An Analysis of Tree Algorithms and Data Structures

Bill Lin

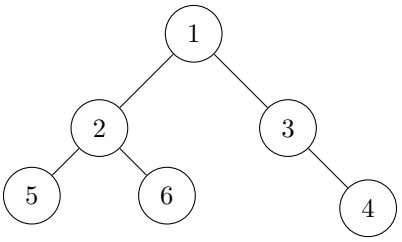
1 Introduction

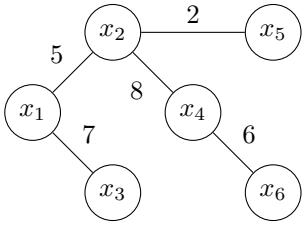
The purpose of this paper is to analyze a certain collection of tree algorithms and tree data structures, and to give an intuition as to why they work.

We will first define what a tree is and look at its properties. A tree is defined as a **connected acyclic undirected graph**. The **connected** part means that every node can be reached from every other node. **Acyclic** means that the tree does not have a sequence of edges that joins a sequence of vertices that starts and ends on the same vertex (basically, no cycles). **Undirected** means that the edges in the tree go in both directions. As a result of these properties, trees will always have $n-1$ edges, where n is the number of nodes.

Why would trees always have $n-1$ edges? Let's start with the simplest case: a single node. With only a single node, there will be no edges, as the only edge we could add is from the node to itself, which would create a self-loop and would create a cycle. If we were to add a new node, we would need exactly one edge from the previous connected tree to the new node in order to make the graph connected. So, that means the tree starts with 1 node and no edges, and every new node added would require one extra edge. Thus, the number of nodes will always be one more than the number of edges in the tree.

Here are a couple of examples of trees:





2 Minimum Spanning Trees

A **minimum spanning tree (MST)** of a graph is some subset of the edges where the resulting subset of edges forms a tree that connects all of the nodes in the graph, with the smallest possible sum of edge weights. To find the minimum spanning tree, there are two algorithms that is normally used: **Kruskal's**, and **Prim's**.

Note: If the graph is not connected, then there is no MST.

2.1 Kruskal's MST Algorithm

Prerequisites: Disjoint Set Union

Kruskal's algorithm is a greedy algorithm that is guaranteed to give a minimum spanning tree. If there are multiple MSTs, it will give any one of them.

Firstly, we will sort all the edges in the graph by increasing order of edge weight. Then, we will loop through all of the edges in the sorted order, and we will only add the edge **if and only if it connects two nodes in separate components.** This means that we will need a quick way to determine if two nodes are in the same set. We will need a data structure that is able to merge two nodes from separate components, and which can also determine if two nodes are in the same set, with relatively fast time complexity. The **disjoint set union** data structure is the perfect fit for this, as it can merge two components and it can also find if two nodes are in the same component, both with $O(\log n)$[1] time complexity.

[1] In this paper, log is defined as $\log_2$.

The following algorithm only finds the cost of the MST. To find the edges in the resulting MST, we can just add the edges into a list.

2.1.1 Pseudo-code for Kruskal's

```
edges = ... // list of edges, in the form [cost, edge begin, edge end]
sort edges by cost

dsu = initialize disjoint set union
total_cost = 0

for each [edge_cost, u, v] in edges:
    if u and v are not in the same component:
        add edge_cost to total_cost
        merge node u and v in the dsu
```

2.1.2 Time complexity analysis

Sorting the edges takes $m \log m$ time, and merging in the DSU takes either $O(\log n)$ or $O(\alpha(n))$ (where $\alpha(n)$ is the inverse Ackermann function), depending on the implementation. We will have to merge nodes every time we add a new edge. Since there are $n-1$ edges, the total amount of time taken will be either $O(n \log n)$ or $O(n \cdot \alpha(n))$. Thus, the final time complexity is $O(n \log n + m \log n)$ or $O(n \cdot \alpha(n) + m \log n)$.

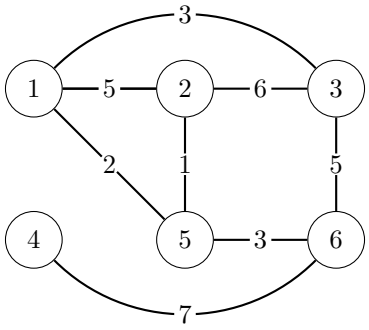
2.1.3 Intuition for Kruskal's

Why would this method always work? It always selects the unprocessed edge with least cost, but what if this isn't optimal to select this edge?

Let's think about this. We will call the cost of the unprocessed edge with least cost c, and the two vertices it connects u and v. If u and v are already in the same component, we don't add the edge. However, if they are not, it is always optimal to add this edge into the MST. Why? Since, in the final MST, all vertices are in the same connected component and there must be $n-1$ edges, by not choosing this edge, we will have to choose another unprocessed edge later on that connects u and v. But remember that we have already sorted the list of edges by their edge weight, so any other unprocessed edge will have an edge cost that is greater than c. Therefore, it is always optimal to select the unprocessed edge with least cost if it connects two different components.

2.1.4 Example of Kruskal's

Let's say we have the graph below, and we want to find the MST of this graph.



Here is the list of all the edges, sorted by their edge weight. Edges with equal weights are just placed in random order, since it won't affect the MST.

weight	first node	second node
1	2	5
2	1	5
3	1	3
3	5	6
5	1	2
6	2	3
7	4	6

This following table will display which set each node is in, handled by the disjoint set union data structure. For simplicity purposes, while merging two sets, it will choose the set with the parent with the least number, even though in practice, the DSU will use other methods of merging. Changes are highlighted throughout the steps.

Node	1	2	3	4	5	6
Parent	1	2	3	4	5	6

Note: $[u, v]$ refers to the edge connecting node u and node v.

Let's run the algorithm, step by step. First of all, we will choose the edge [2, 5], since it has the lowest cost. Nodes 2 and 5 are in different sets, so let's merge nodes 2 and 5 and add this edge to the MST.

Node	1	2	3	4	5	6
Parent	1	2	3	4	2	6

Next, we consider the edge [1, 5]. Nodes 1 and 5 are in different sets, so let's merge them in the DSU and add this edge to the MST. Since node 2 is also part of the same set as 5, we will also set parent of 2 to 1.

Node	1	2	3	4	5	6
Parent	1	1	3	4	1	6

For edges [1, 3] and [5, 6], both nodes in the edges are in different sets, so the same merging process applies. Changes are shown below.

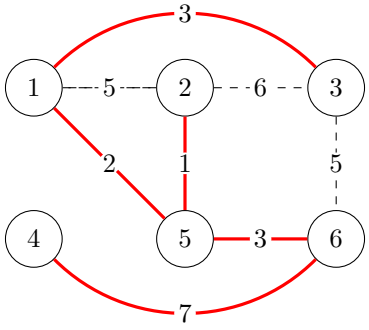
Node	1	2	3	4	5	6
Parent	1	1	1	4	1	1

Now we consider the edge [1, 2]. Both nodes 1 and 2 are in the same set (set 1), so we will skip over this edge. Similarity, for the edge [2, 3], nodes 2 and 3 are also both in set 1, so this edge is skipped. Recall: if we add an edge, whose two nodes it connects are in the same set, we don't make any new connections, so we can skip over this edge. No changes are made.

Lastly, considering edge [4, 6]. Node 4 is in set 4, and node 6 is in set 1, so we will merge these two.

Node	1	2	3	4	5	6
Parent	1	1	1	1	1	1

Since every node is in the same set, the graph made by the edges we chose is connected. Thus, the resulting minimum spanning tree, determined by the edges we took, is the following (indicated by the solid red lines):



The cost of the MST is the sum of the edges we took, which is $1+2+3+3+7=43$. The edges in our MST are [1, 3], [1, 5], [2, 5], [5, 6], and [4, 6].

2.2 Prim's MST Algorithm

Recommended: Dijkstra

Prim's algorithm is also a greedy algorithm which, similar to Kruskal's, finds the minimum spanning tree of a graph. Prim's algorithm is very similar to Dijkstra. We need a priority queue that sorts by the cheapest edge cost. We will also need a boolean array to mark which nodes we have already visited. We then select any node (typically node 1), mark the node as visited, and then push all the edges into a priority queue. We take the cheapest edge in the priority queue and process it. We will only add the edge into the MST if it connects a marked node and an unmarked node. This process is repeated until we have $n-1$ edges, which will give us the MST.

In Prim's algorithm, we do not need a disjoint set union like Kruskal's, as we are using the boolean array to determine if we take edges or not.

2.2.1 Pseudo-code for Prim's

```
adj = ... // an adjacency list, storing [destination, cost]
q = priority queue, sorting by least edge weight // pairs of [dest, cost]

vis = boolean array of size n, initialized to false
push [1, 0] into q // push node 1 into the priority queue, with cost 0
                   // so it doesn't affect MST cost

total = 0 // cost of the MST
while q is not empty:
        dest = q.first()'s dest, cost = q.first()'s cost
        pop q's first element

        if dest is already visited
                continue

        vis[dest] = true
        add cost to total

        for [to, cost] in adj[dest]:
                push [to, cost] into q
```

2.2.2 Time complexity analysis

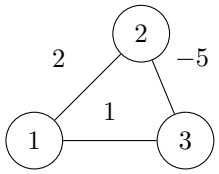
The main bottleneck for this algorithm is the priority queue, as there are a total of m edges that needs to be inserted in the priority queue, each taking $O(\log n)$ time. Other operations like initializing the boolean array, popping top element from priority queues, etc. are all either $O(n)$ or $O(1)$, so the final time complexity of this algorithm is $O(m \log n)$.

2.2.3 Intuition for Prim's

Prim's will keep taking the minimum cost edge that connects a marked node to an unmarked node until the MST is found. This is quite similar to Dijkstra. It should seem reasonable that a greedy method would work for MST, just like how it works for finding shortest paths. By taking the edge with least cost that connects a marked node to an unmarked one, we keep connecting two separate components while adding the least possible cost to our total MST cost. However, at first glance, it seems like negative weights may cause this algorithm to give an non-optimal result.

Dijkstra's does not work for negative weights, so why does Prim's? Since we know that we will have $n - 1$ edges, we can just add a big constant factor c such that $c \geq |\text{cost of smallest edge}|$, so that all of the edges becomes positive, and we can just subtract $c \cdot (n - 1)$ from the final. But we didn't add c to all the edges, so why does this still work?

Consider the following graph:



We would first start from node 1. We would take the edge [1, 3] first, and then immediately consider the edge with a negative cost. It would not be optimal to take edge [1, 2] just because it might lead us to a negative edge weight, since if we can reach the node with a negative cost edge using edges that have lower cost, it would result in a lower total cost of the MST.

3 Binary Lifting, Euler Tour, and Lowest Common Ancestor

Let's say we have a tree, and we have q queries, each one of them asking us to find the k-th parent of a node u. We could just have an array that keeps track of every node's parent, start at node u just move up the tree k times, but this will give us a time complexity of $O(k)$. If there are $q = 10^5$ queries, our final time complexity is $O(qk)$, which in most cases will be too slow for a given problem. Therefore, we will use a technique called **binary lifting** to reduce each one of these queries to $O(\log n)$.

3.1 Preprocessing Jumps

For each node, we will calculate the 2^k-th ancestor for $0 \leq k \leq \lceil \log(n) \rceil$. This allows us to "jump up" any power of two, from any node. We can calculate the jumps while doing the DFS. Let us define an 2-d array lift[][], with first dimension of size n and the second dimension with size $\lceil \log(n) \rceil$. Every time we process a node u, we set lift[u][0] = parent of u, and then we can loop i from 1 to $\lceil \log(n) \rceil - 1$, defining lift[u][i] = lift[lift[u][i - 1]][i - 1]. This means that we go up 2^{n-1} nodes from the 2^{n-1}-th parent of u. This works because going up by 2^{n-1} and then 2^{n-1} again is the same as going up 2^n: $2^{n-1} + 2^{n-1} = 2(2^{n-1}) = 2^n$. Since we are doing a DFS, any jump from a parent node is already calculated, so lift[lift[u][i - 1]][i - 1] will already be calculated since node lift[u][i - 1] will be processed before node u.

After processing the jumps, we can go up any amount in the tree from any node in $O(\log n)$ time. Let's say we are asked to find the k-th parent from the node. This is the same as jumping up k times. We can loop i from $\lceil \log(n) \rceil$ to 0, and every time $2^i \leq k$, we can jump up 2^i nodes and

subtract 2^i from k. Eventually, k will equal 0, and thus we will have jumped up the desired amount of nodes.

3.1.1 Binary Lifting Code

The code in this entire section is based off of the cp-algorithms article "Lowest Common Ancestor - Binary Lifting." Click this to see more.

```
let n = number of nodes in the tree
adj = adjacency list of tree

let k = ceil(log(n))
let lift = 2d array of size n and size k + 1

let timer = 0
void dfs(int node, int par) {
        lift[node][0] = par
        for i from 1 to k-1:
                lift[node][i] = lift[lift[node][i-1]][i-1]

        for child in adj[node]:
                if child != parent
                        dfs(child, node)
}
```

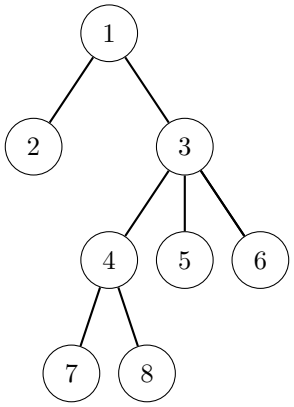
3.1.2 Time complexity analysis

We can calculate all the "jumps" in $O(1)$. Since there are $\lceil \log(n) \rceil$ jumps we need to calculate for all of n nodes, the time complexity will be $O(n \log n)$.

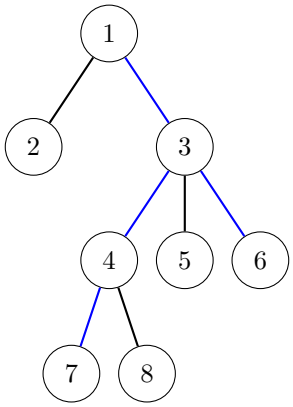
3.2 Lowest Common Ancestor

For two nodes u_1 and u_2, the lowest common ancestor v is the farthest node from the root which has both u_1 and u_2 as descendants. We define a v to be a descendant of u if the path between u and v only goes "down" the tree. More formally, it is any node that can be reached as long as we cannot choose the edge that goes from the node we're currently on to its parent node. A node is also defined to be a descendant of itself, so it is possible that v is one of u_1 or u_2.

Consider the following tree, rooted at 1:



Let's take a look at an example query. Let's find the LCA between nodes 7 and 6. There are two nodes that have both 6 and 8 as descendants: nodes 1 and 3, as they can both reach nodes 6 and 7 by using the blue edges, and we can see that we the paths only go "down" the tree. We will take node 3 as it is farther away from the root node (node 1), than node 1.



Now, how can we calculate the LCA of two nodes quickly?

3.3 DFS Traversal/Euler Tour

The first step is to do a **DFS traversal** on the tree. We keep a variable called *timer* to keep track of when we enter and exit each node. We start at the root of the tree (again, usually node 1), and run a DFS until all the nodes are visited. Every time we process a node, we add one to the timer and store this value in an array, and we do the same when we finish processing the node, storing this value in another array. This is also called the **Euler tour** of the tree.

Why is this necessary? While processing the queries, we need a quick way to see if one node is the ancestor of another. The information we get from the DFS traversal will allow us to do this in $O(1)$.

3.3.1 DFS Traversal/Euler Tour Code

```
let n = number of nodes in the tree
adj = adjacency list of tree
let tin = integer array of size n // time in for each node
let tout = integer array of size n // time out for each node

let k = ceil(log(n))
let lift = 2d array of size n and size k + 1

let timer = 0
void dfs(int node, int par) {
        timer++
        tin[node] = timer

        lift[node][0] = par
        for i from 1 to k-1:
                lift[node][i] = lift[lift[node][i-1]][i-1]

        for child in adj[node]:
                if child != parent
                        dfs(child, node)

        timer++
        tout[node] = timer
}
```

Notice how we simultaneously calculated jumps while doing the DFS traversal.

3.3.2 Determining if u is an ancestor of v in constant time

Now that we have completed the DFS traversal, we can now determine whether u is an ancestor of v in constant time using our *tin* and *tout* arrays. u is an ancestor of v if and only if **tin[u]** $\leq$ **tin[v] and tout[u]** $\geq$ **tout[v]**. tin[u] $\leq$ tin[v] is asking if we have entered node u before node v, and tout[u] $\geq$ tout[v] is asking whether we have exited node v before node u. The only way that this condition is true is if v is in the subtree of u, and thus node u would be an ancestor of node v.

The following code implements the algorithm above:

```
boolean is_ancestor(int u, int v) {
        return tin[u] <= tin[v] and tout[u] >= tout[v]
}
```

3.4 Calculating LCA in $O(\log n)$

After preprocessing the jumps and doing the Euler tour, we now have all the tools we need to get the LCA of two nodes in $O(\log n)$ time. First of all, we will start with the two nodes u and v for which we want to find the LCA of. We check whether or not u is an ancestor of v and vice versa, stopping if either of this is the case. Then, we loop i from $\lceil \log(n) \rceil$ to 0, going up 2^i nodes if and only if the 2^i-th parent is not an ancestor of v. The LCA of the two nodes will be the parent of the resulting node u.

3.4.1 Code

This code assumes the DFS traversal has already been done and that tin and tout are both processed.

```
let n = number of nodes in the tree
adj = adjacency list of tree
let tin = integer array of size n // time in for each node
let tout = integer array of size n // time out for each node

let k = ceil(log(n))
let lift = 2d array of size n and size k + 1

boolean is_ancestor(int u, int v) {
        return tin[u] <= tin[v] and tout[u] >= tout[v]
}

int lca(int u, int v) {
        if (is_ancestor(u, v)) return u
        if (is_ancestor(v, u)) return v
        for i = k to 0:
                if (not is_ancestor(lift[u][i], v))
                        u = lift[u][i]
        return lift[u][0]
}
```

3.4.2 Intuition

Why does this work? Let's look at this section of code specifically:

```
for i = k to 0:
        if (not is_ancestor(lift[u][i], v))
                u = lift[u][i]
return lift[u][0]
```

This code is finding the highest parent of u that is **not** an ancestor of v. If lift[u][i] is an ancestor of v, then we don't go up 2^i nodes, so this code will keep going "up" the tree, but stops right before it gets to the node which is an ancestor of v. Now, let's call l the parent of the resulting node u. It should make sense that l is the lowest node which has v as a descendant (since our code found the highest node that isn't an ancestor of v) which also has the original node u as a descendant (since we have only gone "up" from the original node u, and this entire path consists of all the ancestors of u).

4 Dynamic Programming on Trees

Prerequisite: Basic knowledge of dynamic programming

Another important property in that in any rooted tree, every node will have at one parent except the root node, which has none. We can use this fact in order to solve problems using dynamic programming, as we will usually only have to consider the state of the parent node in our DP transitions. This technique is called **dynamic programming on trees**.

As an example of this technique in action, we will take a look at S5 from the 2022 Senior Canadian Computing Competition: Good Influencers.

4.1 Example Problem

Summary of the problem statement: Given a tree of n nodes, $n-1$ edges and an array of size n with values "Y" or "N" referring to the nodes. You are also given an integer array c of size n, referring to the cost of each node. You can choose a node with value "Y" and pay the respective cost to turn all the node's neighbors to value "Y". Find the minimum cost to turn all nodes to "Y".

4.2 DP States and Transitions

Let us create a 2-dimensional array and let the first dimension represent i, being the i-th node in the graph and for each node, we will have 4 different states. Let's also define *activating* a node to be the operation of paying the cost of that node in order to turn all the node's neighbors to 'Y'.

Here are our DP states:

dp[i][0] - node i is currently 'N', and we don't have to activate this node. (We can still activate it if it's optimal to do so)

dp[i][1] - node i is currently 'N', and we need to activate this node.

dp[i][2] - node i is currently 'Y', and we don't have to activate this node.

dp[i][3] - node i is currently 'Y', and we need to activate this node.

Now, let's take a look at our DP transitions.

dp[i][0] - Let's first look at the case where we don't activate this node. We can assume that the parent node will not influence this node, so this node has to be turned to 'Y' by one of its child nodes. We will then choose the cheapest option for one of the child nodes to be activated. Therefore, we have the following:

$$\text{dp[i][0]} = \min\left(\min_{\forall \text{child} \in i}(\text{dp[child][1 or 3]} - \text{dp[child][0 or 2]} + \sum \text{dp[child][0 or 2]}), \text{dp[i][1]}\right)$$

Similarly, we can deduce transitions for the other DP states using the same idea:

$$\text{dp[i][1]} = \min(\text{dp[child][1 or 3]} + \text{c[i]} + \sum \text{dp[child][2]})$$

$$\text{dp[i][2]} = \min(\sum \text{dp[child][0 or 2]}, \text{dp[i][3]})$$

$$\text{dp[i][3]} = \min(\text{c[i]} + \sum \text{dp[child][2]})$$

Note: This can also be done with only 3 states for each node by combining dp[i][1] and dp[i][3].

4.3 General Approaches to Tree DP Problems

Since in a tree, each node will only have 1 parent (except the root, which will have 0), we can usually break up our DP states and transitions into scenarios in which:

The current node has some influence on the parent node, or:

The current node has no influence on the parent node.

This is particularly useful in cases like the problem above, where a node influences all its neighbors.

5 Sources and Further Reading

Introduction

[1] `https://en.wikipedia.org/wiki/Tree_(graph_theory)`

[2] `https://en.wikipedia.org/wiki/Path_(graph_theory)`

Minimum Spanning Trees

[3] `https://stackoverflow.com/questions/10414043/is-minimum-spanning-tree-afraid-of-negative-weights`

Binary Lifting

[4] `https://cp-algorithms.com/graph/lca_binary_lifting.html`

Other

[5] `https://cses.fi/book/book.pdf`