02-613 Week 2

Algorithms and Advanced Data Structures

Aidan Jan

September 5, 2025

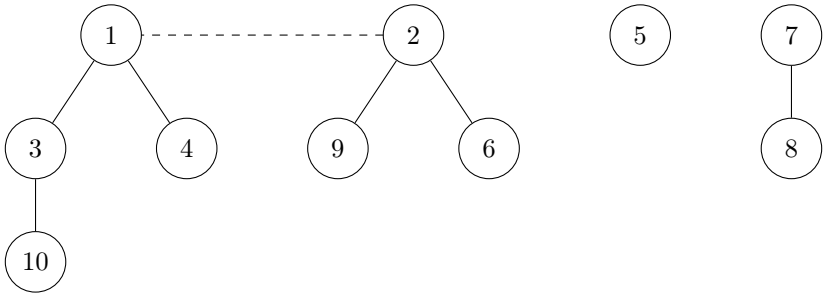
Union-Find Abstract Data Type

In Kruskal's algorithm, we need a data structure to quickly find cycles. We need a way to:

- quickly determine which subgraph a given node is in
- quickly merge two subgraphs
- a constructor for the data structure

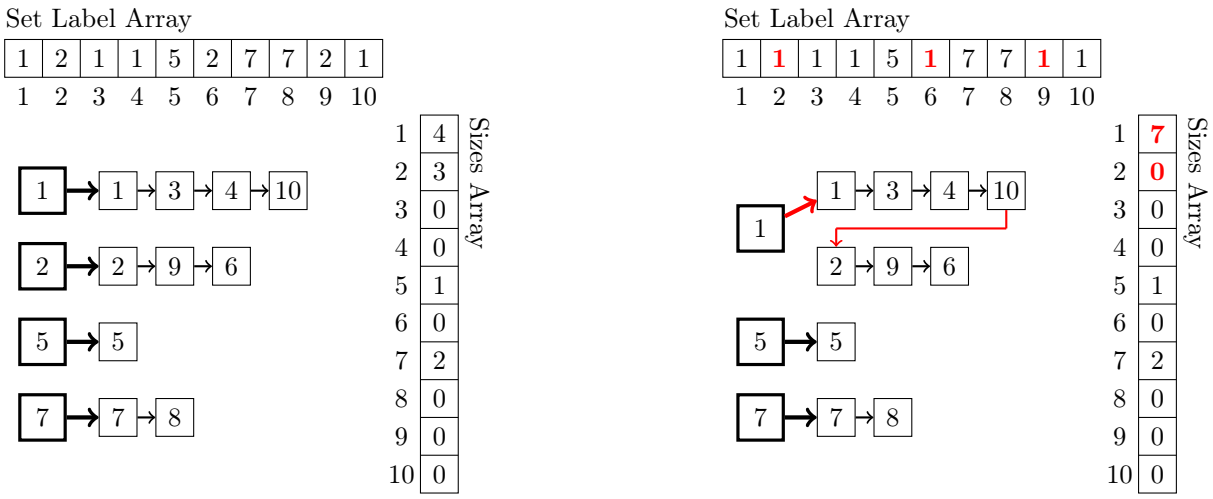
One data structure we can use is the Union-Find data structure. What we basically do is store a collection of trees.

- Create a struct for a node, containing the connections for the graph in an array, but also, each has an additional variable, a parent pointer.
- To determine which group the node is in, follow the parent pointers up until you get to a node with no parent pointer. (This gives every subgraph a unique identifier.)
- To merge two groups, simply take the root of the smaller tree, and add the root of the bigger tree as a parent to the smaller tree.
  - To do this quickly, we need a size array. Without size array would be $O(n)$. With size array would be $O(1)$.



- In the diagram above, to find the node's group, simply follow the tree up to the root. The root is the node's group.
- In the image above, we would merge trees 1 and 2 by making the parent of node 2 node 1. (see dotted line)

We can actually optimize this further if we convert the structs to arrays. The below image represents the same structure as above:



- The diagram on the right shows the result after merging groups 1 and 2. Since 2 is a smaller group, it is merged into 1. Notice the changes in the set label and sizes array.
- This increases the speed of the find operation from $O(\log n)$ to $O(1)$.
- However, this decreases the speed of the union operation from $O(1)$ to $O(k \log k)$, where k is the number of unions that has occurred before.
- This is an upgrade since we will likely use the find operation much more than the union operation.

Aside: theorem

After k unions on n items, $O(k \log k)$ total time.

Proof. k unions touches at most $2k$ items, and any item v is relabeled $\log_2(2k)$ times. This is because we start with each vertex in its own group, and every union at maximum doubles its size. As a result, each union has $2k \log_2(2k)$ of work, which is $O(k \log k)$. In the worst case, this converges to $O(\log n)$. □

Kruskal's Runtime

With this data structure, we can now calculate the running time of Kruskal's algorithm:

- Sort the edges: $O(m \log m)$
- Test every edge: $O(m)$ edges $\times$ $(2 \cdot$ `find()`$)$ + `union()`
- Therefore, the total time is $O(m \log m + 2m + m \log n)$. Since there are more edges than vertices, the $O(m \log m)$ dominates. Sorting the edges takes the longest amount of time.

Heaps

A **heap** is a data structure that has the ability to provide the smallest (or largest) value in constant time. For example, a series of integers can be added to the heap, one at a time, and removing the root of the heap would provide the values back in increasing order. One inspiration of this structure requires us to go back to Prim's Algorithm.

Implementation of Prim's Algorithm

1. Define `distToT[]` as an array of length `numVertices`, and initialize every value to `INTEGER_MAX`
2. Let `u` be an arbitrary node
3. Set `distToT[u] = INTEGER_MIN`
4. Set `distToT[v] = d(u, v)` if smaller for $v \in \text{Neighbors}(v)$
5. Set `parent(v) = v` if updated
6. Set `u = ClosestVertex(T)`
7. While `v` exists, go to (3).

Aside: Asymptotics

Asymptotics is a way of measuring running time relative to input size. This is often written as the Big-O notation.

- $x = 4 \cdot y$ is O(1), since no matter the inputs, there are 2 operations. ($\cdot$ and $=$). The exact number of operations does not matter, just that it is a constant.
- $\forall i \ : \ z[i] = 4 + w[i]$ is O(n), since there are 2 times $|z|$ operations. The number of operations scales with the length of $|z|$, or n.
- $\forall i, j \ : \ a[i] = b[i] \cdot b[j]$ is $O(n^2)$, since there are $2 \cdot |b| \cdot |b|$ operations.

Remember that what we care about is not the exact number of operations, but rather what the operation count curve would look like as the number of elements approaches infinity. Therefore, a line with $2n$ operations and one with $3n$ operations are both $O(n)$, since when approaching infinity, they are both straight lines. Similarly, a function with n^2 operations and $n^2 + 500n$ operations are also both $O(n^2)$, since when approaching infinity, the n^2 term will dwarf the $500n$ term, so they are both $O(n^2)$.

- You can always drop the coefficient and terms that are not of the highest power when writing Big-O notation.
- Some other consequences of this include that
  - Factorial time is worse than Exponential time, which is worse than any polynomial time. In polynomial time, higher powers are worse than lower powers.
  - $\log(n)$ lies between constant time and n, so by the same logic, $\log(n)$ is better than n, and $n \log(n)$ is better than n^2
  - Since coefficients can be dropped, $O(\log_2(n))$ and $O(\log_3(n))$ are actually equivalent.

Going back to Prim's algorithm, if we do the runtime analysis, the slowest steps are steps 4-6, which without including the loop, takes $O(m)$. With the loop, it is $O(m \cdot n)$. Can we improve this?

- The issue is that when we search for the closest vertex using an array, we have to search every vertex of the graph, which is why that specific step is $O(m)$.
- We can make this faster! This goes back to the heap.

As it turns out, the heap allows you to get the smallest value in the entire array in O(1) time. This is much faster than $O(n)$.

Heap Structure

A heap uses a binary tree structure, where the smallest element is always at the root. Suppose we want the smallest element. This is O(1) time since we can just extract the root! However, we have to do some other things to maintain the structure.

- Find() is O(1), as described.
- Insert() is O($\log n$).
  - First, insert node as a leaf at the bottom of the tree.
  - If that node is smaller than its parent, swap its position with its parent.
  - Repeat until the node's value is larger than its parent.
  - Note that to optimize the number of times to "bubble" up, we need an array to keep track of pointers to the leaves for quick access, but also to ensure the lengths of all the branches are approximately equal.
- Delete() is also O($\log n$)
  - Pop the node to be deleted (often the root).
  - Replace the node with the *smaller* child, if there is one.
  - Repeat until there are no more children.

Delete and insert are O($\log n$) since they are dependent on the depth of the tree. Since the tree is (ideally) a complete, binary tree, the height is on the order of $\log n$, where n is the number of nodes in the tree. [insert diagrams for delete/insert]

Note that a heap can be stored in an array.