CS-204: Design and Analysis of Algorithms

220001050, 220001052, 220001060

March 15, 2024

1 Union-Find Data Structure

Union-find is a data structure designed for managing collections of elements divided into separate groups. These groups *never overlap*, meaning an element can only belong to one group at a time. A union-find structure has an operation to combine any two sets, and it is able to tell in which set or component a specific element is.

The basic functions included are:

- **Make-Set():** creates new disjoint sets consisting of the nodes v_i
- **Union(S_x, S_y):** unites two disjoint, dynamic set components that contain elements x_i and y_i, say S_x and S_y, into a new set that is the union of these two sets. The representative of the resulting set is any members of $S_x \cup S_y$.
- **Find-Set(x):** returns a pointer to unique componrnt containing x.

Arrays used in the implementation:

- component[x] $= k$ (the component ID of the component containing x)
- members[x] $= \{...\}$ (a lis representing members of the component x)
- size[x] $= k$ (the number of members in the component x)

The following is a basic implementation of the union-find operations.

1.1 Basic Implementation

1.1.1 Make-Set

This function creates a new component with the given element and appends it to the *component* array.

Algorithm 1 Make-Set

```
1: Input: A graph G containing the nodes whose components set has to be made.
2: Procedure Make-Set(G)
3:   int n = $|G.V|$
4:   int num-components = 0
5:   Declare component : Integer Array of size n
6:   Declare size : Integer Array of size n
7:   Declare members : Array of size n containing empty Integer Arrays
8: for each vertex in G.V : do
9:     component[vertex] $\leftarrow$ num-components
10:    member[num-components] $\leftarrow$ member[num-components] U vertex
11:    size[num-components] $\leftarrow$ 1
12:    num-components $\leftarrow$ num-components + 1
13: end for=0
```

1.1.2 Find-Set

This function finds the component a vertex belongs to. It retrieves the component ID from the *component* array.

Algorithm 2 Find-Set(x)

```
1: Procedure Find-Set($u$)
2:   componentID $\leftarrow$ component[vertex]
3:   return componentID =0
```

1.1.3 Union(S_x, S_y)

This function first receives two disjoint components componentIDs S_x and S_y which have been found by $Find - Set(x)$ and $Find - Set(y)$ respectively called before the calling of Union. It performs **union by size** (*small-to-large* merging). It inserts the members of the smaller component into larger one, while simultaneously updating the *component* array for each element added and then finally deleting the smaller component.

Suppose size of the component S_x is k and that of component S_y is k'. Then, if $k > k'$, then the component of y is merged into that of x, and vice versa.

Note: In the Union-Find function we will replace Make-Set(v) function by the initialization of an array and a map.

Algorithm 3 Union(S_x , S_y)

```
Procedure Union($S_x$ , $S_y$)
   componentID1 ← $S_X$
   componentID2 ← $S_y$
if  componentID1 == componentID2  then
     return
end if
if  size[componentID1] < size[componentID2]  then
   smallerComponentID := componentID1
   largerComponentID := componentID2
else
   smallerComponentID := componentID2
   largerComponentID := componentID1
end if
for element in members[smallerComponentID] do
   members[largerComponentID] ← {element} ∪ members[largerComponentID]
   component[element] ← largerComponentID
end for
size[largerComponentID] ← size[largerComponentID] + size[smallerComponentID]
size[smallerComponentID] ← 0
members[smallerComponentID] ← ∅ =0
```

Initial STATE:

$$\begin{aligned} \text{members} &= [] \\ \text{size} &= [] \\ \text{component} &= [] \end{aligned}$$

Step 1: Make-Set(1)

$$\begin{aligned} \text{members} &= [[1]] \\ \text{size} &= [1] \\ \text{component} &= [1] \end{aligned}$$

Step 2: Make-Set(2)

$$\begin{aligned} \text{members} &= [[1], [2]] \\ \text{size} &= [1, 1] \\ \text{component} &= [1, 2] \end{aligned}$$

Union(1, 2):

$$\begin{aligned} &\text{Finding componentID of 1:} \quad \text{componentID1} = 1 \\ &\text{Finding componentID of 2:} \quad \text{componentID2} = 2 \end{aligned}$$

Since they belong to different components, we proceed with the union.

$$\begin{aligned}
\text{Merging Component 1 into Component 2:} & \\
\text{Update members of Component 2:} \quad & [1,\ 2] \\
\text{Update component IDs:} \quad & [2,\ 2] \\
\text{Clear Component 1:} \quad & \varnothing
\end{aligned}$$

After Union(1, 2):

$$\begin{aligned}
\text{members} &= [\varnothing, [1,2]] \\
\text{size} &= [0,2] \\
\text{component} &= [2,2]
\end{aligned}$$

Step 3: Make-Set(3)

$$\begin{aligned}
\text{members} &= [\varnothing, [1,2], [3]] \\
\text{size} &= [0,2,1] \\
\text{component} &= [2,2,3]
\end{aligned}$$

Step 4: Make-Set(4), Make-Set(5)

$$\begin{aligned}
\text{members} &= [\varnothing, [1,2], [3], [4], [5]] \\
\text{size} &= [0,2,1,1,1] \\
\text{component} &= [2,2,3,4,5]
\end{aligned}$$

Union(4, 5):

$$\begin{aligned}
\text{Merging Component 5 into Component 4:} & \\
\text{Update members of Component 4:} \quad & [4,\ 5] \\
\text{Update component IDs:} \quad & [4,\ 4] \\
\text{Clear Component 5:} \quad & \varnothing
\end{aligned}$$

After Union(4, 5):

$$\begin{aligned}
\text{members} &= [\varnothing, [1,2], [3], [4,5], \varnothing] \\
\text{size} &= [0,2,1,2,0] \\
\text{component} &= [2,2,3,4,4]
\end{aligned}$$

Union(3, 5):

$$\begin{aligned}
\text{Merging Component 3 into Component 4:} & \\
\text{Update members of Component 4:} \quad & [3,\ 4,\ 5] \\
\text{Update component IDs:} \quad & [2,\ 2,\ 4,\ 4,\ 4] \\
\text{Clear Component 3:} \quad & \varnothing
\end{aligned}$$

After Union(3, 5):

$$\begin{aligned}
\text{members} &= [\varnothing, [1,2], \varnothing, [3,4,5], \varnothing] \\
\text{size} &= [0,2,0,3,0] \\
\text{component} &= [2,2,4,4,4]
\end{aligned}$$

1.2 Analysis of The Algorithm

1.2.1 Time Complexity Analysis

The Make-Set operation takes $O(n)$ time where n is the number of nodes in G which is due to the initialization of the component array and hash map. Find-Set operation runs in $O(1)$ time returning the corresponding component index.

The Union operation involves the merging of the smaller component with the bigger component. Let the size of the components be k_1 and k_2 where $k_1 > k_2$. Since in this operation we have to iterate over the complete smaller component therefore the time complexity of this operation will be O(k_2).

Also in the worst case scenario the size of k_2 can be $\lfloor \frac{n}{2} \rfloor$, hence in the worst case the time complexity of the union function becomes $O(n/2)$ which simplifies to $O(n)$.

1.2.2 The amortized time complexity of the union operation is $O(\log n)$.

Proof:
Consider an element x belonging to the set of vertices/elements and the component of size k_1. Now assume we want to take union of the component to which x belongs with a component of size k_2 assuming that $k_1 < k_2$ then we can say

$$k_1 + k_2 \geq 2k_1$$

Hence the element x now belongs to a component of size at least *twice* the component it previously belonged to. Therefore, if x initially belonged to component of size 2, then it goes to at least 4, then to 8 and so on...

$$1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 16...$$

The trend of the least size of the component to which x belongs

Thereby, we can easily conclude that in the worst case scenario for the element x, we will have to modify the componentID for x a maximum number of $\log n$ times where n is the total number vertices/elements we have.

Since we already proved that the size of component of any element which is getting merged into another larger component, grows by at least 2, the maximum merge operations performed on any x until the component to which it belongs becomes of size n will be $log_2(n)$, and the time taken will be O($\log n$).

Amortised Time Complexity
There are n vertices in the graph so the max final component size can be n and since for each vertex the time taken to become a member of component of size n is $O(\log n)$, the overall time complexity of all the union operations will be $O(n \log n)$.
Therefore amortised time complexity of single union operation becomes $O(\log n)$.

1.3 Dual Implementation

Another implementation of the Union-Find structure uses trees. Each node consists of two values: the value at that node, and its parent node. Each component consists of a root node that has its parent as itself. Here again, we maintain the sizes of the components (trees) in an array with the indices being the root of the respective components. An array parent[ ] is also maintained where parent[x] represents the parent of x, for all nodes x.

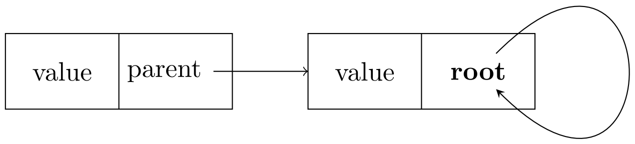


Figure 1: Nodes of the tree

For simplicity, we shall omit the reference cells in the following diagrams.

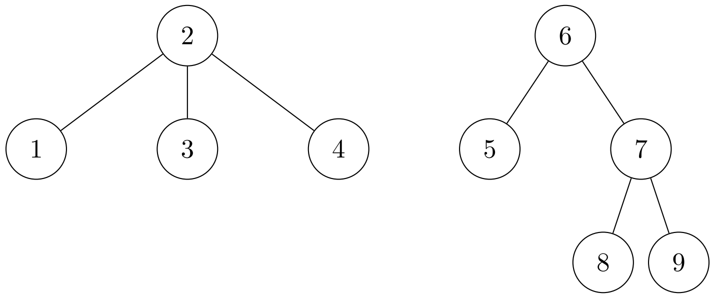
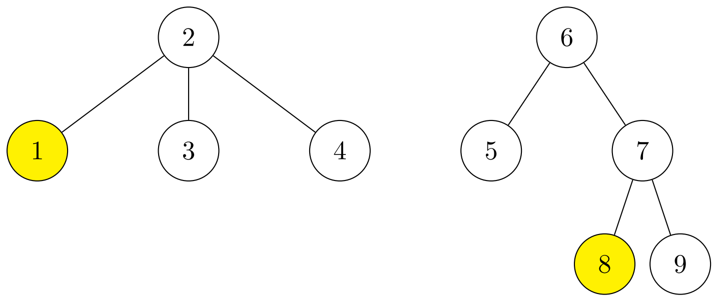
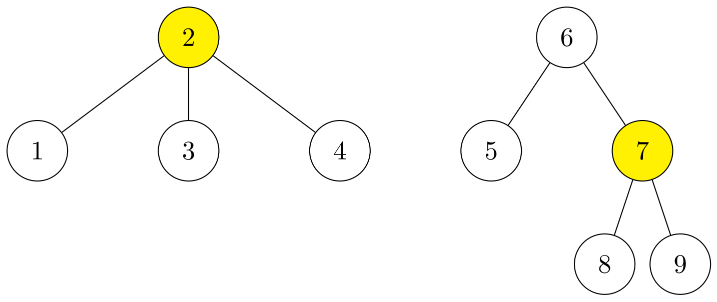


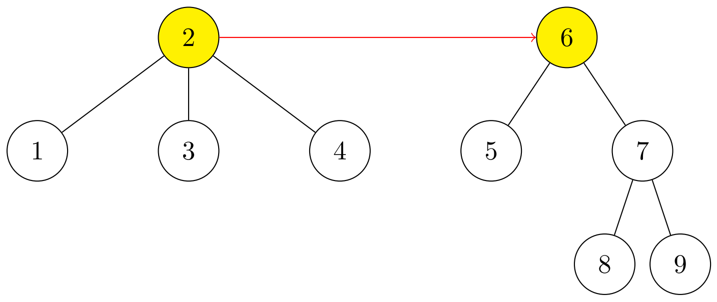
Figure 2: Dual Implementation of Union-Find

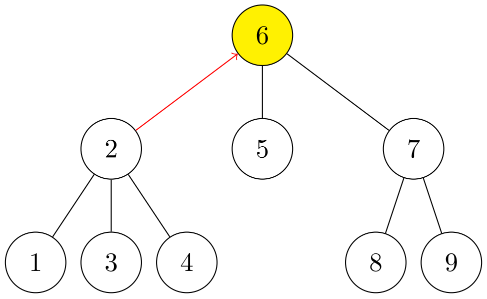
node:	1	2	3	4	5	6	7	8	9
size:	1	4	1	1	1	5	1	1	1
parent:	2	2	2	2	6	6	6	7	7

Suppose we call **Union(1, 8)**. Then recursively, the root nodes of the respective components will be found using the stored *ref* pointers. Here, the root nodes are 2 and 6 respectively for the given vertices.









node:	1	2	3	4	5	6	7	8	9
size:	1	1	1	1	1	9	1	1	1
parent:	2	6	2	2	6	6	6	7	7

Figure 3: Union operation, simultaneously implementing Find operation

Algorithm 4 Make-Set

```
1. **Procedure** Make-Set($x$):
2.     parent[$x$] $\leftarrow x$
3.     size[$x$] $\leftarrow 1$
4. **end Procedure**
```

Algorithm 5 Find-Set

```
1. **Procedure** Find-Set($x$):
2.     **if** $x \neq$ parent[$x$] **then**
3.         parent[$x$] $\leftarrow$ Find-Set(parent[$x$])
4.     **end if**
5.     **return** parent[$x$]
6. **end Procedure**
```

Algorithm 6 Union

```
1. Procedure Union($x, y$):
2.     $x\_root \leftarrow$ Find-Set($x$)
3.     $y\_root \leftarrow$ Find-Set($y$)
4.     if $x\_root \neq y\_root$ then
5.         if size[$x\_root$] $>$ size[$y\_root$] then
6.             parent[$y\_root$] $\leftarrow x\_root$
7.             size[$x\_root$] $\leftarrow$ size[$x\_root$] $+$ size[$y\_root$]
8.         else
9.             parent[$x\_root$] $\leftarrow y\_root$
10.            size[$y\_root$] $\leftarrow$ size[$y\_root$] $+$ size[$x\_root$]
11.        end if
12.    end if
13. end Procedure
```

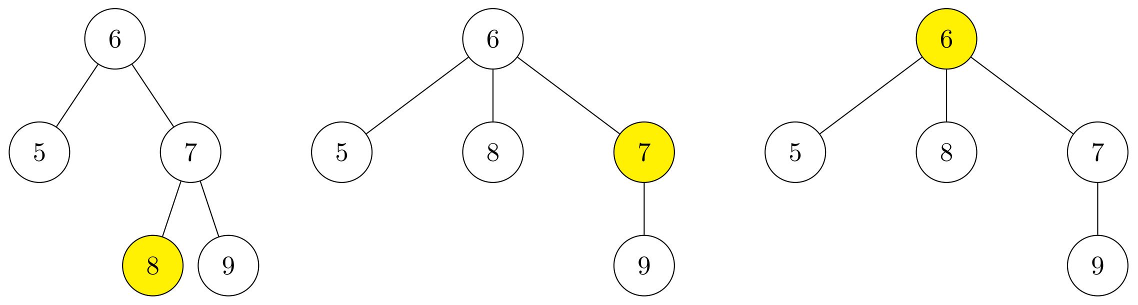
1.3.1 Analysis of Dual Implementation

Find-Set(x) takes $O(\log n)$ amortized time which is proportional to the height of the tree, if small-to-large merging is followed. This is because, during merging, we are increasing the degree of branching at the root of the larger component, which facilitates the reduction in the height of the tree.
The **Union(x, y)** operation is hence also of $O(\log n)$ time complexity, while the **Make-Set(x)** operation is of $O(1)$ time complexity.

The space complexity is $O(n)$ due to the extra memory used for the storage of the *size* array and the *parent* values at each node.

However, the time complexity of **Find-Set(x)** can further be improved to $O(\alpha(n))$ where $\alpha(n)$ represents the **inverse Ackermann function,** which grows incredibly slowly. In fact, it grows so slowly that its value does not exceed 4 for all reasonable n (approximately $n < 10^{600}$). This can be achieved using union by size, coupled with **path compression** in the recursive traversal towards the root of a component.

For instance, in the previous example, the second component could have its path from 8 to the root compressed as:



In this optimization while traversing from a specific node to the root, all the nodes that are visited in the process will be attached to the final root node of the component found eventually. This will drastically speed up future queries of finding components.

2 Max Flow Min Cut

It consists of a few key concepts:

Network Flow graph: A flow network is a directed graph where each edge has a capacity and two special nodes: a **source (S)** and a **sink (T)**. The goal is to push as much flow from the source to the sink respecting the edge capacities.

$$\text{Network Flow Graph} \rightarrow \text{Directed Graph } G(V, E)$$

A flow in a network is a function that assigns a value to each edge, representing the amount of flow passing through that edge. The flow must satisfy two properties:

1. **Capacity constraint:** The flow on each edge must be less than or equal to the edge's capacity.

   $$\text{C : E} \rightarrow R^+$$

   indicates maximum load transmissible through edge.

2. **Conservation constraint:** At each node (except the source and sink), the total incoming flow must equal the total outgoing flow. Therefore,

   $$\sum V_{in} = \sum V_{out}$$

The maximum flow minimum cut theorem states that the maximum value of a flow in a network is equal to the minimum capacity of a cut in the network. Here, a **cut** is a partition of the nodes into two sets, S and T, such that the source (S) is in S, the sink (T) is in T, and no edge goes from a node in S to a node in T.

$$\text{Max Flow Problem} \equiv \text{Min Cut Problem}$$