Union-Find

The Union-Find data structure is an efficient way to maintain disjoint sets in a group of elements. It has the following methods:

- **Union**(u, v): Joins the set containing u and the subset containing v (performing set union, or ∪).
- **Find**(v): Reports the representative id of the set containing v.

Each subset is organized as a tree. **Unions** are performed by making the parent pointer of one set’s root point to another’s root. The former should be the tree of lesser rank (the shorter tree); the latter should be the tree of greater rank (the taller tree). Maintaining the rank of each subtree may be done during the algorithm without incurring asymptotically more time. A **Find** may simply be performed by following the parent pointers up the tree until the root is found.

Since both these operations will take time proportionate to the height of the tree representations, it is in our best interest to reduce those heights whenever possible. To this end, when performing a **Find**, redirect the parent pointers of all nodes encountered to point to the root (an optimization called path compression). This flattens the tree and reduces subsequent operations’ running times.

The amortized analysis of union-find is complex, but we can take for granted that it can perform union and find in \(O(\alpha(V))\) (where \(\alpha\) is the very slowly increasing inverse Ackermann function\(^1\)).

Generic MST Creation

Suppose we want to create a minimum spanning tree algorithm. We might suppose that a good way to accomplish this would be to maintain an invariant that the set of edges we accrue during construction is always a subset of a valid MST. If at each iteration, we add an edge while maintaining this invariant, we will eventually arrive at a complete minimum spanning tree. We can represent the pseudocode as follows:

- Initialize an empty set \(S\)
- As long as \(S\) is not yet a minimum spanning tree, repeat:
  - Find some edge \(e\) such that \(S \cup e\) is a subset of an MST.
  - \(S = S \cup e\)

While this doesn’t tell us much about the actual implementation details, it is clear why the resulting edge set is correct. The issue comes when determining our edge \(e\). Kruskal’s algorithm offers one solution to that problem.

Kruskal’s Algorithm

This undirected minimum spanning tree algorithm can be described as follows:

- Start off with all vertices as standalone trees within a forest.
- Consider the next minimum edge from the graph.
- Add the edge to the forest if it connects two disjoint trees, otherwise discard it.

\(^1\)https://en.wikipedia.org/wiki/Ackermann_function#Inverse
• Continue this until every vertex has been considered, and return the tree formed.

The premise is simple, but there are two complex operations that are taking place: union — the merging of two subsets into one, and find — determining if two subsets are connected. The fastest data structure for these operations is the union-find data structure, described above. Using the union-find data structure, the run time of Kruskal’s with sorted input is \( O(E\alpha(V)) \). If the input is unsorted, sorting the edges by weight becomes the bottleneck operation, making our running time \( O(E \log(V)) \).

**Kruskal’s vs. Prim’s**

• Both are algorithms to find the minimum spanning tree.

• Prim starts with a single vertex, and grows a tree from this vertex.

• Kruskal starts with every vertex as a separate tree, and combines them to form a single tree.

• Complexity of eager Prim (with a binomial heap): \( O((E + V) \log V) \), complexity of Kruskal (with union-find with path compression): \( O(E\alpha(V)) \).

• In Kruskal, checking to see if adding an edge will create a cycle can be slow. Thus, Kruskal’s algorithm works better when there are fewer edges to vertices.

• Prim’s algorithm works better for dense graphs with more edges than vertices.

• Since Prim’s algorithm “grows” the tree by adding vertices, it always has a partial tree. If you only need a partial solution, use Prim.

**Testing your understanding**

**Problem 1.** Is it guaranteed that a call to \( \text{Find}(v) \) will always return the same result throughout the algorithm? If not, is it possible to modify the algorithm such that it does?

**Problem 2.** How could one use Union-Find to detect cycles in an undirected graph?

**Problem 3.** Does Kruskal’s algorithm work on a graph with negative weights?

**Problem 4.** Say we have some MST, \( T \), in a positively weighted graph \( G \). Construct a graph \( G' \) where for any weight \( w(e) \) for edge \( e \) in \( G \), we have weights \( (w(e))^2 \) in \( G' \). Does \( T \) still remain an MST in \( G' \)? Prove your answer. Now if \( G \) also had negative weights, would your answer change from the previous part? Prove your answer.

**Problem 5.** Imagine we have a graph \( G \) where all edge weights are equal. Design an algorithm to efficiently find an MST of \( G \). Analyze the running time.

**Problem 6.** Suppose that we have found an MST \( T \) of a graph \( G \), but soon after, we are told that an edge in \( T \) has a higher weight than we at first thought, and as such our MST is now invalid. Is it guaranteed that we can fix our tree by removing an edge and adding a different one? If so, explain how. If not, provide a counterexample.

**Problem 7.** Suppose that we have found an MST \( T \) of a graph \( G \), but soon after, we are told that an edge not in \( T \) has a lower weight than we at first thought, and as such our MST is now invalid. Is it guaranteed that we can fix our tree by removing an edge and adding a different one? If so, explain how. If not, provide a counterexample.