1971. Find if Path Exists in Graph

Depth-First Search Breadth-First Search Union Find Graph

Problem Description

Easy

between these vertices are provided in the form of a 2D array called edges, where each element of the array represents an edge connecting two vertices. For example, if edges[i] = [u, v], there is an edge connecting vertex u to vertex v and vice versa since the graph is bi-directional. It's also specified that each pair of vertices can be connected by at most one edge and that no vertex is connected to itself with an edge.

Our goal is to find out if there exists a path from a given source vertex to a destination vertex. If such a path exists, the function should return true; otherwise, it should return false.

In this problem, we are given a bi-directional graph consisting of n vertices, labeled from 0 to n - 1. The connections or edges

Intuition

The intuitive approach to determine if a valid path exists between two vertices in a graph is by using graph traversal methods

such as <u>Depth-First Search</u> (DFS) or <u>Breadth-First Search</u> (BFS). These methods can explore the graph starting from the <u>source</u> vertex and check if the <u>destination</u> vertex can be reached.

However, the solution provided uses a different approach known as Disjoint Set Union (DSU) or Union-Find algorithm, which is an efficient algorithm to check whether two elements belong to the same set or not. In the context of graphs, it helps determine if two vertices are in the same connected component, i.e., if there is a path between them.

The Union-Find algorithm maintains an array p where p[x] represents the parent or the representative of the set to which x

belongs. In the given solution, the find function is a recursive function that finds the representative of a given vertex x. If x is the representative of its set, it returns x itself; otherwise, it recursively calls itself to find x's representative and performs path compression along the way. Path compression is an optimization that flattens the structure of the tree by making every node

Then, in the solution, every edge [u, v] is considered, and the vertices u and v are united by setting their representatives to be the same. In the end, it checks whether the source and the destination vertices have the same representative. If they do, it means they are connected, and thus, a valid path exists between them, returning true; else, it returns false.

By using Union-Find, we can avoid the need to explicitly traverse the graph while still being able to determine connectivity

two main operations: find and union. In the context of the problem, Union-Find helps to efficiently check if there is a path between two vertices in the graph.

set operation where each vertex is initially the parent of itself, meaning each vertex starts in its own set.

The solution provided employs the Union-Find (Disjoint Set Union) algorithm. The implementation of this algorithm consists of

The algorithm uses an array named p where p[i], initially, is set to i for all vertices 0 to n-1. This step constitutes the make-

The find function is then defined. This function takes an integer x, which represents a vertex, and it recursively finds the representative (parent) of the set that x belongs to. This is done by checking if p[x] is equal to x. If p[x] != x, then x is not

false, implying there is no valid path.

The edges array is provided as follows:

We would use the Union-Find algorithm for this.

Initially, all vertices are their own parents:

After step 1, our updated p array is:

previously). Hence, p[2] = 0.

p = [0, 0, 0, 3, 4]

p = [0, 0, 0, 3, 3]

representatives.

edges = [[0, 1], [1, 2], [3, 4]]

can visualize the graph:

between vertices efficiently.

Solution Approach

setting p[x] = find(p[x]). Path compression is a crucial optimization as it helps to reduce the time complexity significantly by flattening the structure of the tree.

Next, a loop iterates over each edge in the edges list. For each edge [u, v], the union operation is performed implicitly by

the representative of its set, so the function is called recursively with p[x]. During recursion, path compression is performed by

setting p[find(u)] = find(v). This essentially connects the two vertices u and v by ensuring they have the same representative. By performing this operation for all edges in the graph, all connected components in the graph are merged.

Finally, the solution checks whether there is a valid path between source and destination. This is done by comparing their

representatives: if find(source) == find(destination), then source and destination are in the same connected component,

signifying that there is a path between them, and hence true is returned. If the representatives are different, the function returns

It is worth noting that the Union-Find algorithm is particularly effective for problems involving connectivity queries in a static graph, where the graph does not change over time. This algorithm allows such queries to be performed in nearly constant time, making it a powerful tool for solving such problems.

Example Walkthrough

0 --- 1 --- 2 3 --- 4

Our goal is to determine if there is a path between the source vertex 0 and the destination vertex 3. From the edges given, we

We start by uniting the vertices that have an edge between them using the union operation.

p = [0, 1, 2, 3, 4]

p = [0, 0, 2, 3, 4]

Let's consider a small graph with 5 vertices (n = 5) labeled from 0 to 4.

```
After step 2, our updated p array is:
```

3. Union operation on edge [3, 4]: set the parent of the representative of 4 (p[4]) to be the representative of 3 (p[3]). Hence, p[4] = 3.

Now we check if there is a valid path between the source vertex 0 and the destination vertex 3 by comparing their

1. Union operation on edge [0, 1]: set the parent of the representative of 1 (p[1]) to be the representative of 0 (p[0]). Hence, p[1] = 0.

2. Union operation on edge [1, 2]: set the parent of the representative of 2 (p[2]) to be the representative of 1 (which was set to 0

Find the representative of the source vertex 0:

confirming our conclusion.

Solution Implementation

Python

Java

C++

public:

};

TypeScript

const parent: number[] = [];

parent[i] = i;

function initializeParent(n: number): void {

parent[x] = find(parent[x]);

function unionSet(x: number, y: number): void {

return find(source) === find(destination);

def find root(node: int) -> int:

if parent[node] != node:

for start node, end node in edges:

which uses find, becomes nearly constant.

return parent[node]

parent = list(range(n))

// Function to perform the union operation on two subsets

for (let i = 0; i < n; i++) {

function find(x: number): number {

if (parent[x] !== x) {

const rootX = find(x);

const rootY = find(y);

if (rootX !== rootY) {

from typing import List

class Solution:

parent[rootX] = rootY;

return parent[x];

class Solution {

vector<int> parent;

// n - number of vertices

// source - starting vertex

// destination - ending vertex

// Parameters:

class Solution {

private int[] parent;

for (int[] edge : edges) {

private int find(int x) {

find(0) will return 0 since p[0] is 0.

Find the representative of the destination vertex 3:

them, and the function should return false.

def find root(node: int) -> int:

if parent[node] != node:

for start node, end node in edges:

return parent[node]

parent = list(range(n))

After step 3, our updated p array is:

```
• find(3) will return 3 since p[3] is 3.

Since the representatives of vertex 0 and vertex 3 are different (0 is not equal to 3), we conclude there is no path between
```

It is evident from the p array and the disconnected components in the graph visualization that vertices 0, 1, and 2 are in one

connected component, and vertices 3 and 4 form another component. There's no edge that connects these two components,

from typing import List

class Solution:
 def validPath(self, n: int, edges: List[List[int]], source: int, destination: int) -> bool:

Helper function to find the root of a node 'x' in the disjoint set.

Union the sets by updating the root of one to the root of the other.

// Parent array that stores the root of each node's tree in the disjoint set forest

It also applies path compression to optimize future lookups.

Initialize parent pointers for each node to point to itself.

parent[node] = find_root(parent[node])

Iterate through each edge and perform union operation.

parent[find_root(start_node)] = find_root(end_node)

Check if the source and destination are in the same set.

return find_root(source) == find_root(destination)

parent[find(edge[0])] = find(edge[1]);

return parent[x]; // Return the root of the set that contains x

// Parent vector to represent the disjoint set (union-find) structure

// Global parent array to represent the disjoint set (union-find) structure

// Function to initialize the parent array with each vertex as its own parent

// Function to find the root parent of a vertex 'x' using path compression

// edges - list of edges represented as pairs of vertices

// Main function to check if there is a valid path between source and destination

bool validPath(int n, vector<vector<int>>& edges, int source, int destination) {

return find(source) == find(destination);

If the find root of both is the same, they are connected.

public boolean validPath(int n, int[][] edges, int source, int destination) {
 // Initialize the parent array where each node is initially its own parent (representative of its own set)
 parent = new int[n];
 for (int i = 0; i < n; ++i) {
 parent[i] = i;
 }

// Union operation: merge the sets containing the two nodes of each edge</pre>

// Method to determine if there is a valid path between the source and the destination nodes within an undirected graph

// If the source and destination nodes have the same parent/root, they are connected; otherwise, they are not

// Method to find the root of the set that contains node x utilizing path compression for efficiency

if (parent[x] != x) { // If x is not its own parent, it's not the representative of its set

parent[x] = find(parent[x]); // Recurse to find the root of the set and apply path compression

```
// Initialize parent array so that each vertex is its own parent initially
    parent.resize(n);
    for (int i = 0; i < n; ++i)
        parent[i] = i;
    // Iterate through all edges to perform the union operation
    for (auto& edge : edges)
        unionSet(find(edge[0]), find(edge[1]));
   // Check if the source and destination have the same root parent
    // If they do. there is a valid path between them
    return find(source) == find(destination);
// Helper function to find the root parent of a vertex x
int find(int x) {
   // Path Compression: Recursively makes the parents of the vertices
    // along the path from x to its root parent point directly to the root parent
    if (parent[x] != x)
        parent[x] = find(parent[x]);
    return parent[x];
// Helper function to perform the union operation on two subsets
void unionSet(int x, int v) {
    // Find the root parents of the vertices
    int rootX = find(x);
    int rootY = find(y);
   // Union by setting the parent of rootX to rootY
    if (rootX != rootY)
        parent[rootX] = rootY;
```

```
function validPath(n: number, edges: number[][], source: number, destination: number): boolean {
   initializeParent(n);

   // Perform the union operation for each edge to connect the vertices in the union-find structure
   for (const edge of edges) {
      unionSet(find(edge[0]), find(edge[1]));
   }

   // If the source and destination have the same root parent, a valid path exists
```

def validPath(self, n: int, edges: List[List[int]], source: int, destination: int) -> bool:

Union the sets by updating the root of one to the root of the other.

// Main function to check if there is a valid path between 'source' and 'destination'

Helper function to find the root of a node 'x' in the disjoint set.

It also applies path compression to optimize future lookups.

Initialize parent pointers for each node to point to itself.

parent[node] = find_root(parent[node])

Iterate through each edge and perform union operation.

parent[find_root(start_node)] = find_root(end_node)

Check if the source and destination are in the same set.

return find_root(source) == find_root(destination)

If the find root of both is the same, they are connected.

```
Time and Space Complexity

The given Python code represents a solution to check if there's a valid path between the source and destination nodes in an undirected graph. The code utilizes the Union-Find algorithm, also known as the Disjoint Set Union (DSU) data structure.

Time Complexity:

The time complexity of this algorithm can be considered as 0(E * α(N)) for the Union-Find operations, where E is the number of
```

edges and $\alpha(N)$ is the Inverse Ackermann function which grows very slowly and is nearly constant for all practical values of N. The reason for this time complexity is that each union operation, which combines the sets containing u and v, and each find operation, which finds the root of the set containing a particular element, takes $\alpha(N)$ time on average.

The find function uses path compression which flattens the structure of the tree by making every node point to the root

whenever find is used on it. Because of path compression, the average time complexity of find, and thus the union operation

Given that there are E iterations to process the edges array, the time complexity of the for loop would be $O(E * \alpha(N))$.

Additionally, there are two more find operations after the for loop, but these do not significantly affect the overall time

complexity, as they also work in $\alpha(N)$ time. Space Complexity:

```
The space complexity of the code is O(N), where N is the number of nodes in the graph. This is because we maintain an array p that holds the representative (parent) for each node, and it's of size equal to the number of nodes.
```

Therefore, the space complexity of maintaining this array is linear with respect to the number of nodes.