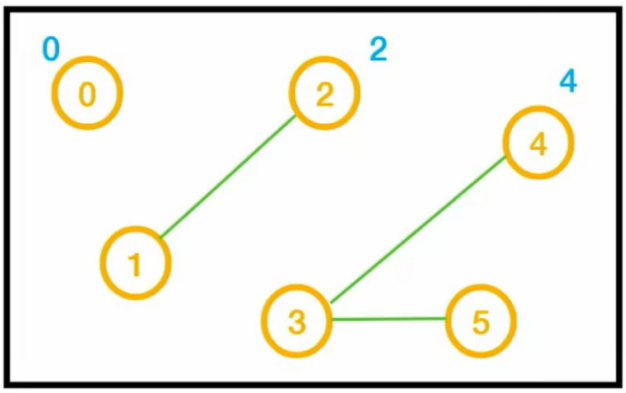
Average Rating: 4.68 (34 votes)

Premium

Video Solution



21:04

Solution Article

Approach 1: Depth-First Search (DFS)

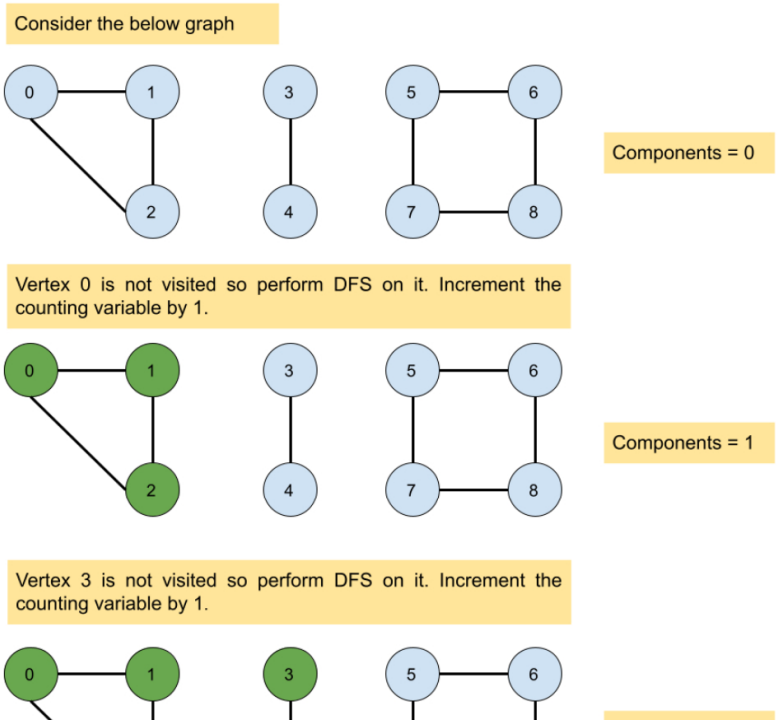
Intuition

> If you're not familiar with DFS, check out our Explore Card.

In an undirected graph, a connected component is a subgraph in which each pair of vertices is connected via a path. So essentially, all vertices in a connected component are *reachable* from one another.

Let's see how we can use DFS to solve the problem. If we run DFS, starting from a particular vertex, it will continue to visit the vertices depth-wise until there are no more adjacent vertices left to visit. Thus, it will visit all of the vertices within the connected component that contains the starting vertex. Each time we finish exploring a connected component, we can find another vertex that has *not been visited yet*, and start a new DFS from there. The number of times we start a new DFS will be the number of connected components.

Here is an example illustrating this approach.



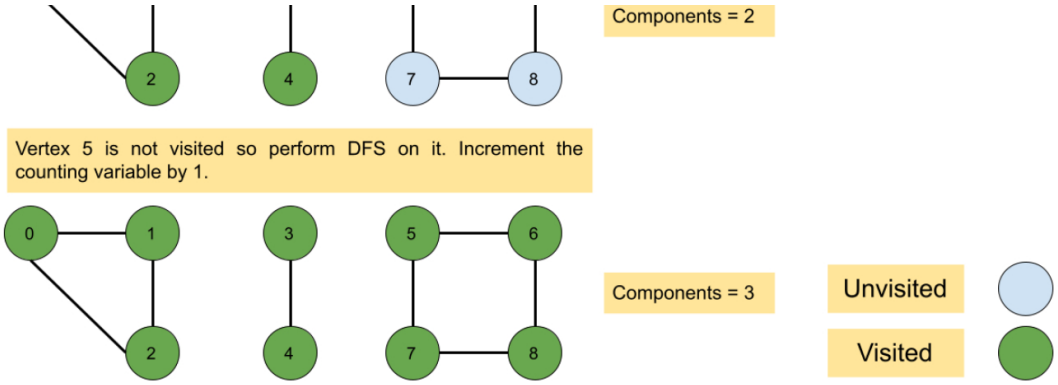


Figure 1. An example demonstrating the DFS approach.

Algorithm

1. Create an adjacency list such that `adj[v]` contains all the adjacent vertices of vertex `v`.
2. Initialize a hashmap or array, `visited`, to track the visited vertices.
3. Define a `counter` variable and initialize it to zero.
4. Iterate over each vertex in `edges`, and if the vertex is not already in `visited`, start a DFS from it. Add every vertex visited during the DFS to `visited`.
5. Every time a new DFS starts, increment the `counter` variable by one.
6. At the end, the `counter` variable will contain the number of connected components in the undirected graph.

C++ | Java — Copy

```java
class Solution {

    private void dfs(List<Integer>[] adjList, int[] visited, int startNode) {
        visited[startNode] = 1;

        for (int i = 0; i < adjList[startNode].size(); i++) {
            if (visited[adjList[startNode].get(i)] == 0) {
                dfs(adjList, visited, adjList[startNode].get(i));
            }
        }
    }

    public int countComponents(int n, int[][] edges) {
        int components = 0;
        int[] visited = new int[n];

        List<Integer>[] adjList = new ArrayList[n];
        for (int i = 0; i < n; i++) {
            adjList[i] = new ArrayList<Integer>();
        }

        for (int i = 0; i < edges.length; i++) {
            adjList[edges[i][0]].add(edges[i][1]);
            adjList[edges[i][1]].add(edges[i][0]);
        }

        for (int i = 0; i < n; i++) {
            if (visited[i] == 0) {
```

Complexity Analysis

Here E = Number of edges, V = Number of vertices.

- Time complexity: $O(E + V)$.

  Building the adjacency list will take $O(E)$ operations, as we iterate over the list of edges once, and insert each edge into two lists.

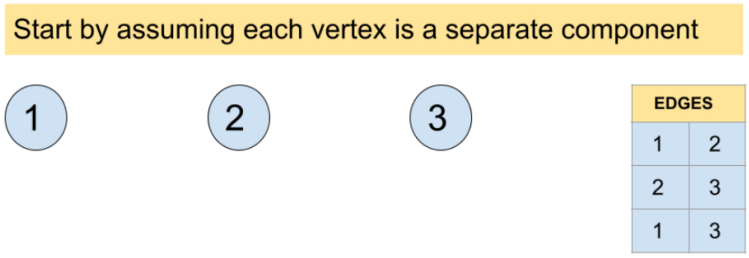
  During the DFS traversal, each vertex will only be visited once. This is because we mark each vertex as visited as soon as we see it, and then we only visit vertices that are not marked as visited. In addition, when we iterate over the edge list of each vertex, we look at each edge once. This has a total cost of $O(E + V)$.

- Space complexity: $O(E + V)$.

  Building the adjacency list will take $O(E)$ space. To keep track of visited vertices, an array of size $O(V)$ is required. Also, the run-time stack for DFS will use $O(V)$ space.

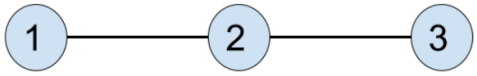
Approach 2: Disjoint Set Union (DSU)

Imagine we have a graph with `N` vertices and `0` edges. The number of connected components will be `N` in that graph.



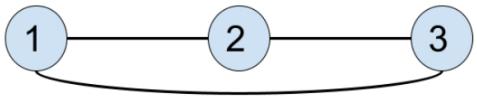
Let's now add the edge from vertex 1 to vertex 2. This will decrease the number of components by 1. This is because vertices 1 and 2 are now in the same component.

When we then add the edge from vertex 2 to vertex 3, the number of components will decrease by 1 again.



EDGES	
1	2
2	3
1	3

However, this pattern will not continue when we add the edge from vertex 1 to vertex 3. The number of components will not change because vertices 1, 2, and 3 are already in the same component.



EDGES	
1	2
2	3
1	3

The above observation is the main intuition behind the DSU approach.

Algorithm

1. Initialize a variable `count` with the number of vertices in the input.
2. Traverse all of the edges one by one, performing the union-find method `combine` on each edge. If the endpoints are already in the same set, then keep traversing. If they are not, then decrement `count` by 1.
3. After traversing all of the `edges`, the variable `count` will contain the number of components in the graph.

C++ Java Copy

```java
public class Solution {

    private int find(int[] representative, int vertex) {
        if (vertex == representative[vertex]) {
            return vertex;
        }

        return representative[vertex] = find(representative, representative[vertex]);
    }

    private int combine(int[] representative, int[] size, int vertex1, int vertex2) {
        vertex1 = find(representative, vertex1);
        vertex2 = find(representative, vertex2);

        if (vertex1 == vertex2) {
            return 0;
        } else {
            if (size[vertex1] > size[vertex2]) {
                size[vertex1] += size[vertex2];
                representative[vertex2] = vertex1;
            } else {
                size[vertex2] += size[vertex1];
                representative[vertex1] = vertex2;
            }
            return 1;
        }
    }

```

Complexity Analysis

Here E = Number of edges, V = Number of vertices.

- Time complexity: $O(E \cdot \alpha(n))$.

  Iterating over every edge requires $O(E)$ operations, and for every operation, we are performing the `combine` method which is $O(\alpha(n))$, where α(n) is the inverse Ackermann function.

- Space complexity: $O(V)$.

  Storing the representative/immediate-parent of each vertex takes $O(V)$ space. Furthermore, storing the size of components also takes $O(V)$ space.

Report Article Issue

Comments: 19 Best Most Votes Newest to Oldest Oldest to Newest

Type comment here... (Markdown is supported)

Post

hansrajdas ★ 49 April 5, 2021 12:03 PM

Pyton `union find` solution

```python
class DSU:
    def __init__(self, n):
        self.parent = [i for i in range(n)]
        self.rank = [0 for _ in range(n)]

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]
```

Read More

17 Show 2 replies Reply

jackjia2021 ★ 10 July 14, 2021 8:07 PM

For approach 2, the time complexity should be O(V + E) because it takes O(V) to initialize the dsu.

10 Reply

shreshtavinayak ★ 15 April 15, 2021 3:29 PM

Python DFS Solution:

```
class Solution:
    def countComponents(self, n: int, edges: List[List[int]]) -> int:
        count = 0
        graph = [[] for _ in range(n)]
        seen = [False for _ in range(n)]

        for a, b in edges:
            graph[a].append(b)
            graph[b].append(a)
```

Read More

12 Show 1 reply Reply

buvaneshwarant ★ 9 April 10, 2021 4:00 PM

shouldn't the answer to this input be 1? [[2,3],[1,2],[1,3]]
because all the nodes are connected. Don't understand why it would be 2.
very confusing.
based of their sample inputs and outputs in the description, the output for the input in question should be 1.
this is my python solution.
class Solution(object):

```
def countComponents(self, n, edges):

    from collections import defaultdict
    adjacencyList = defaultdict(list)
```

Read More

9 Show 10 replies Reply

sanjeevurao ★ 4 September 13, 2021 4:25 AM

2
[[1,0]]
How is this a valid input when there is a constraint ai <= bi?

3 Show 1 reply Reply

pagrus ★ 23 May 3, 2021 10:42 PM

Union-find solution is fun.
One nitpick is time complexity of the UF solution introduces the undefined `n` in a(n). Yet `V` is not used, although `for (int i = 0; i < n; i++)` suggests that it impacts the time complexity.

3 Show 1 reply Reply

jindalakshunn ★ 3 May 2, 2021 2:48 PM

Can someone please explain "α(n) is the inverse Ackermann function"

3 Show 4 replies Reply

sjl5bz ★ 2 August 16, 2021 12:24 PM

This is similar to Number of Provinces
https://leetcode.com/problems/number-of-provinces/

2 Reply

MutouMan ★ 31 June 24, 2021 5:53 PM

Iterative approach. Beat 99% Python3

```
class Solution:
    def countComponents(self, n: int, edges: List[List[int]]) -> int:
        visited = [False] * n
        graph = {}
        for o, d in edges:
            graph.setdefault(o, []).append(d)
            graph.setdefault(d, []).append(o)

        ans = 0
```

Read More

1 Reply

Kenchlo ★ 1 3 days ago

For the DSU approach, the time complexity should be O(V + E alpha(V)), where O(V) is due to the initialization of the representative and size vectors.

0 Reply