Average Rating: 4.95 (487 votes)
Premium

Solution

For this article, we'll assume that you've already learned the basics of graph theory, and how to perform a simple depth-first search. Most text books on algorithms or discrete math have a section on it.

There are a *lot* of different ways to solve this problem, and it's impossible to cover them all. For that reason, I've tried to split the ways into a few general approaches, and then for each I discuss a few additional variants. Some of the ideas that were covered in Approach 1 might apply to Approach 2 as well.

Approach 1: Graph Theory + Iterative Depth-First Search

Intuition

Note that this same approach also works with recursive depth-first search and iterative breadth-first search. We'll look at these briefly in the Algorithm section.

Recall that a graph, `G`, **is a tree iff** the following two conditions are met:

1. `G` is fully connected. In other words, for every pair of nodes in `G`, there is a path between them.
2. `G` contains no cycles. In other words, there is exactly *one* path between each pair of nodes in `G`.

Depth-first search is a classic graph-traversal algorithm that can be used to check for both of these conditions:

1. `G` is fully connected if, and only if, we started a depth-first search from a single source and discovered *all* nodes in `G` during it.
2. `G` contains no cycles if, and only if, the depth-first search never goes back to an already discovered node. We need to be careful though not to count trivial cycles of the form `A → B → A` that occur with most implementations of undirected edges.

Depth-first search requires being able to look up the *adjacent* (immediate neighbours) of a given node. Like many graph interview problems though, the input format we're given doesn't allow us to quickly get the neighbours of a node. Therefore, our first step is to convert the input into an **adjacency list**. Recall that an **adjacency list** is where we have a list of sub-lists, where each sub-list is the list of the immediate neighbours for the i'th node.

Java

```java
// Create a new list of lists.
List<List<Integer>> adjacencyList = new ArrayList<>();
// Initialise an empty list for each node.
for (int i = 0; i < n; i++) {
    adjacencyList.add(new ArrayList<>());
}
// Go through the edge list, populating the adjacency list.
for (int[] edge : edges) {
    adjacencyList.get(edge[0]).add(edge[1]);
    adjacencyList.get(edge[1]).add(edge[0]);
}
```

Before we move onto actually carrying out the depth-first search, let's quickly reassure ourselves that an **adjacency list** was the best graph representation for this problem. The other 2 choices would have been an **adjacency matrix** or a **linked representation**.

- An **adjacency matrix** would be an acceptable, although not ideal, representation for this problem. Often, we'd only use an adjacency matrix if we *know* that the number of edges is substantially higher than the number of nodes. We have no reason to believe that is the case here. Approach 2 will also provide some useful insight into this.
- A **linked representation**, where you make actual nodes objects, would be an overly complicated representation and could suggest to an interviewer that you have a limited understanding of adjacency lists and adjacency matrices. They are not commonly used in interview questions.

Anyway, let's get started on the **depth-first search**. Recall that most depth-first searches follow a template like the one below for **iterative depth-first search**. Note that this *doesn't yet solve the problem of determining whether or not the input graph is a tree*—we're simply using it as a step towards building up a solution.

Java

```java
// Use a stack to keep track of unexplored nodes.
Stack<Integer> stack = new Stack<>();
stack.push(0);
// Use a set to keep track of already seen nodes to
// avoid infinite looping.
Set<Integer> seen = new HashSet<>();
seen.add(0);

// While there are nodes remaining on the stack...
while (!stack.isEmpty()) {
    int node = stack.pop(); // Take one off to visit.
    // Check for unseen neighbours of this node:
    for (int neighbour : adjacencyList.get(node)) {
        if (seen.contains(neighbour)) {
            continue; // Already seen this node.
        }
        // Otherwise, put this neighbour onto stack
        // and record that it has been seen.
        stack.push(neighbour);
        seen.add(neighbour);
    }
}
```

If you're now really confused, we recommend checking out the chapter on DFS in the Stack and Queue Explore Card. The common templates for depth-first search are given and explained in the card.

Let's now figure out how we can modify the basic depth-first search template to do the two checks we need.

The first check is straightforward. **If the graph is fully connected**, then every node must have been seen. This means that all nodes must be in the `seen` set at the end. Because a set removes duplicates, and the only values going into it were valid node numbers, then we know that the graph was fully connected *if, and only if,* the `seen` set contains `n` values at the end.

JavaScript
Autocomplete

```javascript
/**
 * @param {number} n
 * @param {number[][]} edges
 * @return {boolean}
 */
var validTree = function(n, edges) {

};
```

Console Contribute

Java

Copy

```java
// Return true iff the depth first search discovered ALL nodes.
return seen.size() == n;
```

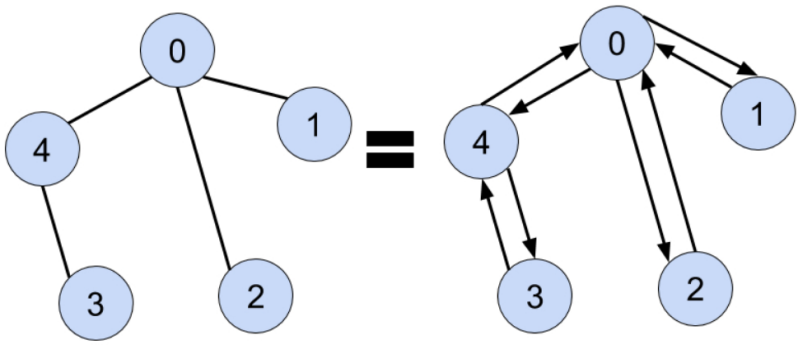
For the second check, you might be thinking: can't we just modify the above algorithm to return `false` when a neighbour *is* in visited? i.e.

Java

Copy

```java
if (seen.contains(neighbour)) {
    return false;
}
```

This, however, would only work on a *directed* graph. On an *undirected* graph, like the one we're working with here, trivial "cycles" will be detected. For example, if there's an undirected edge between node `A` and node `B`, a detected cycle will include `A → B → A`. This is because an undirected edge is actually *2* edges in the adjacency list, and so forms a trivial cycle.



There are several strategies of detecting whether or not an *undirected* graph contains cycles, while excluding the trivial cycles. Most rely on the idea that a depth-first search should only go along each edge *once*, and therefore only in *one* direction. This means that when we go along an edge, we should do something to ensure that we don't then later go back along it in the opposite direction. Here are a couple of ways of achieving this.

The first strategy is to simply delete the opposite direction edges from the adjacency list. In other words, when we follow an edge `A → B`, we should lookup `B`s adjacency list and delete `A` from it, effectively removing the opposite edge of `B → A`.

Java

Copy

```java
// While there are nodes remaining on the stack...
while (!stack.isEmpty()) {
    int node = stack.pop(); // Take one off to visit.
    // Check for unseen neighbours of this node:
    for (int neighbour : adjacencyList.get(node)) {
        // Check if we've already seen this node.
        if (seen.contains(neighbour)) {
            return false;
        }
        // Otherwise, put this neighbour onto stack
        // and record that it has been seen.
        stack.push(neighbour);
        seen.add(neighbour);
        // Remove the link that goes in the opposite direction.
        adjacencyList.get(neighbour).remove(node);
    }
}
```

The second strategy is, instead of using a `seen` set, to use a `seen` *map* that also keeps track of the "parent" node that we got to a node from. We'll call this map `parent`. Then, when we iterate through the neighbours of a node, we ignore the "parent" node as otherwise it'll be detected as a trivial cycle (and we know that the parent node has already been visited by this point anyway). The starting node (`0` in this implementation) has no "parent", so put it as `-1`.

At first, it's a little more difficult to understand why this strategy even works. A good way to think about it is to remember that like the first approach, we just want to avoid going along edges we've already been on (in the opposite direction). The parent links prevent that, as each node is only entered for exploration *once*. So, imagine you're walking through a maze, with the condition that you're not allowed to go back along *any* path you've already been on. If you still somehow end up somewhere you were previously, there must have been a cycle!

Java

Copy

```java
// Use a map to keep track of how we got into each node.
Map<Integer, Integer> parent = new HashMap<>();
parent.put(0, -1);

// While there are nodes remaining on the stack...
while (!stack.isEmpty()) {
    int node = stack.pop(); // Take one off to visit.
    // Check for unseen neighbours of this node:
    for (int neighbour : adjacencyList.get(node)) {
        // Don't look at the trivial cycle.
        if (parent.get(node) == neighbour) {
            continue;
        }
        // Check if we've already seen this node.
        if (parent.containsKey(neighbour)) {
            return false; // There must be a cycle.
        }
        // Otherwise, put this neighbour onto stack
        // and record that it has been seen.
        stack.push(neighbour);
        parent.put(neighbour, node);
    }
}
```

The best strategy for this problem is probably the second one, because it doesn't require modifying the adjacency list. For more complex graph problems, the first strategy can be useful though.

Algorithm

In the example, we used an *iterative depth-first search*. Here is the complete code for this.

Java Python Copy

```java
public boolean validTree(int n, int[][] edges) {

    List<List<Integer>> adjacencyList = new ArrayList<>();
    for (int i = 0; i < n; i++) {
        adjacencyList.add(new ArrayList<>());
    }
    for (int[] edge : edges) {
        adjacencyList.get(edge[0]).add(edge[1]);
        adjacencyList.get(edge[1]).add(edge[0]);
    }

    Map<Integer, Integer> parent = new HashMap<>();
    parent.put(0, -1);
    Stack<Integer> stack = new Stack<>();
    stack.push(0);

    while (!stack.isEmpty()) {
        int node = stack.pop();
        for (int neighbour : adjacencyList.get(node)) {
            if (parent.get(node) == neighbour) {
                continue;
            }
            if (parent.containsKey(neighbour)) {
                return false;
            }
            stack.push(neighbour);
            parent.put(neighbour, node);
```

Alternatively, you could use recursion, as long as you're fairly confident with it. The recursive approach is more elegant, but is considered *inferior* to the iterative version in some programming languages, such as Python. This is because the space used by run-time stacks vary between programming languages.

On the plus side, we can use a simple `seen` set and just pass a `parent` parameter. This makes the code a bit simpler!

Java Python Copy

```java
class Solution {

    private List<List<Integer>> adjacencyList = new ArrayList<>();
    private Set<Integer> seen = new HashSet<>();


    public boolean validTree(int n, int[][] edges) {

        if (edges.length != n - 1) return false;

        for (int i = 0; i < n; i++) {
            adjacencyList.add(new ArrayList<>());
        }
        for (int[] edge : edges) {
            adjacencyList.get(edge[0]).add(edge[1]);
            adjacencyList.get(edge[1]).add(edge[0]);
        }

        // We return true iff no cycles were detected,
        // AND the entire graph has been reached.
        return dfs(0, -1) && seen.size() == n;
    }

    public boolean dfs(int node, int parent) {
        if (seen.contains(node)) return false;
        seen.add(node);
        for (int neighbour : adjacencyList.get(node)) {
```

Yet another variant is to use *iterative breadth-first search*. Recall that *breadth-first search* and *depth-first search* are almost the same algorithm, just with a different data structure.

Java Python Copy

```java
public boolean validTree(int n, int[][] edges) {

    List<List<Integer>> adjacencyList = new ArrayList<>();
    for (int i = 0; i < n; i++) {
        adjacencyList.add(new ArrayList<>());
    }
    for (int[] edge : edges) {
        adjacencyList.get(edge[0]).add(edge[1]);
        adjacencyList.get(edge[1]).add(edge[0]);
    }

    Map<Integer, Integer> parent = new HashMap<>();
    parent.put(0, -1);
    Queue<Integer> queue = new LinkedList<>();
    queue.offer(0);

    while (!queue.isEmpty()) {
        int node = queue.poll();
        for (int neighbour : adjacencyList.get(node)) {
            if (parent.get(node) == neighbour) {
                continue;
            }
            if (parent.containsKey(neighbour)) {
                return false;
            }
            queue.offer(neighbour);
            parent.put(neighbour, node);
```

Complexity Analysis

Let E be the number of edges, and N be the number of nodes.

- Time Complexity : $O(N + E)$.

  Creating the adjacency list requires initialising a list of length N, with a cost of $O(N)$, and then iterating over and inserting E edges, for a cost of $O(E)$. This gives us $O(E) + O(N) = O(N + E)$.

  Each node is added to the data structure *once*. This means that the outer loop will run N times. For each of the N nodes, its adjacent edges is iterated over once. In total, this means that all E edges are iterated over once by the inner loop. This, therefore, gives a total time complexity of $O(N + E)$.

  Because both parts are the same, we get a final time complexity of $O(N + E)$.

- Space Complexity : $O(N + E)$.

  The adjacency list is a list of length N, with inner lists with lengths that add to a total of E. This gives a total of $O(N + E)$ space.

  In the worst case, the stack/ queue will have all N nodes on it at the same time, giving a total of $O(N)$ space.

  In total, this gives us $O(E + N)$ space.

Approach 2: Advanced Graph Theory + Iterative Depth-First Search

Intuition

Depending on how much graph theory you know, there's a better definition for determining whether or not a given graph is a tree.

For the graph to be a valid tree, it must have *exactly* `n - 1` edges. Any less, and it can't possibly be fully connected. Any more, and it *has* to contain cycles. Additionally, if the graph is fully connected *and* contains exactly `n - 1` edges, it can't *possibly* contain a cycle, and therefore must be a tree!

These facts are fairly straightforward to prove. We won't go into why they are true here, but if you're not familiar with these facts, then we recommend reading up on graph theory. It is *very important* to be confident with graph theory in-order to pass the interviews at a top tech company.

Going by this definition, our algorithm needs to do the following:

1. Check whether or not there are `n - 1` edges. If there's not, then return `false`.
2. Check whether or not the graph is *fully connected*. Return `true` if it is, `false` if otherwise.

Recall that the most complicated part of Approach 1 was in checking whether or not the graph contained cycles. This was because in an undirected graph, we needed to be careful of *trivial cycles*. Checking whether or not a graph is fully connected is straightforward—we simply checked if all nodes were reachable from a search starting at a single node.

Like before, we can can check for connectivity using recursive depth-first search, iterative depth-first search, or iterative breadth-first search. We still need to use a `seen` set to prevent the algorithm getting caught in an infinite loop if there are indeed cycles (and to prevent looping on the trivial cycles).

Algorithm

Again, we've provided code for all three variants.

Iterative Depth-First Search.

Java | Python | Copy

```java
public boolean validTree(int n, int[][] edges) {

    if (edges.length != n - 1) return false;

    // Make the adjacency list.
    List<List<Integer>> adjacencyList = new ArrayList<>();
    for (int i = 0; i < n; i++) {
        adjacencyList.add(new ArrayList<>());
    }
    for (int[] edge : edges) {
        adjacencyList.get(edge[0]).add(edge[1]);
        adjacencyList.get(edge[1]).add(edge[0]);
    }

    Stack<Integer> stack = new Stack<>();
    Set<Integer> seen = new HashSet<>();
    stack.push(0);
    seen.add(0);

    while (!stack.isEmpty()) {
        int node = stack.pop();
        for (int neighbour : adjacencyList.get(node)) {
            if (seen.contains(neighbour)) continue;
            seen.add(neighbour);
            stack.push(neighbour);
        }
    }
```

Recursive Depth-First Search.

Java | Python | Copy

```java
class Solution {

    private List<List<Integer>> adjacencyList = new ArrayList<>();
    private Set<Integer> seen = new HashSet<>();


    public boolean validTree(int n, int[][] edges) {

        if (edges.length != n - 1) return false;

        // Make the adjacency list.
        for (int i = 0; i < n; i++) {
            adjacencyList.add(new ArrayList<>());
        }
        for (int[] edge : edges) {
            adjacencyList.get(edge[0]).add(edge[1]);
            adjacencyList.get(edge[1]).add(edge[0]);
        }

        // Carry out depth first search.
        dfs(0);
        // Inspect result and return the verdict.
        return seen.size() == n;
```

```
    }

    public void dfs(int node) {
        if (seen.contains(node)) return;
```

Iterative Breadth-First Search.

Java | Python

Copy

```java
public boolean validTree(int n, int[][] edges) {

    if (edges.length != n - 1) return false;

    // Make the adjacency list.
    List<List<Integer>> adjacencyList = new ArrayList<>();
    for (int i = 0; i < n; i++) {
        adjacencyList.add(new ArrayList<>());
    }
    for (int[] edge : edges) {
        adjacencyList.get(edge[0]).add(edge[1]);
        adjacencyList.get(edge[1]).add(edge[0]);
    }

    Queue<Integer> queue = new LinkedList<>();
    Set<Integer> seen = new HashSet<>();
    queue.offer(0);
    seen.add(0);

    while (!queue.isEmpty()) {
        int node = queue.poll();
        for (int neighbour : adjacencyList.get(node)) {
            if (seen.contains(neighbour)) continue;
            seen.add(neighbour);
            queue.offer(neighbour);
        }
    }
```

Complexity Analysis

Let E be the number of edges, and N be the number of nodes.

- Time Complexity : $O(N)$.

  When $E \neq N - 1$, we simply return `false`. Therefore, the worst case is when $E = N - 1$. Because E is proportional to N, we'll say $E = N$ to simplify the analysis.

  As said above, creating an adjacency list has a time complexity of $O(N + E)$. Because E is now bounded by N, we can reduce this slightly to $O(N + N) = O(N)$.

  The iterative breadth-first search and depth-first search are almost identical. Each node is put onto the queue/stack *once*, ensured by the `seen` set. Therefore, the inner "neighbour" loop runs once for each node. Across all nodes, the number of cycles this loop does is the same as the number of edges, which is simply N. Therefore, these two algorithms have a time complexity of $O(N)$.

  The recursive depth-first search's "neighbour" loop runs only once for each node. Therefore, in total, the function is called once for each edge. So it is called $E = N times, and N of those times, it actually enters the "neighbour" loop. Collectively, the total number of iterations of the "neighbour" loop is $E = N$. So we get $O(N)$, as these all simply add.

- Space Complexity : $O(N)$.

  Previously, we determined that the adjacency list took $O(E + N)$ space. We now know this is simply $O(N)$.

  In the worst case, the search algorithms will require an additional $O(N)$ space; this is if *all* nodes were on the stack/queue at the same time.

  So again we get a total of $O(N)$.

Approach 3: Advanced Graph Theory + Union Find

Intuition

In Approach 2, we used this definition for a tree:

> For the graph to be a valid tree, it must have *exactly* `n - 1` edges. Any less, and it can't possibly be fully connected. Any more, and it *has* to contain cycles. Additionally, if the graph is fully connected *and* contains exactly `n - 1` edges, it can't *possibly* contain a cycle, and therefore must be a tree!

This definition simplified the problem down to checking whether or not the graph is fully connected. If it is, and if it contains `n - 1` edges, then we know it's a tree. In the previous approaches, we used graph search algorithms to check whether or not all nodes were reachable, starting from a single source node.

Another way we could approach the problem is by considering each connected component to be a set of nodes. When an edge is put between two separate connected components, they are merged into a single connected component. We'll use the example `n = 6` and `edges = [(0, 5), (4, 0), (1, 2), (4, 5), (3, 2)]`. Before we look at the edges, we have `6` sets.

$$\{0\} \quad \{1\} \quad \{2\} \quad \{3\} \quad \{4\} \quad \{5\}$$

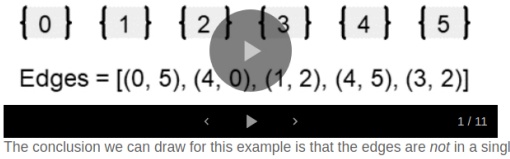
We can then go through the list of edges, and merge sets together. For example, because the first edge is `(0, 5)`, we merge the sets with `0` and `5`. This means that we now have five connected components.

$$\{0, 5\} \quad \{1\} \quad \{2\} \quad \{3\} \quad \{4\}$$

The next edge is `(4, 0)`. Therefore, we merge the sets `{0, 5}` and `{4}` (remember that because `0` and `5` are connected, and `4` and `0` are connected, this means that `4` and `5` must also be connected).

$$\{0, 5, 4\} \quad \{1\} \quad \{2\} \quad \{3\}$$

We can continue this same process until we've gone through all the edges. Here is an animation showing this.



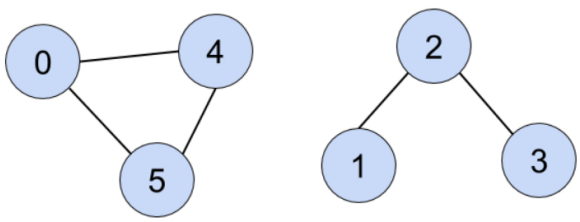
The conclusion we can draw for this example is that the edges are *not* in a single connected component, and therefore must contain a cycle. The algorithm should return `false`.

And here is another example where the edges *do* form a single connected component.

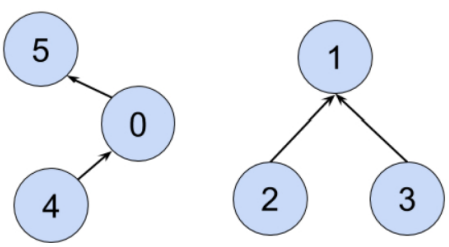
Did you notice that in the second animation, every edge resulted in a merge operation, but in the first animation, some didn't? This was because of cycles in the graph. Each time there was no merge, it was because we were adding an edge between two nodes that were already connected via a path. This means there is now an additional path between them—which is the definition of a cycle. Therefore, *as soon as this happens*, we can terminate the algorithm and return `false`.

Now, how should we implement this? Well, we could create a list of `Set`s and then carry out the algorithm as it was done in the animation. There is, however, a better way; a really clever algorithm we call **union find**. I'll give a brief introduction to the algorithm here, however, if you aren't familiar with it, I strongly recommend reading up on it out of a good algorithm textbook. We don't *yet* have an explore card on union find (at the time of writing this!). Union find is a very useful algorithm that can be used to solve many graph problems.

Union find represents each of the sets as a directed tree, with the edges pointing *towards* the root node. For example, consider this graph from our first example above (the one that was *not* a valid tree).

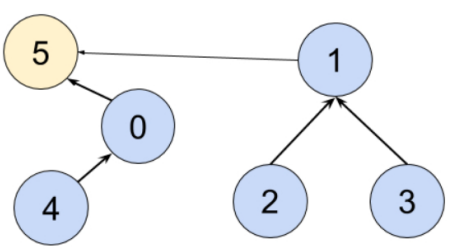


One way its connected components could be represented by union find is as follows:

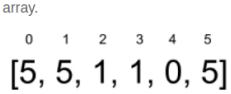


Union find is a **data structure** with 3 methods; `makeset(A)`, `find(A)` and `union(A, B)`.

- The `makeset(A)` method is the simplest. It creates a new size-1 set containing just element `A`.
- The `find(A)` method starts at `A`, and traces parent links up until it gets to `A`'s tree root. It then returns the tree root ID. Two nodes that are in the same connected component have the same root. If they're in different connected components, then they will have different roots. For the above example, `find(0)`, `find(4)`, and `find(5)` will all return `5`. Whereas `find(1)`, `find(2)` and `find(3)` will all return `1`. This method can be used to check whether or not 2 elements are in the same connected component, and is also used by the `union(A, B)` method, as we're about to see.
- The `union(A, B)` method works by finding the root for `A` and the root for `B`, using the `find(...)` operation. It then sets `B`s parent to be `A`, thus combining the two trees into one. For example, if we add the edge `(4, 3)` to the example above, the algorithm will find that the root of `4` is `5`, and the root of `3` is `1`, and merge those subtrees. Once this is done, all of the nodes have the same root of `5`, and therefore we know they all belong to the same connected component.



We don't need to use linked nodes to represent this structure; we can simply maintain an array of parent pointers. For example, here's how the above tree is represented as an array.



Notice how `5` simply points to itself, as it's the root. The `find(...)` operation works its way up parent links until it finds a node that points back to itself.

This algorithm might not seem very efficient, after all, the `find(...)` operation could be $O(n)$ in the worst case. However, there are two straightforward optimizations we can apply that bring the amortized time close to $O(1)$ for both `union(...)` and `find(...)`.

1. Tracking the sizes of each set; this helps to ensure the tree depth is minimised, as we can ensure the smaller set is attached onto the larger set, and not the other way around. The modifications for this are in the `union(...)` method.
2. When doing a `find(...)`, keeping track of all the nodes along the path so that afterwards we can make each point *directly* at the root, so that next time any of those nodes

are searched for, it is $O(1)$. The modifications for this are all in the `find(...)` method.

Variants of these also exist, that result in the same overall time complexity.

Algorithm

Firstly, here's the code *without* the optimizations. Below, I've also included the code with the optimizations. If you're new to union find, then I recommend reading the code without optimizations first, as it's a lot easier to understand!

Java Python Copy

```java
class UnionFind {

    private int[] parent;

    // For efficiency, we aren't using makeset, but instead initialising
    // all the sets at the same time in the constructor.
    public UnionFind(int n) {
        parent = new int[n];
        for (int node = 0; node < n; node++) {
            parent[node] = node;
        }
    }

    // The find method, without any optimizations. It traces up the parent
    // links until it finds the root node for A, and returns that root.
    public int find(int A) {
        while (parent[A] != A) {
            A = parent[A];
        }
        return A;
    }

    // The union method, without any optimizations. It returns True if a
    // merge happened, False if otherwise.
    public boolean union(int A, int B) {
        // Find the roots for A and B.
        int rootA = find(A);
```

These are the solutions using the optimizations *path compression* and *union by size*.

Java Python Copy

```java
class UnionFind {

    private int[] parent;
    private int[] size; // We use this to keep track of the size of each set.

    // For efficiency, we aren't using makeset, but instead initialising
    // all the sets at the same time in the constructor.
    public UnionFind(int n) {
        parent = new int[n];
        size = new int[n];
        for (int node = 0; node < n; node++) {
            parent[node] = node;
            size[node] = 1;
        }
    }

    // The find method, with path compression. There are ways of implementing
    // this elegantly with recursion, but the iterative version is easier for
    // most people to understand!
    public int find(int A) {
        // Step 1: Find the root.
        int root = A;
        while (parent[root] != root) {
            root = parent[root];
        }
        // Step 2: Do a second traversal, this time setting each node to point
        // directly at A as we go.
```

Complexity Analysis

Let E be the number of edges, and N be the number of nodes.

$\alpha(N)$ is the Inverse Ackermann Function.

- Time Complexity : $O(N \cdot \alpha(N))$.

  When $E \neq N - 1$, we simply return `false`. Therefore, the worst case is when $E = N - 1$. Because E is proportional to N, we'll say $E = N$ to simplify the analysis.

  We are putting each of the N edges into the `UnionFind` data structure, using the `union(...)` method. The `union(...)` method itself has no loops or recursion, so the entire cost of calling it is dependent on the cost of the `find(...)` method that it calls.

  `find(...)` 's cost is dependent on how far the node it was searching for is from the root. Using the naïve implementation of union find, this depth could be N. If this was the case for all of the calls, we'd have a final cost of $O(N^2)$.

  However, remember those optimizations we did? Those keep the tree depths very shallow. It turns out that `find(...)` *amortizes* to $O(\alpha(N))$, where α is the Inverse Ackermann Function. The incredible thing about this function is that it grows *so slowly* that N will *never* go higher than 4 in the universe as we know it! So while in "practice" it is effectively $O(1)$, in "theory" it is not.

  Actually *proving* this upper bound on the depth is a very advanced proof, which I'd certainly *hope* you'd never need to do in an interview! If you're interested though, I recommend looking in a good algorithm's text book or paper.

  Anyway, this gives us a total of $N \cdot O(\alpha(N)) = O(N \cdot \alpha(N))$.

- Space Complexity : $O(N)$.

  The UnionFind data structure requires $O(N)$ space to the store the arrays it uses.

So, why is this better than Approach 2?

Complexity analysis ignores constants. For example, $O(10 \cdot N) = O(N)$. Even $O(10000 \cdot N) = O(N)$. Sometimes the constants we're ignoring in the analysis are still having a big impact on the run-time in practice.

Approach 2 had a lot of overhead in needing to create an adjacency list with the edges before it could even begin the depth-first search. This is all treated as a constant, as it ultimately had the same time complexity as the depth-first search itself.

ultimately had the same time complexity as the depth-first search itself.

Approach 3 doesn't need to change the input format, it can just get straight to determining whether or not there is a cycle. Additionally the bit that stops it being constant, the $\alpha(N)$, will *never* have a value larger than 4. So in practice, it behaves as a constant too—and a far smaller one at that!

When weighing up the pros and cons of different algorithms for solving problems, it's best to treat union find's operations as $O(1)$ to get a fair and accurate comparison.

Report Article Issue

Comments: 106

Best | Most Votes | Newest to Oldest | Oldest to Newest

Type comment here... (Markdown is supported)

Post

ZoroDuncan ★ 244 May 16, 2020 9:55 PM

LeetCode should pay the author, seriously.

165 Show 1 reply Reply

Takkes ★ 300 March 26, 2020 1:47 PM

I'm almost sad this has so few comments - the article is really great! :D

254 Show 5 replies Reply

Dirk41 ★ 949 Last Edit: April 13, 2020 8:21 AM

At first, I feel the paper will be too long,won't read; then after read it, I really learnt a lot !!!!! Love this feeling.

87 Show 1 reply Reply

mangrer ★ 1056 Last Edit: May 14, 2020 8:56 AM

I wish, I see all solutions were written like this one.

40 Reply

sadyojat ★ 43 March 28, 2020 5:12 PM

@Hai_dee - great article, thanks for this.

32 Show 2 replies Reply

azimbabu ★ 247 April 11, 2020 12:16 AM

This is one of the best articles. Thanks @Hai_dee . Also as you recommended in approach #2, it would be great to have some articles on graph theory at Leetcode.

21 Show 3 replies Reply

dlrowolleh ★ 18 April 7, 2020 10:34 PM

One of the best solution articles I've seen on leetcode.

18 Reply

ASH1990 ★ 22 March 17, 2020 12:11 AM

Really great article..

17 Reply

rvpk ★ 207 March 28, 2020 11:14 PM

Thanks for the detailed explanation!

9 Reply

8emnero8 ★ 7 March 29, 2020 12:23 PM

Great article!

7 Reply

1 2 3 4 5 ... 11

Problems | Pick One | Prev | 4/56 | Next | Run Code | Submit