KRUSKAL'S MINIMUM SPANNING TREE

The Kruskal's algorithm for generating a minimum spanning tree from an undirected weighted graph was first described in his 1956 paper [1]. Unlike Dijkstra-Prim algorithm, the Kruskal's algorithm focuses on the edges of the graph. A version of an implementation for the algorithm is presented in this assignment the disjoint-set/union-find data structure to represent the graph input with union by rank and path compression. The basic idea behind the algorithm is to add edges in increasing order of their weight to a spanning tree provided that the edge does not form a cycle. The algorithm used in this implementation is discussed in detail below. For the purpose of this assignment, the Python 2.7 programming language was used due to the ease of implementation and flexibility.

DESIGN AND IMPLEMENTATION

The Kruskal's algorithm builds the Minimum Spanning Tree (MST) by initializing vertex as its own tree in a forest. Note that an edge is represented as (edge-weight, vertex1, vertex2); for example, (35, 1, 2) is an edge connecting vertices 1 and 2. Since the graph is undirected in our case, there is no order of the vertices in the edge. The edges are sorted in non-decreasing order by edge weight before initialization of the MST.

Each vertex is treated as its own tree in the initialization process, also known as tree farm method. The algorithm selects the edge with the least weight from the sorted list and uses the **find** method (referred as *find_root()* in this implementation) of the Union-find structure to find the parent of two vertices in the tree. Vertices having the same parents are ignored since adding the edge to the tree will create a cycle; otherwise, the edge is added to the tree using the *union()* method which uses the union by rank strategy. This step is repeated until the list of sorted edges is exhausted or until the MST represents a complete graph.

THE ALGORITHM

More formally, the algorithm below describes the whole process:

Kruskal (V, E)

1. Initialize data structures used for partitioning.
    a. Initialize parent[V] and rank[V].
    b. For each vertex v in V, *parent[v] = v* and *rank[v] = 0.*
2. Sort each edge e in E in non-decreasing order of their weight w.
3. Initialize an empty edge-set S, i.e. the empty MST.
4. For each edge e = (w, v1, v2) in E.
    a. If both vertices have the same parent, i.e *find_root(v1) == find_root(v2).*
        1. Add the edge e to the set S.
        2. Combine the two partitions, i.e *union(v1, v2).*

FIGURE 1 KRUSKAL'S ALGORITHM PSEUDOCODE

An illustrations below describes how the *find_root()* and *union().* For these methods, the edge weight holds no significance since the edges are already picked in non-decreasing order of their weight from the sorted edge list.

PARTITIONING TECHNIQUE

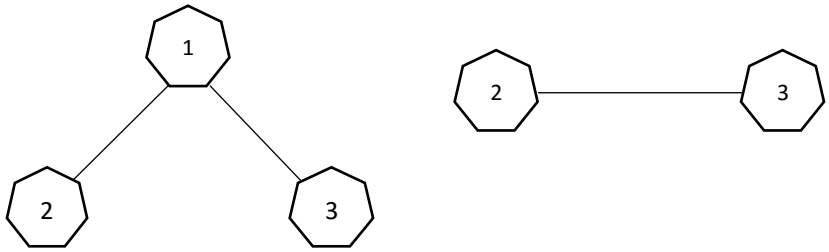

FIGURE 2 VERTEX IS IGNORED IF ADDING TO MST RESULTS IN A CYCLE

Assume the figure on the left shows an MST with edges e1 = (w1, 1, 2) and e2 = (w2, 1, 3). Consider the edge to be added, e3 = (w3, 2, 3). Note that the figure represents two disjoint trees. As you can see, both vertices 2 and 3 have the same parent, i.e. *find_root(2) = find_root(3)*, so e1 forms a cycle. Hence e3 is ignored.

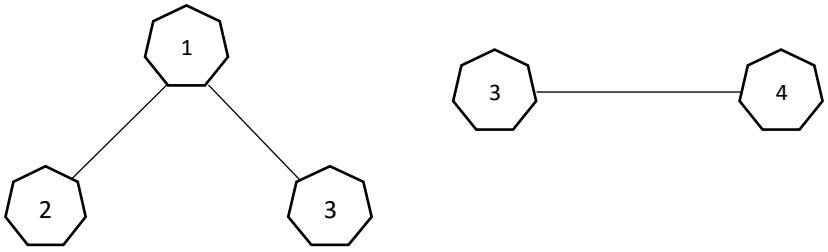

FIGURE 3 VALID EDGE IS ADDED USING UNION-BY-RANK METHOD

Assume that the next edge in order of weight is e4 = (w4, 3, 4). Note that w4 > w3 since the edges are sorted before the MST construction. It can be observed that *find_root(3) = 1*. Note that parent of an orphan vertex is the vertex itself. Hence, *find_root(4) = 4 ≠ find_root(3)*; hence, e4 is added to the MST using the *union()* method.

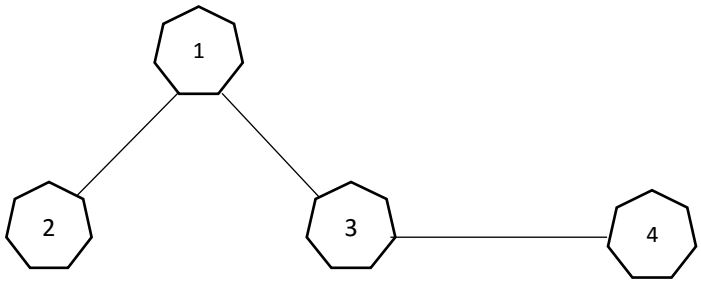

FIGURE 4 MST AFTER ADDING A VALID VERTEX

Since before the union, vertex 1 had two connected edges, its rank is 2. Hence, when the new edge is added, the new parent is considered to be the vertex in the MST with the highest

rank. Hence, the union method updates the parents of all vertices in the MST – 1, 2, 3, 4 as 1. In case of a tie in the rank, a random vertex among the two is chosen. Note that the rank is bounded above by log(|E|) since the highest rank, i.e. the height of a sub-tree, with |V| vertices is log(|V|) and $|E| \geq |V| - 1$.

ANALYSIS

Kruskal (V, E):

1. O(1) for initializing partition structure.
2. O(|E| lg |E|) for sorting.
3. O(1) for initializing MST.
4. O(|E| lg |E|) if priority queue is used *without* path compression and union by rank.

FIGURE 6 STEP BY STEP ANALYSIS OF KRUSKAL'S ALGORITHM

However, since our implementation uses path compression and union by rank instead of priority queue method, the runtime for The space and time complexity of the disjoint-set/union-find data structure and its key methods are given in Table1.

TABLE 1 DISJOINT-SET/UNION-FIND COMPLEXITY OF SPACE AND OPERATIONS [2]

Algorithm	**Average**	**Worst Case**
Space	O(|V|)	O(|V|)
Find	O(α(|V|))	O(α(|V|))
Union	O(α(|V|))	O(α(|V|))

Here, n is the number of vertices and α(|E|) is the inverse of a single valued Ackermann's function, which grows really slowly [3]. However, Tarjan and Leeuwen later discovered in 1984 that the step 4 of Figure 6 can be calculated in O((|E|+|V|) α(|V|)) runtime complexity if any intermixed sequence of $|E| \geq |V|$ *find* and |V| - 1 *union by rank* are used , which is the case for our implementation where α(|V|) is the inverse of Ackermann's function [3]. α grows really slowly, almost linearly [4]. Assume we have a fairly sparse graph, i.e. $|E| \geq |V| - 1$. Since we know α(|V|) = O(lg |V|) = O(lg |E|) and $2|E| \geq |E|+|V|$, the step 4 of Figure 6 can be approximated to O(|E| lg (|E|)).

Therefore, the total runtime complexity of the Kruskal's algorithm with disjoint-set/union-find data structure using path compression and union by rank is the sum of the runtime complexities of all the steps in Figure 6 which results in O(|E| lg |E|).

WORKS CITED

[1] R. B. Muhammad, "Kruskal's Algorithm," [Online]. Available: http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/kruskalAlgor.htm.

[2] R. E. Tarjan, "Efficiency of a Good But Not Linear Set Union Algorithm," *Journal of the ACM,* vol. 22, no. 2, pp. 215-225, April 1975.

[3] Cornell University, "Union Find," [Online]. Available: http://www.cs.cornell.edu/courses/cs6110/2014sp/Handouts/UnionFind.pdf.

[4] R. E. Tarjan and J. v. Leeuwen, "Worst-case Analysis of Set Union Algorithms," *Journal of the ACM,* vol. 31, no. 2, pp. 245-281, April 1984.