Path Detection with Disjoint Sets

Loren Holl

(SML) Graph Theory Spring 2024

In this report, I will introduce the disjoint sets data structure and how it can be used to create an efficient algorithm for path detection in graphs. It is used in many graph algorithms thanks to its low time complexity, and it is especially well-suited for path detection. I will start by introducing the definitions necessary to understand the disjoint sets data structure, then I will move on to how it relates to graphs. Finally, I will explain the algorithm and a thorough example of how it works.

Disjoint sets (also called *Union-Find*) is a data structure that consists of a set of disconnected sets and two operations, *Union* and *Find*. Let S be the set of disjoint sets with sets $s_0, s_1, \ldots, s_n \in S$ such that $S = \{ s_0, s_1, \ldots, s_n \}$. Each $s_i \in S$ ($i = 0, 1, \ldots, n$) is also its own individual set such that $s_i = \{$ *elements of* $s_i \}$. Next, let's take a look at the first operation defined, *Union.* For two sets $s_i, s_j \in S$, let the *Union* of s_i, s_j be the set $s_k = s_i \cup s_j = \{$ *elements of* s_i *and*[1] *elements of* $s_j \}$. Finally, I will explain the second operation, *Find.* For some set $s_i \in S$, let *Find* be the function that takes some input x and returns the set s_i if $x \in s_i$, and -1 otherwise (assuming -1 is not a valid value in the set).

Let's look at some examples. For a disjoint sets structure S, where $S = \{ s_0, s_1, s_2, s_3 \}$, let $s_0 = \{ 0 \}$, $s_1 = \{ 1 \}$, $s_2 = \{ 2 \}$, and $s_3 = \{ 3 \}$. Figure 1.1 below shows multiple *Union* and *Find* operations on these sets. On the left side, the sets that will be affected by some operation are shown, and on the right side, the result of the operation is shown. For *Union*, an updated list of the sets of S are shown on the right to see how the set shrinks each time a *Union* operation is performed. For *Find*, the value that is returned by the operation is shown.

There is another useful way to represent the *Union* operation that will be used in our path detection algorithm. Instead of creating a new set everytime the *Union* operation is performed on s_i and s_j, we can simply set $s_i = s_j$ and add s_i's elements to s_j, and s_j's elements to s_i. An example of this strategy is shown in Figure 1.2, and will be the basis of the *Union* strategy we use to detect paths.

[1] Here, "and" is used in the sense that it is a combination of the elements of the sets, not the logical AND.

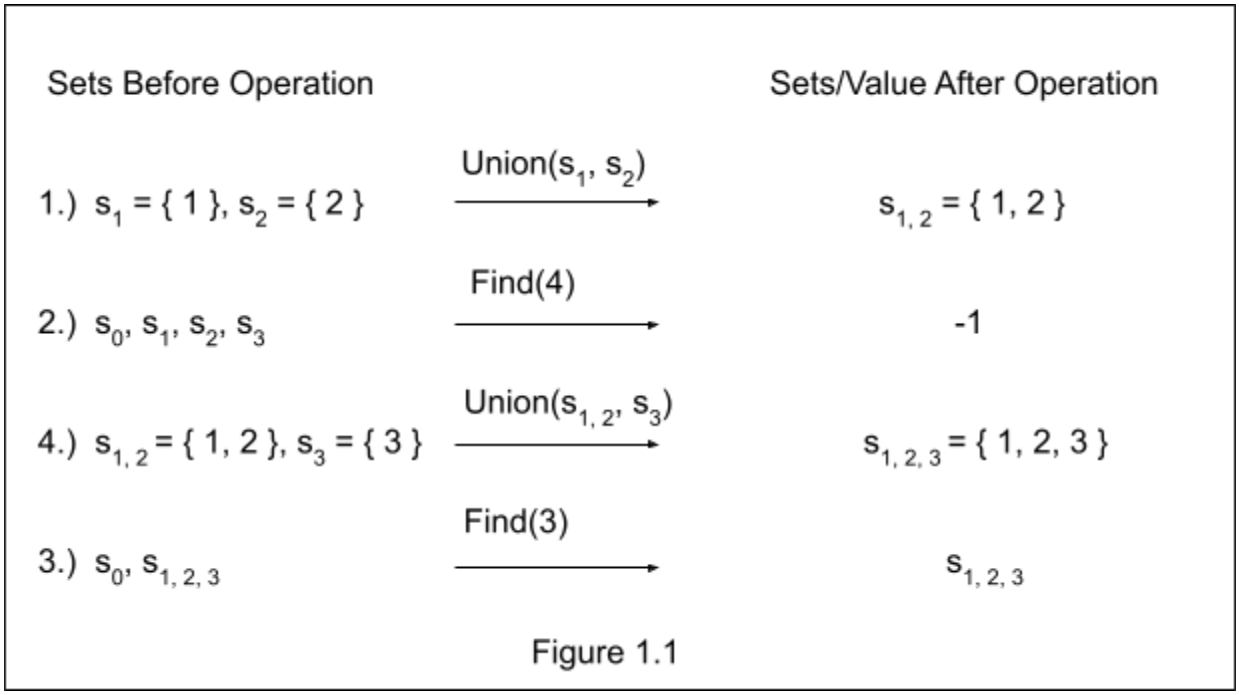


Figure 1.1

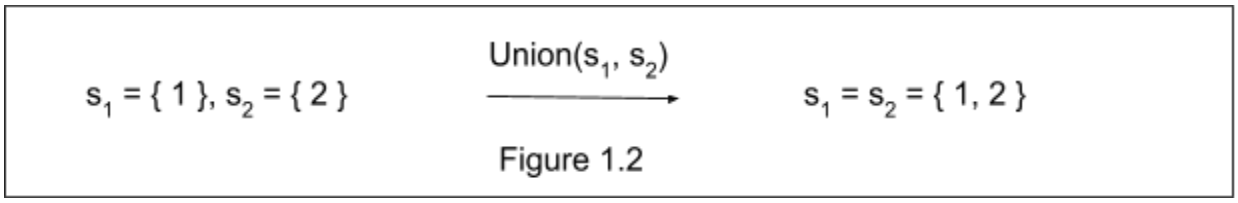


Figure 1.2

The final topic that needs to be discussed before moving on to paths is the idea of representatives. A *representative* is some value R_i in the set $s_i \in S$ that holds a unique value representing the set it belongs to. This allows us to efficiently distinguish one set from another without unnecessary comparisons. At first, every set $s_i \in S$ has a unique representative representing s_i. However, every time a *Union* operation is performed on s_i and s_j, one of their representatives becomes the others, since now $s_i = s_j$. For example, if one decides to call *Union* on all sets of *S,* then one will be left with only one representative that would represent the whole set of *S*.

A common way to picture this is through a set of anti-arborescence trees (Figure 1.3). Let *T* be a set of anti-arborescence trees of *n* nodes, where $T_i \in T$ holds the values of $s_i \in S$ and each s_i is of length *n*. For the case of path detection, we will set $n = 1$ (a). Now, let the root of T_i be the representative R_i of the set s_i. This means that the set *T* is a set of single node trees holding only the representative R_i of s_i as the root. Each time a *Union* is performed on s_i and s_j, their

respective up-tree representations, T_i and T_j, will also get merged together (b). For maximum efficiency, the typical strategy is to add the smaller tree to the larger tree (c).

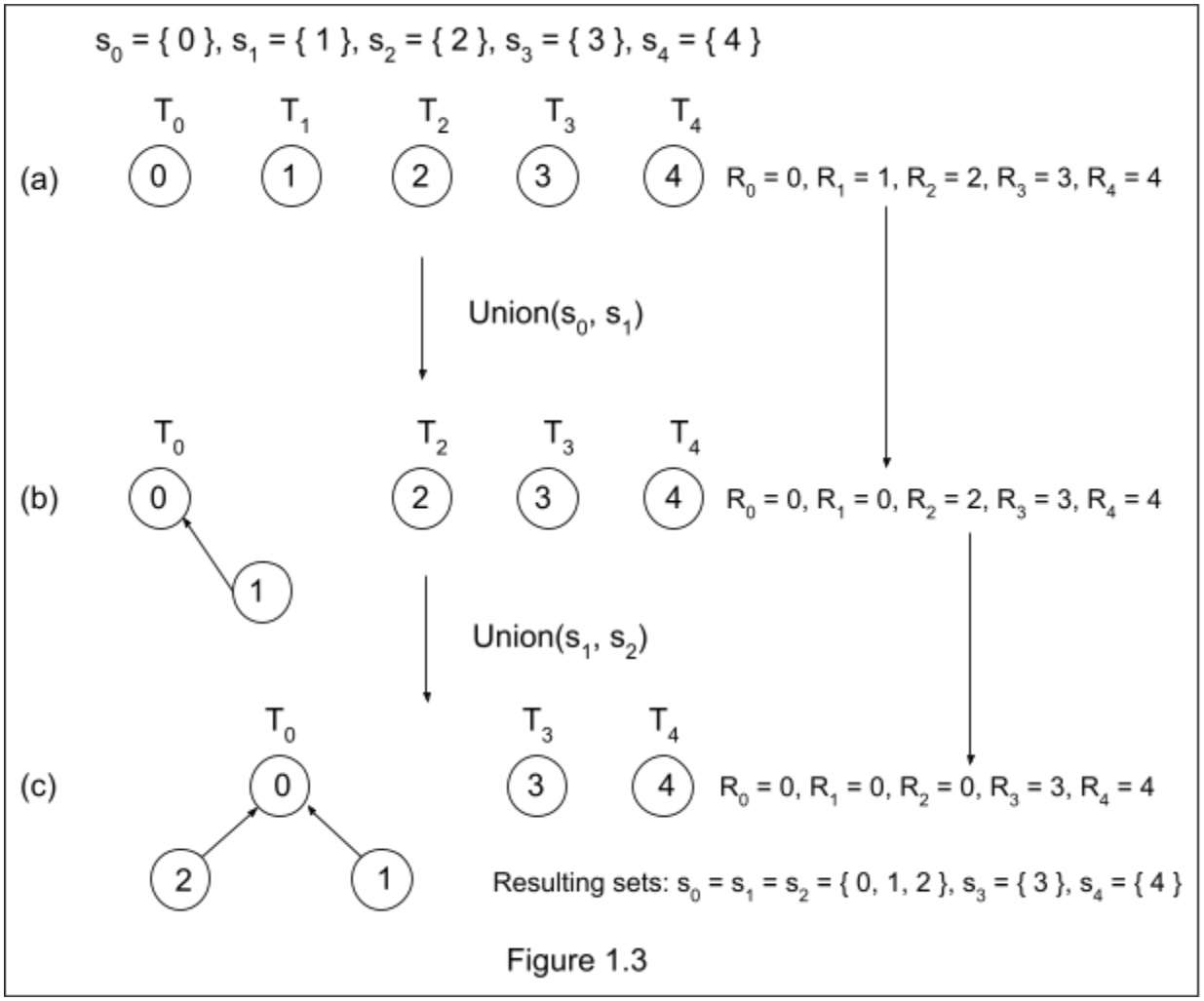


Figure 1.3

Since T_i is an anti-arborescence tree, this means that each node will be pointing toward R_i, giving us an efficient way to get to the representative of the set no matter where we start on the tree. This idea is the foundation for our path detection algorithm.

Now that we've understood the intuition behind disjoint sets, let's move on to how they can be applied to graphs. First, I would like to briefly review the concept of a connected component. A *connected component* C_u of graph G is the subgraph of G that contains the vertex u. Essentially, if there is some path starting from any $v_i \in$ V that can be followed to reach u in G, then C_u is the *connected component* of G. For example, if G is a connected graph, then C_u is the same graph as G. Using this concept, let's see how disjoint sets are applied to graphs. Let G be an edgeless, undirected graph with vertices V, and let each $v_i \in$ V be equal to s_i in the disjoint sets structure S (Figure 1.4). Since G has no edges, every vertex of G is its own connected

component of G. This implies that $S = \{ \{ v_1 \}, \{ v_2 \}, ..., \{ v_n \} \}$, where $n = |G|$. Every time a *Union* is performed on S, the corresponding vertices of G will be "merged," where an edge is added between them. Next time we want to see if a vertex is connected between these two vertices, we only need to check if the new vertex to merge is already connected to one of the above vertices to determine if it is already part of the same connected component, which is exactly the role of representatives. Figure 1.4 below shows how graphs can be represented with disjoint sets and how the *Union* operation works.

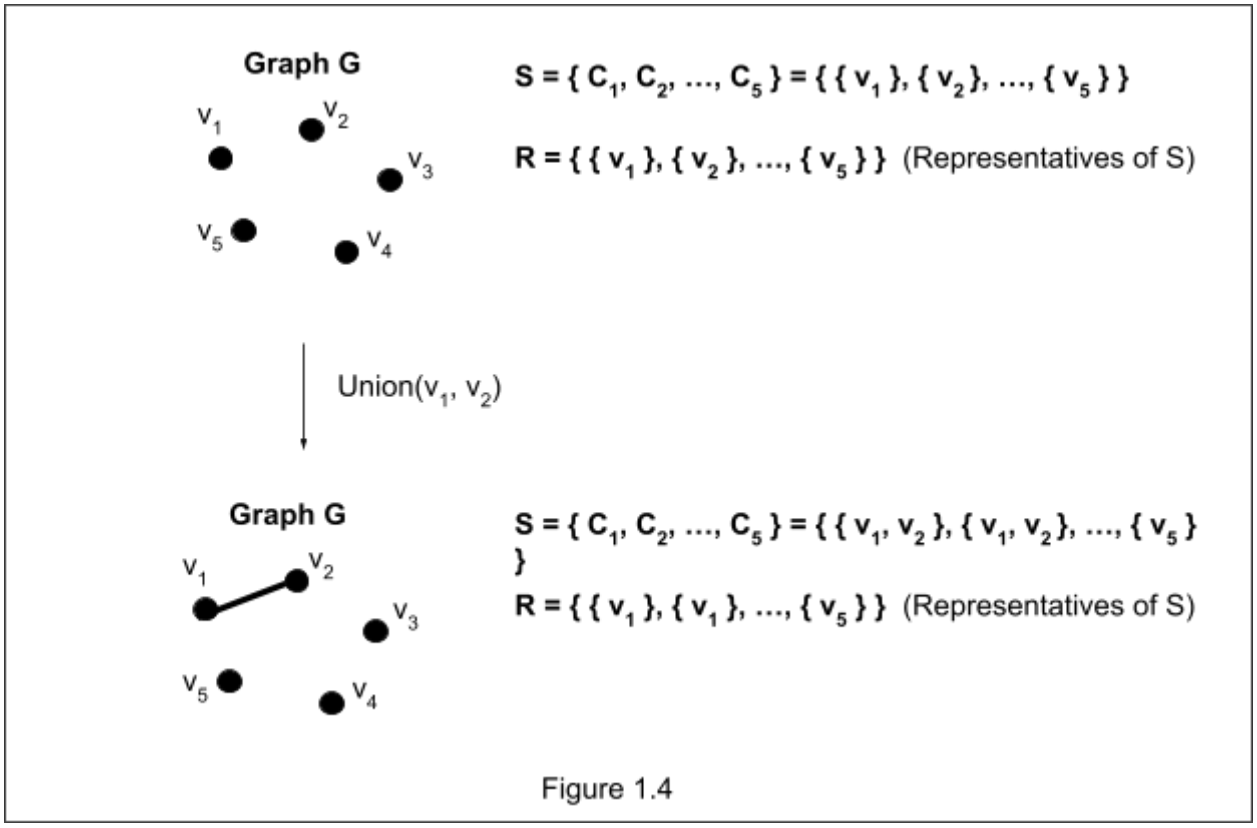


Figure 1.4

Now that we've covered all the prerequisite knowledge, it's time to present the path detection algorithm. It uses a very similar idea to the one above, except it works on any undirected graph. Let G be an undirected[2] graph with n vertices $v_i \in V$. Let S be the disjoint sets structure containing the connected components C_u of graph G, such that $S = \{ C_{v1}, C_{v2} ..., C_{vn} \}$, which starts out as $\{ \{ v_1 \}, \{ v_2 \}, ..., \{ v_n \} \}$. Here, we do not make any assumptions about the connectedness of the graph, so we start out the connected components of G with only one vertex. Our goal for the algorithm is to, given the adjacency lists of graph G, determine whether two vertices lie in the same connected component. If they do lie in the connected component, then

[2] This algorithm also works with directed graphs, however, a little more thought is required.

there must be a path between them, clearly observed from the definition of a connected component.

We start by inputting the list of adjacency lists A of G, where each $A_i \in A$ represents the adjacency list from the vertex v_i, and the two vertices that we want to check if a path exists between. Next, we initialize S as described above, a list of representatives R, and a list of pointers *Ptrs* initialized to $\{-1, -1, ..., -1\}$ for n vertices. *Ptrs* is our way to tell if the representative has been reached, or where we need to go next to reach the representative. If the value of *Ptrs* for some vertex is negative, it means that vertex is the representative. If it is positive, it holds the next index that needs to be checked for the representative. It will also be used later to determine which tree needs to be added to which during a *Union* operation (the larger the value, the bigger the tree).

Before we continue, we need to redefine the *Find* operation for our algorithm. *Find* recursively searches the *Ptrs* list until a negative value (representative) is found, and returns it. It also has a special function called *path compression*, that shortens the path to a representative each time it is called. In terms of up-trees, this means that it shortens the tree by attaching the current node in the search to the root (representative) that we are looking for.

Now that we have our *Find* operation, let's continue with the algorithm. The next step is to start a loop through all vertices, that for each vertex v_i, we take the next connected vertex v_j through its adjacency list A_i, find both representatives and check if they are equal. If they are equal, then we know that they belong to the same connected component, and therefore a path exists between them. If these elements happened to be the ones we wanted to know about, then we can stop our algorithm early. If not, then we need to keep going. If they are not equal, then we call *Union* on the two vertices, which will set the representative of the smaller vertex belonging to the smaller set to the one of the larger set. This is reflected by adding the *Ptrs* list value of the vertex of the smaller set to the one of the larger set. Once the algorithm has run on all the vertices, we simply check the representatives of the vertices we care about with *Find*, and if they are equal, a path exists. If not, then no path exists. The *Find* function and pseudocode for this algorithm is presented below:

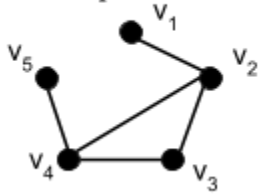
Find(v):
1) If Ptrs[v] < 0 return v *// If v is a representative, simply return it*
2) Else:
    R_v *= Find(Ptrs[v])* *// Recursively search for representative*
    Ptrs[u] = R_v *// Path compression*
    Return R_v *// Return*

Algorithm 1.6 (Path Detection):
Input: Adjacency list A_i and two vertices u_i and u_j.

1) *$n = |G|$ // Set n to the number of vertices*
2) *For $i = 0 \sim n$: $Ptrs[i] = -1$ // Set the pointers*
3) *For $i = 0 \sim n$: // Loop through each vertex v_i*
   For each u in A_i : // Loop through each vertex u in A_i of v_i
   $R_v = Find(v_i)$ // Find representative of v_i
   $R_u = Find(u)$ // Find representative of u
   If ($R_v == R_u$), where $v = u_i$ and $u = u_j$:
   Return True // Representatives of desired vertices match, so True
   If ($R_v != R_u$):
   $Union(R_u, R_v)$ // Merge the two sets together if not equal, update Ptrs
4) *$R_{ui} = Find(u_i)$, $R_{uj} = Find(u_j)$ // Find representatives of desired vertices*
5) *If ($R_{ui} == R_{uj}$) Return True // If a path exists, return true*
6) *Else Return False // If no path exists, return false*

Finally, an example of how this algorithm works is given below, followed by an up-tree representation of the *Union* operations on the sets.

Graph G



Adjacency Lists

$A_1 = \{ v_2 \}$
$A_2 = \{ v_1, v_3, v_4 \}$
$A_3 = \{ v_2, v_4 \}$
$A_4 = \{ v_2, v_3, v_5 \}$
$A_5 = \{ v_4 \}$

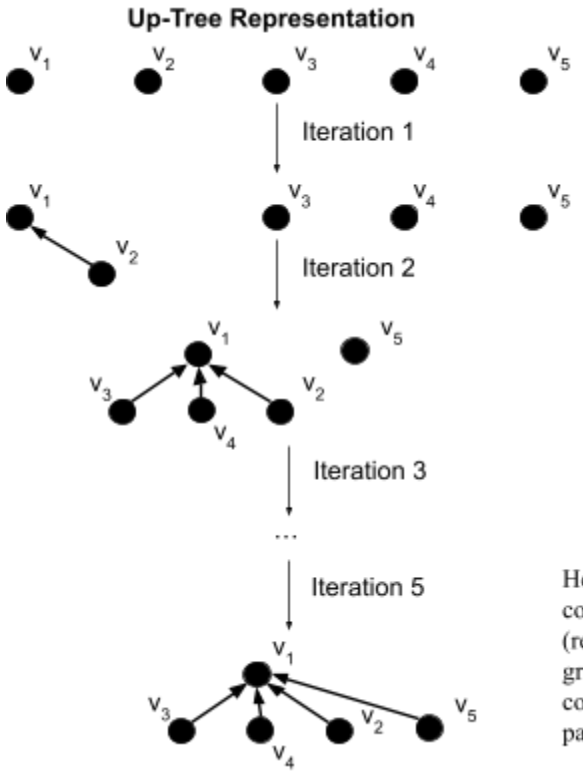
Desired Points

v_1, v_3

n = 5, Ptrs = { -1, -1, -1, -1, -1 }

1st Iteration (v_1):
$R_1 = Find(v_1)$, $R_2 = Find(v_2)$, Not equal, $Union(R_1, R_2)$ // Ptrs = { -2, 0, -1, -1, -1 }
2nd Iteration (v_2):
$R_2 = Find(v_2)$, $R_1 = Find(v_1)$, Equal, Continue
$R_2 = Find(v_2)$, $R_3 = Find(v_3)$, Not Equal, $Union(R_2, R_3)$ // Ptrs = { -3, 0, 0, -1, -1 }
$R_2 = Find(v_2)$, $R_4 = Find(v_4)$, Not Equal, $Union(R_2, R_4)$ // Ptrs = { -4, 0, 0, 0, -1 }
3rd Iteration (v_3):
$Find(v_3)$, $Find(v_2)$, Equal, Continue
$Find(v_3)$, $Find(v_4)$, Equal, Continue
…
5th Iteration(v_5):
$Find(v_5)$, $Find(v_4)$, Equal, Continue

Final Check: $R_1 = Find(v_1)$, $R_3 = Find(v_3)$, Equal, Return True // Ptrs = { -5, 0, 0, 0, 0 } (Path Compression)



Here, all vertices are connected to the root (representative) of the graph, meaning this is a connected graph, and a path $v_1 -> v_3$ must exist