THE DISJOINT-SET DATA STRUCTURE AND ITS PERFORMANCE

Alexandru ANDRIEȘ[1*], Alexandra KONJEVIC[2], Maria AFTENI[2]

[1]*Department of Software Engineering and Automation,, FAF-211, Faculty of Computers, Informatics and Microelectronics, Technical University of Moldova, Chisinau, Moldova*
[2]*Department of Software Engineering and Automation,, FAF-213, Faculty of Computers, Informatics and Microelectronics, Technical University of Moldova, Chisinau, Moldova*

**Corresponding author: Andrieș Alexandru, alexandru.andries@isa.utm.md*

Abstract. *Disjoint-set data structure is a data structure that stores a collection of disjoint (non-overlapping) sets.The Disjoint-Set data structures (also called Union-Find) is a very elegant and efficient way of analyzing connectivity in a graph. It makes use of the concept of dynamic programming to reach its running speed. Using splitting or halving with union by size or by rank, or using path compression, we can reduce the running time of these operations.*

Keywords: *algorithms, inverse Ackermann function, amortized constant time, graph, optimization, shortest spanning tree, union-find*

Introduction

A disjoint-set data structure, also known as a merge–find set or a union–find data structure, represents a data structure that contains a set of disjoint subgraphs. The main usage of this data structure is to detect cycles in graphs/ subgraphs and group elements into disjoint graphs. Three operations that are supported by disjoint-set data structures: creating a new set with a new element, detecting the root element of a subgraph containing an element and merging two sets.

Algorithm concept

The *Union* operation makes one of the groups that we want to merge (usually the smaller one) point to the other as its root, thus also leading all its children there for a further *Find* operation, as shown in *Figure 1*.

The most common and simple way to store the vertices is through an array of size V, where V is the vertex (node) count. Another array of pointers will take care of storing the parent of each vertex, leading to the root vertex of the group. The pointer leading to the same vertex means that the vertex we are searching is its own root. Walking along this path of pointers until we reach a self-referencing one is the *Find* operation itself, and if two vertices have the same root, connecting them would make a cycle [1].

In Disjoint-set forest implementations with Find operation that doesn't update parent pointers and with Union operation that does not control the heights of trees, there can be trees with height O(n). In this case, the Find and Union operations will take O(n) time.

The combination of path compression optimization, as well as union by size or union by height reduces the running complexity to $\Theta (\alpha (n))$ [2], where $\alpha (n)$ is the inverse Ackermanm function, the running time also being called *amortized constant time*.

Figure 1a, 1b represents the union operation without path compression.

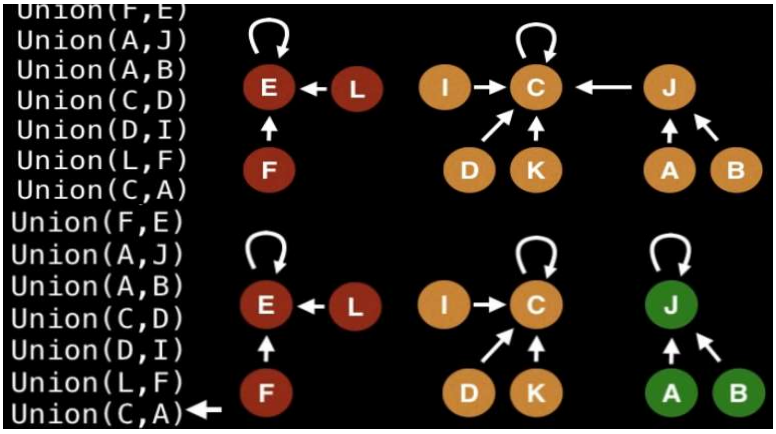

Figure 1a,1b. Union operation of the graph [1]

Method I of the algorithm -The Naive method

Method I represents the unoptimized and not compressed version of the algorithm. The shown functions union() and find() unoptimized for demonstrational purposes and have the time complexity of (O(n))..

```
5   int find(int parent[], int i){
6       if(parent[i]==-1)
7           return i;
8       else return find(parent, parent[i]);
9   }
10
11  void Union(int parent[], int x, int y){
12      int x_p=find(parent, x);
13      int y_p=find(parent, y);
14      parent[y_p]=x_p;
15  }
```

Figure 2. Naive method C++ code [3]

Method II of algorithm optimization -The Union by rank

Method II can be optimized to O(Log n) worst case running time. The smaller depth tree must be attached under the root of the deeper tree. This is known as union by rank. In path compression technique, rank is not necessarily equal to height – that's why the term "rank" is used instead of "height". Using the size of the tree instead of the rank yields the same time complexity of the algorithm – O(Log n).

Figure 3 represents a graph on which we apply union operations.

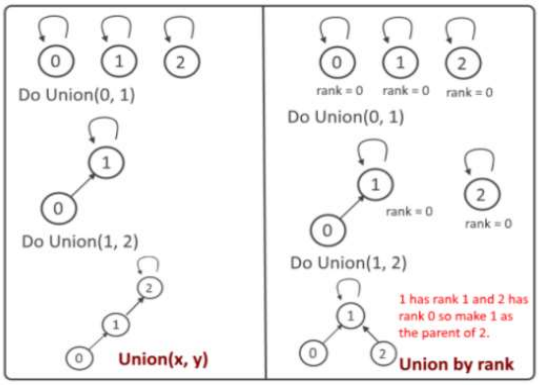

Figure 3. Method II Graph [4]

Figure 4 represents the pseudo-code for the union operation shown in Figure 3.

Pseudo Code:

```
function Union(x, y)
  xRoot := Find(x)
  yRoot := Find(y)

  // x and y are already in the same set
  if xRoot == yRoot
     return

  // x and y are not in same set, so we merge them
   if xRoot.rank < yRoot.rank
     xRoot.parent := yRoot
   else if xRoot.rank > yRoot.rank
     yRoot.parent := xRoot
  else
    xRoot.parent := yRoot
    yRoot.rank   := yRoot.rank + 1
```

Figure 4. Method II Pseudo Code [4]

Method III of algorithm optimization - The Path compression

The best optimization for the find() function is path compression. The idea is to flatten the tree in the process of finding the root node, highly reducing the height of the tree in the process . The find() function goes up the reference tree to find the root node. Once the root node is known, the function goes the same path again, assigning the parent of every node to the root one directly, making a direct reference and thus reducing the running time of the subsequent calls of the function to any node along this path or that are added to it.

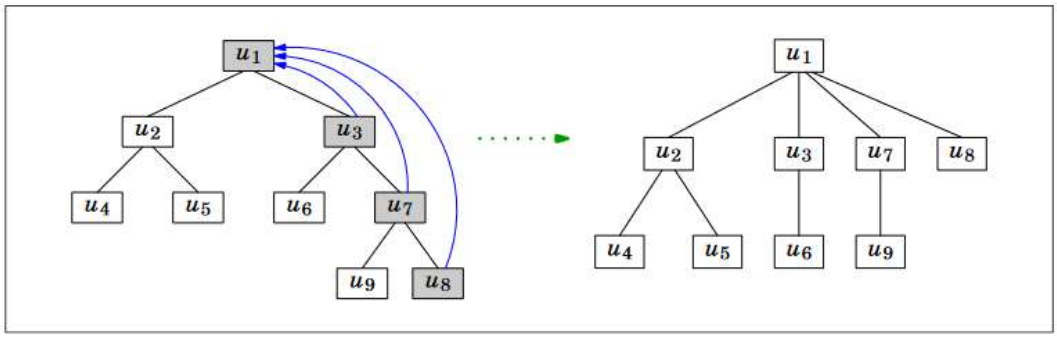

Figure 5. With path compression, calling FIND - SET(u8) will have the side-effect of making u8 and all of its ancestors direct children of the root [5].

The MakeSet operation (fig. 6) creates a new set by pointing the root of the unified set to the root of the parent set. To find the root element we go up the pointer chain until we find an element that points to itself. The MakeSet operation has O(1) time complexity in its simplest form. Comparison of time needed for execution of algorithms without path compression and with path compression:

Algorithms	Worst-case time
Quick-find	mn
Quick-union	mn
QU + Union by Rank	$n + mlogn$
QU + Path compression	$n + mlogn$
QU + Union by rank + Path compression	$\boldsymbol{n + mlog^*n}$

m union-find operations on a set of n objects.

Figure 6. Time complexity of the different implementations, where log*n is the inverse Ackermann function [7]

Conclusions

The most optimized implementation of the disjoint set/union-find data structure has an immense complexity advantage over the least efficient implementation, and an even greater one against other approaches for solving the same problem using a different method, like Breadth First Search. The use of other computer science algorithmic approaches: tree balancing and dynamic programming, makes the data structure so efficient that it should be a must-know for any programmer working who has to work with graphs.

References:

1. FISET. W, *Union Find - Union and Find Operations* [online]. [accessed 20.02.2022]. Available: https://youtu.be/0jNmHPfA_yE
2. NIVASCH. G, *Inverse Ackermann without pain* [online]. [accessed 20.02.2022]. Available: https://www.gabrielnivasch.org/fun/inverse-ackermann
3. *Union-Find Algorithm | Set 2 (Union By Rank and Path Compression)*[online]. [accessed 21.02.2022]. Available: https://www.geeksforgeeks.org/union-find-algorithm-set-2-union-by-rank/
4. Jain. S, *Disjoint Set | Union-Find Algorithm – Union by rank and path compression* [online]. [accessed 28.02.2022]. Available:https://algorithms.tutorialhorizon.com/disjoint-set-union-find-algorithm-union-by-rank-and-path-compression/
5. CORMEN. T. H., LEISERSON. C. E., RIVEST. R. L., STEIN. C, *Introduction to Algorithms* (Chapter 21), London:The MIT Press, 2009.
6. WEI (TERENCE) LI| *Time Complexity of Union-Find*[online]. [accessed 03.03.2022] Available:https://pt.slideshare.net/WeiLi73/time-complexity-of-union-find-55858534
7. EDNOVAS | (Graph) Algorithms [online]. [accessed 03.03.2022] Available: https://ednovas.xyz/2021/10/01/algorithm/#Disjoint-Set-Union-Find

Recommended reads/ watches:

In-depth description:

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2012/lecture-notes/MIT6_046JS12_lec16.pdf

Video explanations:

FISET. W,| Union Find Playlist [online]. [accessed 03.03.2022] Available: https://www.youtube.com/watch?v=ibjEGG7ylHk&list=PLDV1Zeh2NRsBI1C-mR6ZhHTyfoEJWlxvq