Union-Find

Niccolò Piazzesi
n.piazzesi@studenti.unipi.it

March 27, 2021

Abstract

Union-Find, also called Disjoint Set Union (DSU), is an interesting data structure, not only for the algorithmic techniques and analysis involved, but also for its application to certain problems. In this report I will present a high level view, describing the fundamental structure and the various implementations. In the last section, I will show a collection of problems for which the use of Union-Find provides an efficient solution.

1 Description

Suppose we have a collection S of n distinct elements, let's say the integers from 1 to n. Initially, these elements are organized in n disjointed sets {1},{2},...,{n}. The name of set {i} is i. In the Union-Find problem we want to maintain a collection of disjointed sets while supporting the following operations:

- **union**(A, B): join the sets A and B in a single set $A \cup B$. The old sets A and B are destroyed.
- **find**(x): given an element x, return the name of the set that contains it.

Let's now generalize the problem. We have assumed that the elements of the collection are already given. This is not true in general, so we need a way to add new elements (and consequentially new sets). This is solved by introducing a third operation:

- **makeSet**(x): Create a new set x containing the single element x.

We will say that the Union-Find problem consists of maintaining a collection of disjointed sets during an arbitrary sequence of *makeSet, union* and *find* operations, starting from the empty set [1]. Figure 1 shows a simple example where the elements considered are numbers.

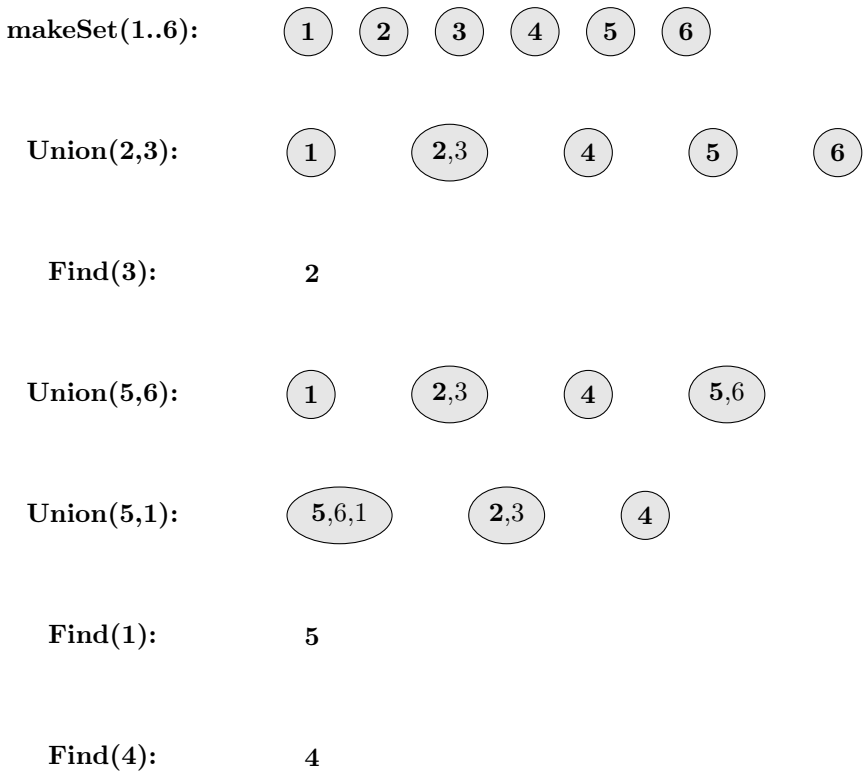

Figure 1: A running example with 6 initial elements. The name of each set is written in bold

2 Naive Implementations

The general idea for implementing Union-Find is to represent the collection as a forest of trees, with each tree representing a single set. In the following code, we will consider integers as elements and use an array representation of the forest. Given an array *parents*, the element *parents[i]* contains the father of element i in the tree hierarchy. For the root r of a tree we will have parents[r] = r. Before analyzing a more efficient implementation, we will see two simpler versions.

2.1 QuickFind

QuickFind aims at having a very good time complexity for the find operation. To achieve this, it represent each set as a tree of height 1. Each element has a pointer to the root of the tree, and the root contains the name of the set. The find operation simply follows the pointer and returns the root. It's easy to see that it has $O(1)$ time complexity. MakeSet(e) creates a new tree composed by two nodes: the root and a single leaf child. It stores e in both of them. MakeSet is $O(1)$ as well. The union operation deletes the pointer from the elements to the root of the old tree and add new pointers to the root of the new tree. In the worst case, this takes $O(n)$. Space complexity is $O(n)$.

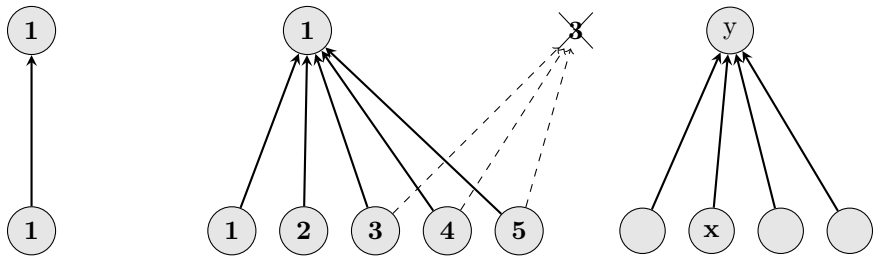

Figure 2: makeSet(1), Union(1, 3) and find(x) = y using QuickFind

Listing 1 shows a possible implementation in $C++$. As i said before, we use an array representation so each pointer becomes simply an element of the array. If we want to compute find(i) we return the element parents[i];

Listing 1: QuickFind implementation

```
class QuickFind
{

private:
    int n;
    std::vector<int> parents;

public:
    QuickFind(int n) : n(n)
    {
        parents.resize(n);
```

```
        std::iota(parents.begin(), parents.end(), 0);
    }

    void makeSet(int x)
    {
        if (x >= parents.size())
            parents.resize(x);
        parents.at(x) = x;
    }
    int find(int x)
    {
        return parents[x];
    }

    void set_union(int a, int b)
    {

        for (auto i : parents)
        {
            if (i == b)
                i = a;
        }
    }
};
```

2.2 QuickUnion

In QuickUnion we also use trees, but they can have height $\neq 1$. Each element has a pointer to their father. MakeSet(x) creates a single node named x. Union(A,B) makes the root of B a child of the root of A. The find operation is slightly more complex. Let's say we want to compute find(e) for a generic e. Starting from e we repeatedly follow the pointer to the father until we find the root of the tree.

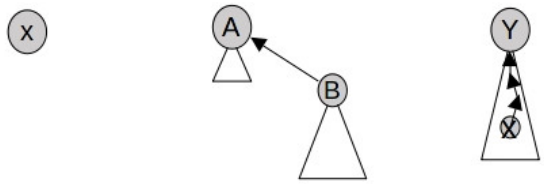

Figure 3: makeSet(x), union(A,B) and find(x) = y in QuickUnion

The time complexity of find grows linearly with the height of the tree. If we start with n disjointed sets, the worst case is having a tree of height n. A sequence of $n-1$ union operations that leads to this height is the following: union(1,2), union(2,3),..., union(n-1,n). In this situation, find has $O(n)$ time complexity.

Listing 2: QuickUnion implementation

```
class QuickUnion
{
private:
    int n;
    std::vector<int> parents;

public:
    QuickUnion(int n) : n(n)
    {
        parents.resize(n);
        std::iota(parents.begin(), parents.end(), 0);
    }

    void makeSet(int x)
    {
        if (x >= parents.size())
            parents.resize(x);
        parents.at(x) = x;
    }
    int find(int x)
    {
        while(parents[x] != x) x = parents[x];
        return parents[x];
    }

    void set_union(int a, int b)
    {
        int pa = find(a);
        int pb = find(b);
        parents[pb] = pa;
    }
};
```

Now we discuss the cost of union. As previously said, in this implementation union introduces a new pointer from the root of the old tree to the root of the new tree. This is done in $O(1)$ time. If we look at the code in Listing 2, we notice that, when doing a union, we first call find to find the root of the trees containing the elements. Find takes $O(n)$ and this dominates the total complexity, so the union method in that specific code takes $O(n)$ as well. But this is just one possible implementation. We are analyzing the abstract operation, so our previous analysis is still valid.

3 Balancing Heuristics

Now we will see some heuristics that aim to improve the time complexity of Union-Find. In particular, we will first see a heuristic to improve union in the QuickFind algorithm, and then various heuristics to improve find in the QuickUnion algorithm. Note that, in this section, we will simply describe the techniques, while the complexity analysis will be shown in the next part.

3.1 Balancing union in QuickFind

To improve the performance of QuickFind we must focus on the most expensive operation, which we know is union. We can observe that the implementation

that we described in section 2.1 is not very smart. The more $|B|$ grows, the more Union(A, B) costs. When we have $|A| < |B|$, it is more efficient to move the nodes of the set A.

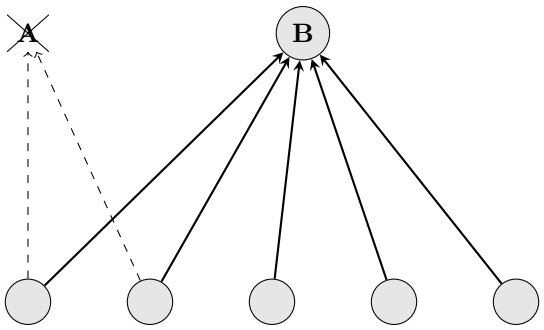

Figure 4: Union(A,B) when $|A| < |B|$

In the implementation, we store an auxiliary array *size*, which keeps track of the cardinality of each set.

Listing 3: updated operations in QuickFind

```
QuickFind(int n) : n(n){
    parents.resize(n);
    std::iota(parents.begin(), parents.end(), 0);
    std::vector<int> size(n, 1);
}

void makeSet(int x){
    if (x >= parents.size())
        parents.resize(x);
    parents.at(x) = x;
    size.at(x) = 1;
}
void set_union(int a, int b)
{
    if (a != b){
        if(size[a] < size[b]) std::swap(a,b);
        for (auto i : parents){
            if (i == b)
                i = a;
        }
        size[a] += size[b];
    }
}
```

3.2 Balancing heuristics for QuickUnion

To improve the QuickUnion algorithm, we need to make *find* more efficient. In section 2.2, we observed that this operation has a worst-case $O(n)$ time complexity. This is due to the fact that we allow the height of the trees to grow without any control when doing a *union*.

Union by Size The first heuristic uses the same idea of the QuickFind union heuristic. When doing union(A,B), we make the smaller set in size child of the bigger set.

Listing 4: Union by size implementation

```
void set_union(int a, int b){
    a = find(a);
    b = find(b);
    if (a != b) {
        if (size[a] < size[b])
            std::swap(a, b);
        parents[b] = a;
        size[a] += size[b];
    }
}
```

Union By Rank In union by rank we use the height of the tree instead of its size. The approach is the same of Union By size: we attach the shorter tree to the taller one. The rank of the new tree will have the same rank as the taller tree. If the two trees have the same rank r, the rank of the new one will be $r + 1$. It's important to notice that, when we use compression techniques that we will discuss later, the rank is not always equal to the actual height. In this case it gives an upper bound of the actual value.

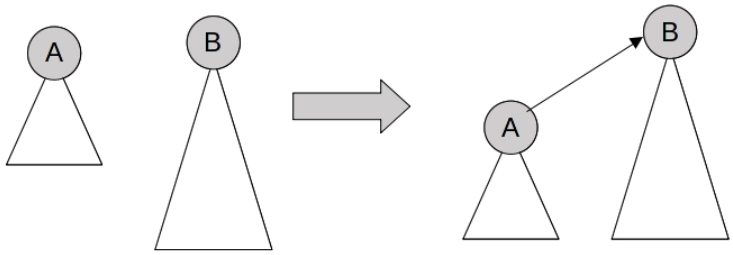

Figure 5: Union(A,B) when rank(A) $<$ rank(B)

Listing 5: QuickUnion using Union by Rank

```
QuickUnion(int n) : n(n){
    parents.resize(n);
    std::iota(parents.begin(), parents.end(), 0);
    ranks.resize(n);
}

void makeSet(int x){
    if (x >= parents.size())
        parents.resize(x);
    parents.at(x) = x;
```

```
    ranks.at(x) = 0;
}
void set_union(int a, int b){
    a = find(a);
    b = find(b);
    if (a != b){
        if (ranks[a] < ranks[b])
            std::swap(a, b);
        parents[b] = a;
        if (ranks[a] == ranks[b])
            ranks[a]++;
    }
}
```

To further improve QuickUnion performance, we can also modify *find* in order to reduce the height of a tree even more. Let $u_0.u_1, \ldots, u_{l-1}$ be the nodes visited during *find(x)*, where $u_0 = x$ and u_{l-1} is the root of the tree containing x. We can assume $l \geq 3$, because if $l \leq 2$ we don't need to reduce the height. We can apply one of the following compression heuristics:

Path compression make all the nodes u_i $(0 \leq i \leq l-3)$ children of the root u_{l-1}[2].

```
int find(int x){
    int p = x;
    if (parents[p] != p)
        parents[p] = find(parents[p]); //path compression
    return parents[p];
}
```

Path splitting make the node $u_i (0 \leq i \leq l-3)$ a child of its grandparent u_{i+2}[3] [4].

```
public int find(int x) {
    while (x != parent[x]) {
        int next = parent[x];
        parent[x] = parent[next]; // path splitting
        x = next;
    }
    return x;
}
```

Path halving make the node $u_{2i} (0 \leq i \leq \lfloor \frac{l-1}{2} \rfloor - 1)$ a child of its grandparent u_{2i+2}[3] [4].

```
public int find(int x) {
    while (x != parent[x]) {
        parent[x] = parent[parent[x]]; // path halving
        x = parent[x];
    }
    return x;
}
```

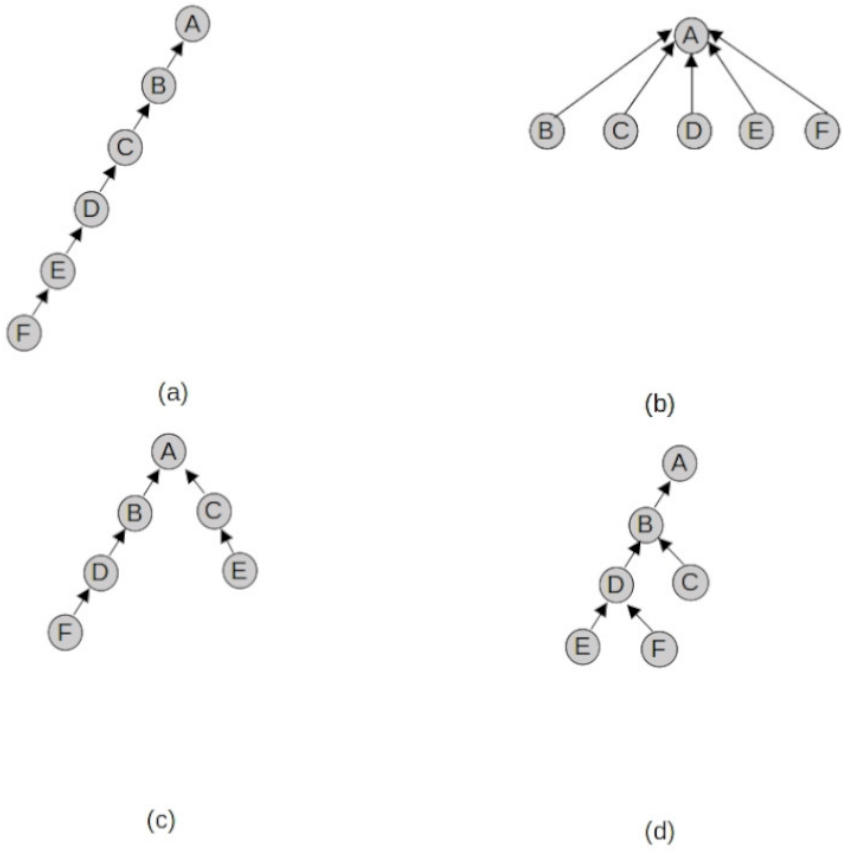

Figure 6: (a) The tree before executing *find(F)*; (b) *find(F)* with path compression; (c) *find(F)* with path splitting; (d) *find(F)* with path halving;

4 Complexity analysis

Let's analyze how much the heuristics improve the total time complexity. Note that any combination of the union and find heuristics shown gives the same improvement in terms of time complexity.

In particular, we will now see how much does union by rank improves the time complexity of find in the QuickUnion version. We will not consider any compression heuristic on the find itself for now. Let's start with the following result:

Lemma 1. *A QuickUnion tree balanced by rank has at least* $2^{rank(x)}$ *nodes, where* x *is the root of the tree.*

Proof. We demonstrate the lemma by induction on the number of operations. The base case is trivially proven since we have no trees. Now let's assume that the lemma is valid before an operation. We prove that the lemma holds after a find, a makeSet or a union. When we execute a find, the rank is not modified and so the lemma still holds. If we execute a makeSet(x), we introduce a new tree with the single node x, and rank(x) = 0. This tree has $2^{rank(x)} = 2^0 = 1$ node and the lemma is verified. We now consider $union(x, y)$. We indicate with $|x|$ and $|y|$ the size of the trees before merging them, while $|x \cup y|$ is the size of the resulting tree. $Rank(x)$ represent the rank of the root of x. Since the two sets are disjointed, $|x \cup y| = |x| + |y|$. We have three separate cases:

1. If $rank(y) < rank(x)$, the root of y becomes a child of the root of x. The root of the new tree will have $rank(x \cup y) = \text{rank}(x)$. By inductive hypothesis we have $|x| \geq 2^{rank(x)}$ and $|y| \geq 2^{rank(y)}$ and the number of nodes in the resulting tree is:

$$|x \cup y| = |x| + |y| \geq 2^{rank(x)} + 2^{rank(y)} > 2^{rank(x)} = 2^{rank(x \cup y)}$$

2. If $rank(y) < rank(x)$, we follow the same reasoning of the first case. The only difference is that the root of x becomes a child of the root of y.

$$|x \cup y| = |x| + |y| \geq 2^{rank(x)} + 2^{rank(y)} > 2^{rank(y)} = 2^{rank(x \cup y)}$$

3. When $rank(x) = rank(y)$, union makes the root of y child of the root of x and updates $rank(x)$ to $rank(x)+1$. This is also the value of $rank(x \cup y)$. By using the inductive hypotheses, the number of nodes in new tree is:

$$|x \cup y| = |x| + |y| \geq 2^{rank(x)} + 2^{rank(y)} \geq 2 \cdot 2^{rank(x)} = 2^{rank(x)+1} = 2^{rank(x \cup y)}$$

In each case the lemma is still true after executing the union. The following corollary is an immediate consequence.

Corollary 1. *During a sequence of makeSet, union and find, the height of a balanced QuickUnion tree has a $\lfloor \log_2 n \rfloor$ upper bound, where n is the total number of makeSet.*

Proof. Let T be a balanced QuickUnion tree, and let x be its root. We call $size(x)$ the number of nodes in T. By definition, the height of T is $rank(x)$ and $size(x) \leq n$. For Lemma 1, we have $size(x) \geq 2^{rank(x)}$ and, if we combine all these results:

$$rank(x) \leq \lfloor \log_2(size(x)) \rfloor \leq \lfloor log_2 n \rfloor.$$

This corollary guarantees that, using union by rank, a QuickUnion tree will have $O(\log n)$ height. As we know, the time complexity of find is directly correlated to the height of the tree. We can affirm the following result:

QuickUnion trees balanced by height support makeSet and union operations in constant time. A find takes $O(\log n)$ in the worst case, where n is the total numbers of makeSet executed.

During any sequence of the operations described, the following properties are verified:

Property 1. The rank of a generic node x is initially 0 and increases until x remains a root. As soon as x is not root anymore, its rank remains unchanged.

Property 2. Let x be a generic, non root node and let $p(x)$ be its father node.

$$rank(p(x)) > rank(x)$$

Property 3. For each node x $rank(x) \leq \lfloor \log_2 n \rfloor$, where n is the total number of nodes.

In the previous analysis, we did not consider any compression heuristic during a find. We can use one of them to improve the time complexity even more. Assume that we use Union By Rank combined with path compression. In 1973, Hopcroft and Ullman proved that a sequence of m makeSet, find or union is done in $O(m \log^* n)$ amortized time, where log^* is the iterated logarithm and n the number of makeSet executed [2]. In 1975, Tarjan proved that the same sequence of operations is done in $O(m\alpha(n))$ amortized time, where α is the inverse of the Ackermann function [5]. This function grows even slower than $\log^*$. The proof of this bound is very complex and goes beyond the scope of this report, so it is omitted.

5 Applications

Let's now see some problems where the DSU data structure shows its power.

5.1 Detect cycle in a graph

Given an undirected graph $G = (V, E)$, check if it contains a cycle.

A trivial solution simply traverse the graph with a DFS visit in $O(|V| + |E|)$ time and space. We can achieve a more space efficient solution using Union-Find. Create a set for each vertex $v \in V$. Now, for every edge $(u, v) \in E$:

1. Find the root of the sets that contain u and v
2. If we find the same root, a cycle has been found and we stop
3. Otherwise, merge the two sets

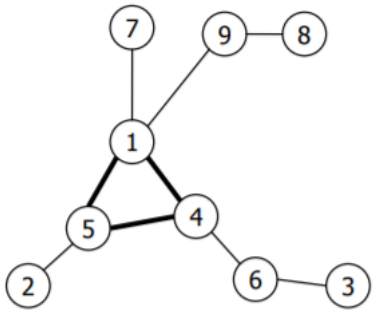

Figure 7: A graph with a cycle

Time complexity. We do $O(|V| + |E|)$ makeSet, find and union operations. Thanks to the results shown in section 4 we know that this takes $O((|V| + |E|)\alpha(|V|))$ time. This is not exactly linear as the trivial solution, but α grows so slowly that, in practice, there is almost no difference in the runtime.

Space complexity. $O(|V|)$ to store the sets.

Listing 6: Cycle detection

```
struct Edge {
    int u, v;

    Edge(int u, int v, int weight) : u(u), v(v){}

};


bool graphcycle(std::vector<Edge> edges, int nodes){

    UnionFind uf = UnionFind(nodes);
    bool cycle = false;
    for(auto edge: edges){
        if(uf.find(edge.u) == uf.find(edge.v)){
            cycle = true;
            break;
        }
        uf.set_union(edge.u,edge.v);
    }
    return cycle;

}
```

5.2 Kruskal's algorithm

Given a connected, undirected and weighted graph $G = (V, E)$, find the ***minimum spanning tree****(MST) of G. The minimum spanning tree is a subtree of G that*

contains all the vertices $v \in V$, and with the minimum possible total edge weight.

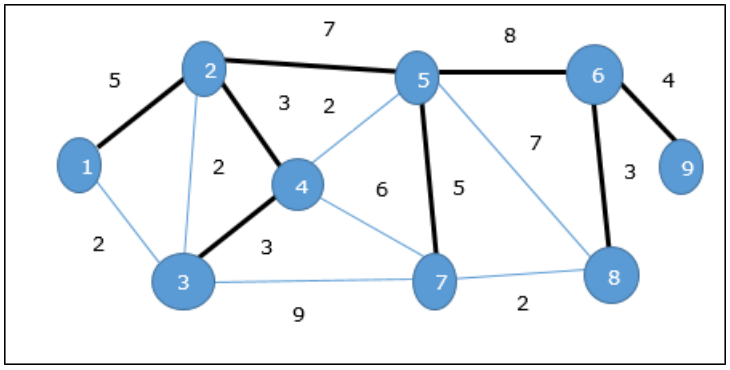

Figure 8: An example of a minimum spanning tree

There are many algorithms to find the MST of a given graph [6] [7], but now we will see the algorithm described by Joseph Kruskal [8]. This algorithm starts from an empty tree, and increasingly builds the MST with the following steps:

1. Sort all the edges by increasing weight.
2. Pick the next smallest edge. If it forms a cycle with the tree formed so far discard the edge, otherwise add it to the tree.
3. Repeat step 2 until there are $|V| - 1$ edges in the tree.

The central point of the algorithm is the cycle detection in step 2. We can use a modified version of the algorithm described in 5.1. Given a edge (u, v) we find the sets to which u and v belong. If they are not in the same set, they do not form a cycle and so we add the edge to the current tree and merge the sets together.

Time complexity. Sorting the edges is $O(|E| \log |E|)$. Finding cycles takes $O((|V| + |E|)\alpha(|V|))$. Assuming that the graph is connected, we have

$$|E| \geq |V| - 1$$

which means that the disjoint sets operations take $O(|E|\alpha(|V|))$. We know that $\alpha(|V|) = O(\log |V|)$, and this tells us that the total time complexity of Kruskal's algorithm is $O(|E| \log |E|)$ [9]. From graph theory we observe that $|E| < |V|^2$. If we apply the logarithm, $\log |E| = O(\log |V|))$, and so we reestablish the total time complexity as:

$$O(|E| \log |V|)$$

Listing 7: Kruskal Algorithm

```
struct Edge {
    int u, v, weight;

    Edge(int u, int v, int weight) : u(u), v(v), weight(weight){}
    bool operator<(Edge const& other) {
        return weight < other.weight;
    }
};

std::vector<Edge> mst(std::vector<Edge>& edges, int nodes){
    UnionFind uf = UnionFind(nodes);
    std::sort(edges.begin(), edges.end());
    std::vector<Edge> mst;
    for(Edge e: edges){
        if(uf.find(e.u) != uf.find(e.v)){
            mst.push_back(e);
            // the mst has n-1 edges
            if(mst.size() == nodes - 1) break;

            uf.set_union(e.u,e.v);
        }
    }
    return mst;
}
```

5.3 Lowest Common Ancestor in a tree

We have a tree T with n nodes and m queries (u,v). For each query (u, v) we want to find the lowest common ancestor(LCA) of u and v. That is, the ancestor of both u and v that has the greatest depth in the tree.

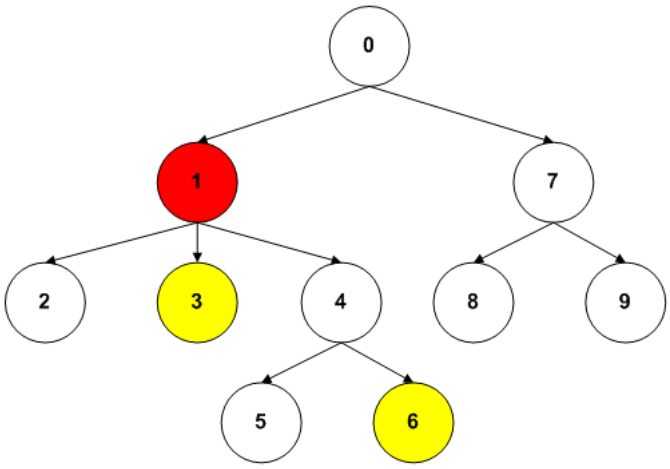

Figure 9: LCA(3,6) = 1

Note that a node v is considered an ancestor of itself, so LCA(u, v) can also be one of the two. The algorithm that we use is offline, meaning that it

computes a solution only if we know all queries beforehand. Let's now see how it works.

The algorithm answers all queries with a single DFS visit. A query (u, v) is answered at node u, if v has already been visited (or vice versa). Let's say we are at node u and want to find LCA(u, v). The answer is either v, or the lowest node among the ancestors of v that is also an ancestor of u. To each ancestor a of v, we assign a set that contains a itself and all the subtrees with roots in the set of its other children that are not in the path to v. The set that contains u is the answer to our query. The main task is maintaining efficiently all these sets, and for this purpose we use Union-Find. We also use an auxiliary vector *ancestor* that will store the real representative of each set [10].

After processing all the children of a node u we answer all the queries(u, v), if v has already been visited. This will be ancestor[*find(u)*]. The full implementation is shown in Listing 8.

Time complexity. Let's say we have n nodes in the tree and m queries. The DFS runs in $O(n)$ time. For each of the n nodes we do a find operation and a union operation. Together with the n makeSet this takes $O(n\alpha(n))$ amortized time. The total time complexity is:

$$O(n\alpha(n) + m)$$

Space complexity. $O(n)$

Listing 8: offline LCA

```
struct Lca
{
    int u, v, ancestor;
    Lca(int u, int v, int ancestor) : u(u), v(v), ancestor(ancestor) {}
};
std::vector<Lca> lca(std::vector<std::vector<int>> &tree,
                     std::vector<std::vector<int>> &queries)
{
    std::vector<int> ancestor(tree.size());
    std::vector<bool> visited(tree.size(), false);
    std::vector<Lca> lca;
    UnionFind uf(tree.size());
    std::function<void(int)> dfs = [&](int v) {
        visited[v] = true;
        ancestor[v] = v;
        for (int u : tree[v])
        {
            if (!visited[u])
            {
                dfs(u);
                uf.set_union(v, u);
                ancestor[uf.find(v)] = v;
            }
        }
        for (int q : queries[v])
        {
            if (visited[q])
            {
                lca.emplace_back(v, q, ancestor[uf.find(q)]);
            }
```

```
        }
    };
    dfs(0);
    return lca;
}
```

5.4 Job Sequencing

There is a list of n jobs, where each job has a deadline d and a profit p if the job is completed before the deadline. Find the sub list of jobs that maximizes the total profit, given that only one job can be scheduled at a certain moment and completing it takes 1 unit of time.

Jobs	J1	J2	J3	J4	J5
Deadlines	2	2	1	3	3
Profits	20	15	1	5	10

Figure 10: A list of jobs with their profit and deadline

A greedy solution [11] sorts the list of jobs by decreasing profit and tries to assign each job to the greatest available slot before the deadline, ignoring it if no slots are found. This algorithm runs in $O(n^2)$.

Lets see now a solution that makes use of Union-Find properties. We start by observing that the crucial aspect is to keep track of the greatest time slot available for a given job. We do this using disjoint sets. The representative of a set is the latest free slot for each job in the set. If a job j is in set 0 it means that no slots can be assigned to it. Finding the representative of the set corresponds to the *find* operation of Union-Find. When we assign a time slot t we merge the set t with the set that contains t-1 so that future queries for time slot t will return the latest time slot that is before t. Let's now describe the algorithm in full.

The first step is sorting the jobs in decreasing order of their profit. After that, we find the maximum deadline m and create $m + 1$ sets, representing the

available time slots. Now we scan the list of jobs and, for each job j, we do the following:

1. find the latest available slot t
2. if $t \neq 0$, assign it to the current job and execute *union(t-1, t)*
3. otherwise, skip this job.

If a job is assigned to slot t, it gets executed from $t-1$ to t. Assigning a job to slot 0 means that it is ignored.

Time complexity. Sorting the jobs takes $O(n \log n)$. A find can take at most $O(\log n)$ and so, the total time complexity is still $O(n \log n)$.

Space complexity. $O(m)$, where m is the maximum deadline.

Listing 9: Job sequencing

```
struct Job
{
    std::string id;
    int profit, deadline;
    Job(std::string id, int profit, int deadline) :
    id(id), profit(profit), deadline(deadline) {}
};

std::vector<Job> jobSequencing(std::vector<Job> jobs)
{
    std::sort(jobs.begin(), jobs.end(),
           [](Job j1, Job j2) { return j1.profit > j2.profit; });

    int maxD = std::max_element(jobs.begin(), jobs.end(),
           [](Job j1) { return j1.deadline; })->deadline;

    std::vector<Job> sol;
    UnionFind uf(maxD + 1);
    for (Job j : jobs)
    {
        int slot = uf.find(j.deadline);
        if (slot > 0)
        {
            uf.set_union(slot - 1, slot);
            sol.push_back(j);
        }
    }

    return sol;
}
```

References

[1] C. Demetrescu, I. Finocchi, and G.F. Italiano. *Algoritmi e strutture dati*. Collana di istruzione scientifica. McGraw-Hill Companies, 2004.

[2] John E. Hopcroft and Jeffrey D. Ullman. Set merging algorithms. *SIAM Journal on Computing*, 2(4):294–303, 1973.

[3] Jan van Leeuwen and R van der Weide. *Alternative path compression techniques*, volume 77. Unknown Publisher, 1977.

[4] Theodorus Petrus van der Weide. *Datastructures: An Axiomatic Approach and the Use of Binomial Trees in Developing and Analyzing Algorithms.* Mathematisch Centrum, 1980.

[5] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.

[6] Robert Clay Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957.

[7] Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová. Otakar boruvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete mathematics*, 233(1-3):3–36, 2001.

[8] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[9] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[10] Jakob Kogler. Lowest common ancestor - tarjan's off-line algorithm. `https://cp-algorithms.com/graph/lca_tarjan.html`, 2018.

[11] GeeksforGeeks. Job sequencing problem. `https://www.geeksforgeeks.org/job-sequencing-problem/`, 2020.