging from the node. The rank value is used instead of explicitly computing tree heights. When merging sets together, the parent pointer of the root of the shorter tree should point to that of the taller tree. This is because tree height affects the computational efficiency, hence this method ensures that the overall height increases only when both trees have the same height .

# Correctness and validation

Definition (cut): A cut is any partition of the vertices into two groups, S and V − S.

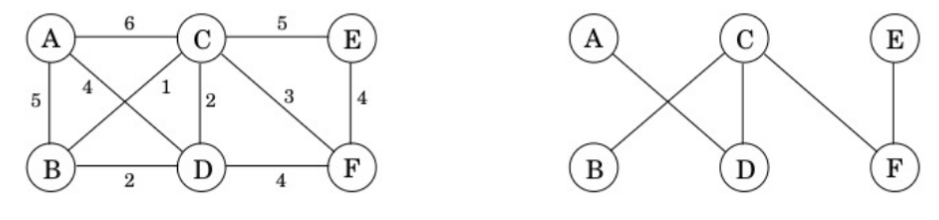
Cut property: Suppose edges X are part of a minimum spanning tree T of G = (V, E). Pick any subset of nodes S for which X does not cross between S and V − S , and let e be the lightest edge across this partition. Then X∪{e} is part of some MST. Inference of this property is that it is safe to add the lightest edges across the cut if X currently has no edges across the cut.

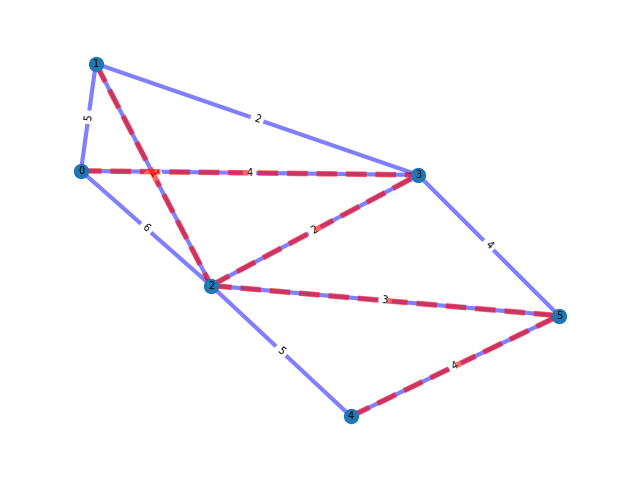
*Proof of correctness:*

Let G = (V,E) be a connected graph. The algorithm has two invariants: at the end of each loop there are no cycles in the graph so far made, and the valid edge with minimal weight is added.

Due to the first invariant, this ensures the final graph T is an acyclic graph. Moreover, since the algorithm loops over all edges, the graph contains every vertex—this is because if there exits a vertex v not in T, then any edge joining v to T cannot form a cycle, hence there must exist a valid edge from v to T, contradicting T’s completeness. Hence, T is a spanning tree.

Due to the second invariant, for any cut (W, G-W) of G, there exists a step where none of the edges between W and G-W are considered. Since each edge is considered by increasing order of weight, it follows that the lightest edge across the cut is considered first and hence accepted. By the cut property, it follows that the resulting graph T is a minimal weight graph. Hence T is an MST.



Above is an example of a graph and the resulting MST found by Kruskal’s algorithm.

This is the same graph generated by the implemented code verifying its correctness in this sample case.

# Complexity analysis

The edges are sorted in O(|E|log|E|) time, by using an appropriately efficient sorting algorithm. Suppose the union and find operations together are O(g(|V|)), then since the loop covers every edge exactly once, the time complexity is equal to O(|E|g(|V|))

Claim: O(g(|V|)) = O(log|V|)

This is easily shown by noting that the time complexity of the union and find operations depend on the maximum rank value of the root vertex, and that any root node of rank k has at least 2^k nodes in its tree. The latter statement can be proved by a simple inductive argument as shown below:

Base Case for k=0:

A node of rank 0 is created with a single node. So, the tree has 2^0=1 node — it holds for the base case.

Inductive Hypothesis: Any root node of rank (k−1) has at least (2^(k−1)) nodes in its tree.

Inductive case for k: To create a root of rank k, a root node's rank increases to k only when two roots, each of rank (k−1), are merged during a union operation. By the induction hypothesis, each of these subtrees rooted at the two nodes of rank (k−1) has at least 2^(k−1) nodes. When these are merged, the new root node's tree contains at least: 2^(k−1)+2^(k−1)=2k nodes. Thus, a root of rank k can only exist if its tree contains at least 2^k nodes. Hence, we are done.

Therefore this proves that the maximum rank of the root vertex is at most log|V|

Hence O(g(V)) = O(log|V|).

Time complexity of Kruskal’s algorithm = O(|E|(log|V|+log|E|))

Since E is at most V^2, log|E|≈ log |V|, hence the overall complexity is O(|E|log|V|).

# Experiments and datasets used

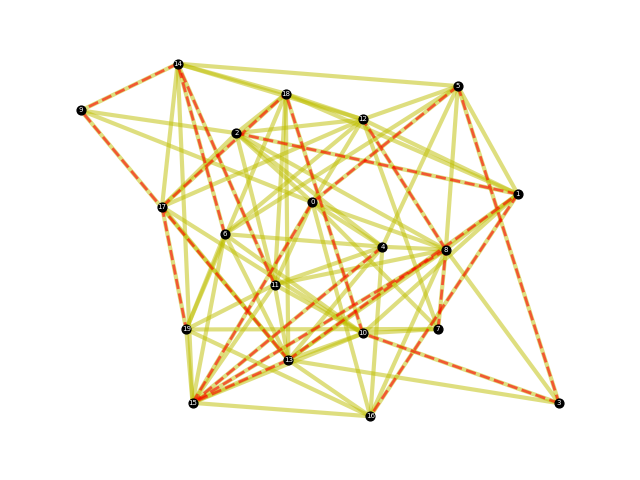
### Test on random graphs with varying edge densities.

The following graphs are of 20, 20 and 50 vertices respectively (for ease of display). The graph edges and edge weights are randomly generated:

G1 = (V1,Ew\_1)

V1 = [0,1,2,…,19]

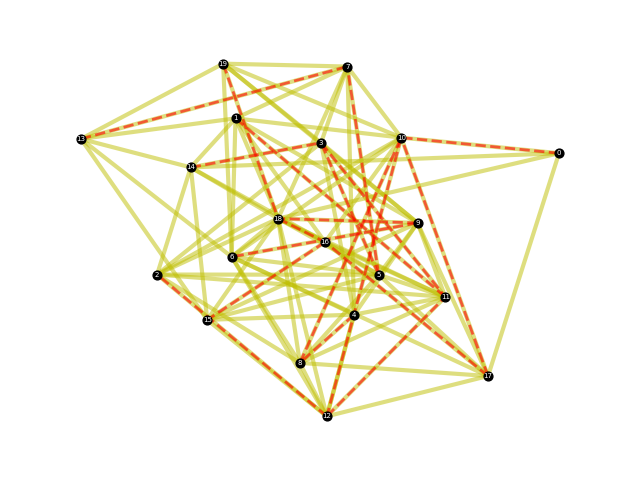
Ew\_1 = [[(17, 10), 42], [(16, 0), 33], [(7, 19), 34], [(12, 1), 26], [(1, 2), 4], [(17, 18), 13], [(8, 12), 14], [(12, 5), 28], [(1, 18), 21], [(2, 0), 42], [(11, 14), 2], [(12, 2), 28], [(6, 15), 18], [(2, 18), 18], [(15, 10), 19], [(5, 8), 16], [(8, 2), 31], [(13, 3), 27], [(10, 7), 47], [(8, 15), 6], [(13, 15), 11], [(13, 10), 22], [(17, 13), 24], [(8, 4), 41], [(13, 9), 7], [(12, 14), 18], [(13, 16), 30], [(14, 18), 44], [(1, 8), 19], [(19, 6), 31], [(13, 6), 35], [(16, 19), 21], [(1, 16), 19], [(11, 0), 39], [(2, 17), 41], [(4, 0), 37], [(5, 6), 22], [(5, 14), 19], [(0, 5), 3], [(15, 13), 42], [(11, 10), 49], [(18, 11), 15], [(3, 10), 14], [(9, 13), 11], [(4, 8), 18], [(6, 4), 17], [(14, 15), 22], [(17, 12), 34], [(19, 11), 48], [(10, 15), 23], [(2, 4), 17], [(16, 15), 37], [(17, 19), 3], [(5, 3), 6], [(16, 8), 34], [(7, 8), 1], [(2, 9), 35], [(11, 18), 15], [(6, 14), 12], [(10, 18), 2], [(16, 4), 22], [(15, 4), 3], [(13, 11), 41], [(18, 12), 36], [(16, 13), 22], [(4, 13), 12], [(11, 8), 43], [(19, 18), 30], [(0, 8), 47], [(13, 1), 1], [(15, 11), 18], [(8, 5), 39], [(15, 0), 1], [(13, 4), 32], [(14, 9), 14], [(7, 0), 49], [(8, 3), 41], [(8, 13), 13], [(18, 13), 18], [(11, 2), 44], [(19, 15), 45], [(12, 6), 30], [(1, 10), 46], [(6, 10), 17], [(7, 12), 32], [(9, 0), 47], [(3, 13), 17], [(11, 4), 21], [(14, 17), 28], [(12, 8), 17], [(19, 17), 27], [(4, 5), 16], [(5, 0), 6], [(5, 1), 29], [(0, 12), 50]]



G2 = (V2,Ew\_2)

V2=V1

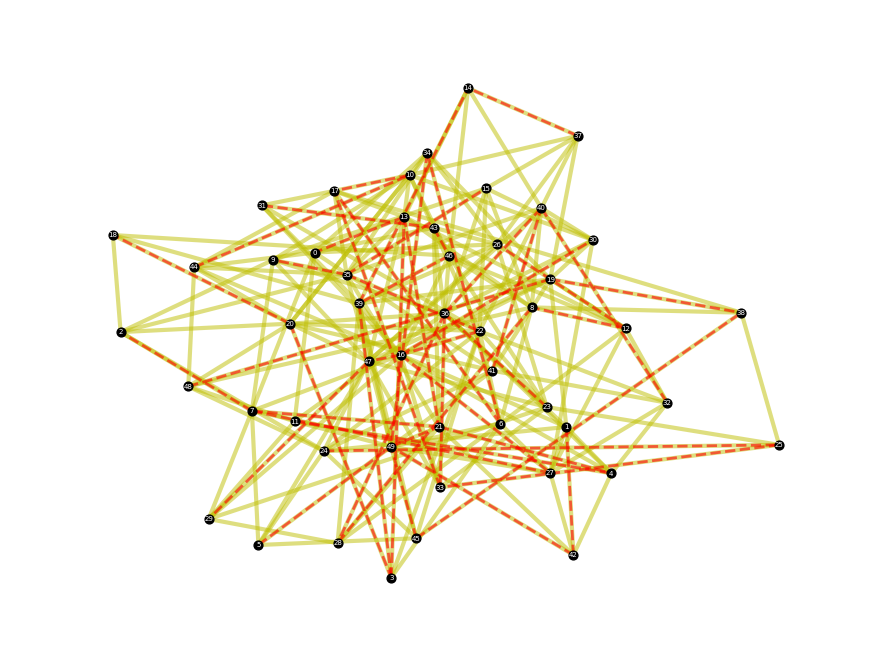
Ew\_2 = [[(18, 11), 48], [(0, 10), 1], [(16, 5), 24], [(0, 18), 8], [(12, 2), 7], [(16, 10), 11], [(2, 8), 38], [(15, 16), 31], [(18, 8), 38], [(18, 14), 47], [(8, 17), 39], [(17, 9), 28], [(17, 0), 16], [(15, 5), 12], [(15, 13), 18], [(7, 1), 23], [(11, 4), 38], [(19, 9), 26], [(11, 8), 34], [(18, 6), 31], [(5, 7), 14], [(15, 14), 16], [(6, 13), 46], [(15, 4), 36], [(10, 1), 41], [(15, 18), 32], [(8, 4), 12], [(7, 19), 28], [(13, 7), 12], [(6, 1), 35], [(14, 3), 1], [(9, 3), 38], [(5, 3), 9], [(10, 19), 20], [(1, 16), 17], [(7, 18), 20], [(4, 12), 39], [(2, 3), 26], [(4, 6), 45], [(15, 12), 16], [(13, 19), 35], [(5, 17), 22], [(18, 2), 13], [(11, 2), 12], [(18, 16), 3], [(10, 0), 25], [(1, 11), 9], [(18, 5), 16], [(1, 18), 21], [(8, 10), 7], [(9, 18), 3], [(1, 9), 14], [(11, 5), 47], [(14, 1), 41], [(17, 16), 4], [(12, 17), 48], [(3, 7), 45], [(17, 10), 1], [(19, 6), 29], [(13, 6), 35], [(9, 11), 48], [(14, 0), 25], [(12, 11), 5], [(16, 18), 35], [(14, 16), 26], [(10, 2), 12], [(6, 17), 11], [(5, 2), 31], [(5, 6), 29], [(9, 6), 6], [(19, 18), 14], [(10, 6), 21], [(7, 4), 19], [(3, 4), 28], [(16, 11), 44], [(3, 11), 4], [(2, 14), 40], [(10, 12), 4], [(12, 18), 9], [(1, 13), 30], [(16, 15), 1], [(3, 14), 32], [(8, 5), 29], [(4, 9), 41], [(4, 8), 9], [(3, 19), 21], [(10, 16), 48], [(9, 5), 30], [(6, 12), 43], [(15, 9), 41], [(8, 6), 47], [(8, 12), 50], [(10, 7), 20], [(13, 14), 48], [(0, 14), 11]]



G3 = (V3,Ew\_3)

V3 = [0,1,2,…,50]

Ew\_3 = [[(49, 41), 24], [(25, 23), 23], [(31, 39), 37], [(35, 22), 2], [(36, 30), 28], [(1, 16), 48], [(19, 34), 33], [(39, 19), 36], [(48, 11), 25], [(22, 6), 36], [(34, 6), 4], [(27, 42), 50], [(16, 27), 3], [(10, 9), 44], [(16, 46), 43], [(47, 20), 20], [(19, 16), 40], [(48, 19), 2], [(9, 2), 35], [(0, 7), 37], [(24, 41), 36], [(13, 8), 45], [(30, 19), 32], [(15, 37), 35], [(22, 16), 5], [(4, 36), 26], [(34, 49), 2], [(40, 32), 4], [(13, 17), 18], [(37, 22), 30], [(47, 24), 14], [(12, 32), 38], [(22, 45), 29], [(31, 17), 30], [(7, 2), 11], [(46, 8), 29], [(13, 14), 38], [(1, 4), 45], [(35, 21), 43], [(32, 27), 28], [(46, 18), 22], [(24, 25), 3], [(12, 13), 18], [(44, 17), 40], [(32, 22), 45], [(43, 31), 5], [(12, 26), 18], [(49, 6), 17], [(46, 12), 40], [(4, 16), 17], [(47, 29), 3], [(22, 24), 16], [(30, 10), 39], [(12, 8), 15], [(26, 47), 37], [(34, 20), 19], [(48, 44), 38], [(21, 43), 48], [(5, 24), 8], [(37, 14), 20], [(42, 21), 50], [(46, 36), 26], [(24, 2), 36], [(20, 18), 16], [(41, 20), 20], [(29, 28), 17], [(8, 28), 13], [(36, 33), 4], [(36, 9), 42], [(35, 13), 30], [(20, 41), 33], [(10, 37), 26], [(21, 7), 6], [(43, 36), 49], [(8, 40), 47], [(45, 38), 4], [(22, 15), 24], [(13, 0), 9], [(15, 8), 39], [(10, 17), 1], [(45, 24), 28], [(23, 36), 4], [(10, 20), 25], [(17, 6), 17], [(7, 29), 18], [(20, 3), 15], [(40, 30), 33], [(39, 14), 13], [(27, 12), 22], [(3, 39), 2], [(11, 4), 17], [(24, 1), 37], [(16, 31), 43], [(40, 35), 23], [(16, 29), 15], [(11, 29), 34], [(2, 35), 19], [(14, 46), 33], [(35, 34), 50], [(47, 9), 39], [(48, 20), 26], [(40, 14), 35], [(0, 43), 47], [(23, 29), 15], [(6, 17), 4], [(43, 26), 27], [(16, 30), 21], [(5, 49), 4], [(36, 15), 43], [(26, 34), 41], [(44, 16), 37], [(27, 6), 28], [(19, 38), 7], [(46, 39), 5], [(4, 32), 46], [(37, 19), 23], [(44, 35), 37], [(15, 9), 45], [(30, 36), 42], [(12, 6), 49], [(49, 35), 42], [(10, 22), 40], [(20, 44), 33], [(7, 16), 36], [(44, 10), 1], [(33, 16), 50], [(25, 38), 40], [(46, 23), 38], [(47, 33), 35], [(26, 0), 39], [(10, 13), 11], [(9, 7), 38], [(3, 13), 4], [(19, 12), 17], [(23, 25), 47], [(22, 20), 32], [(13, 47), 49], [(18, 2), 49], [(42, 1), 31], [(20, 36), 33], [(22, 31), 25], [(25, 33), 3], [(28, 39), 7], [(19, 41), 46], [(14, 39), 16], [(45, 49), 6], [(21, 28), 36], [(47, 19), 17], [(32, 41), 50], [(11, 0), 25], [(33, 17), 6], [(27, 7), 5], [(39, 18), 18], [(27, 49), 26], [(46, 10), 43], [(22, 23), 43], [(38, 8), 34], [(48, 8), 17], [(27, 1), 44], [(17, 35), 14], [(21, 4), 8], [(40, 47), 41], [(2, 20), 17], [(1, 36), 33], [(49, 11), 33], [(1, 19), 36], [(23, 26), 41], [(15, 30), 45], [(16, 47), 8], [(39, 49), 28], [(17, 38), 13], [(11, 36), 48], [(36, 28), 1], [(3, 6), 32], [(49, 21), 6], [(41, 40), 12], [(5, 45), 35], [(6, 28), 12], [(40, 36), 14], [(7, 27), 28], [(5, 47), 49], [(34, 0), 34], [(26, 39), 48], [(41, 0), 24], [(13, 21), 1], [(45, 39), 14], [(44, 0), 22], [(39, 16), 22], [(29, 11), 25], [(5, 7), 28], [(47, 23), 15], [(35, 9), 13], [(42, 49), 19], [(37, 26), 44], [(0, 41), 19], [(35, 15), 23], [(24, 48), 46], [(3, 22), 9], [(8, 6), 40], [(23, 33), 20], [(43, 46), 2], [(11, 48), 50], [(4, 42), 27], [(30, 27), 23]]



From a glance it is clear to see that these are spanning trees of their respective graphs, and by calculation it is possible to verify that they are minimum spanning trees.

### Measure running time.

To measure running time and compare its growth with respect to vertex set size and edge set size, we generate ten experiments for a given vertex size (and edge set proportional to |V|^2) and take the average:

V=100, E=825 : 0.0039419412612915s

V=300, E=7475: 0.0283466100692749s

V=500 E=20791 : 0.107436275482177s

V=700, E=40775: 0.191815972328186s

V=1000 E=83250 : 0.436651682853698s

V=1200 E=119900: 0.63846275806427s

The above graph maps the experimental runtime (blue line) with the calculated theoretical time complexity (orange line). The orange line is calculated as 2\*10^-6 \* |E|log|V|.

Thus this validates the time complexity.

### Analyse union-find data structure efficiency impact.

The union-find data structure brings down time complexity by design by implementing union by rank, thus reducing the operation evaluation to log-time. Additionally, note that the runtime therefore grows much slower than quadratic time. While for small graphs the problem can be solved in roughly linear time due to their size, as the graphs grow larger the algorithm becomes comparatively more efficient than any brute force methods.

# Results and discussion.

Kruskal’s algorithm is known as a greedy algorithm. This means that at every step the locally optimal next choice is taken. In the case of finding a minimal spanning tree of a graph, this method happens to be very effective and efficient at producing desired results. Using more advanced techniques such as randomised algorithms and the use of the soft heap data structure, the time complexity of this problem can be brought to near linear time. However most commonly applied algorithms have time complexity O(|E|log|V|).

In this project, we have successfully implemented Kruskal’s algorithm into code, and discussed its procedure, its correctness, and its efficiency with respect to the union-find data structure used.