CSS.201.1 Algorithms

Instructor: Umang Bhaskar

TIFR 2024, Aug-Nov

Scribe: Soham Chatterjee

soham.chatterjee@tifr.res.in
Website: sohamch08.github.io

Contents

Chapter 1

Kruskal Algorithm with Data Structure

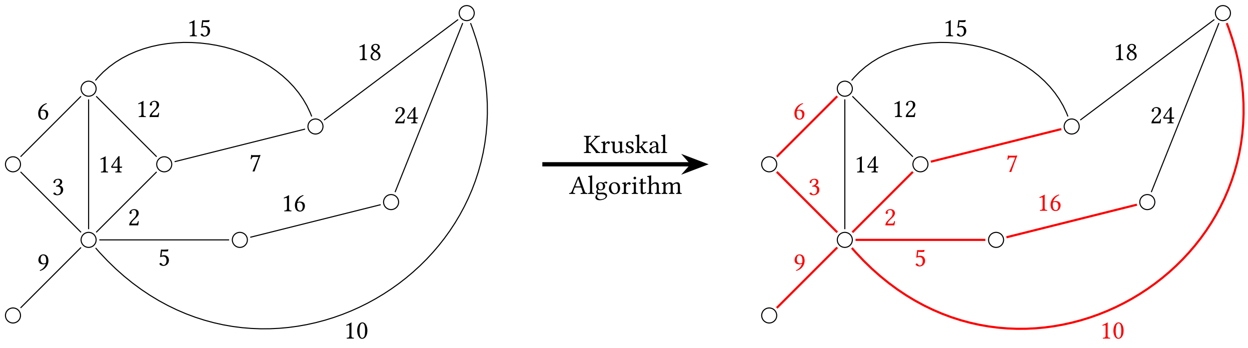
Minimum Spanning Tree

Input: Weighted undirected graph $G = (V, E)$ and weights of edges $W = \{w_e \in \mathbb{Z}_0 : e \in E\}$.

Question: Find a spanning tree $T \subseteq E$ such that $\sum_{e \in T} w_e$ is minimum.

In this chapter we will discuss this problem. We will first discuss the Kruskal algorithm which gives a greedy solution to the problem. Then we will discuss the data structure that we can use to implement the Kruskal algorithm efficiently.

1.1 Kruskal Algorithm



The Kruskal algorithm uses a concept of component to find the minimum spanning tree.

Definition 1.1.1: Component

In a graph $G = (V, E)$, a *component* is a maximal subgraph $G' = (V', E')$ of G such that

(1) (V', E') is connected.

(2) $\forall\, v \notin V'$, there is no edge $e \in E$ such that e connects v to any vertex in V'.

In Kruskal algorithm we maintain a set of components each of them is a tree so basically we maintain a forest. And we find a safe edge which is always the least weight edge in the graph that connects two distinct components and add that edge to the collection of edges in the forest and update the components.

So the algorithm first sorts the edges in non-decreasing order of their weights. Then it initializes a forest F with all the vertices in the graph and no edges. Then it iterates through the sorted edges and checks if the edge connects two distinct components. If it does, then it adds the edge to the forest and merges the two components. The algorithm stops when we have $n-1$ edges in the forest. We have shown in Lemma **??** that the set of collection of acyclic sets in any graph is a matroid. Hence, here we are basically finding a base of the graphic matroid with minimum weight. The algorithm is exactly similar to the greedy algorithm for finding max-weight base of a matroid in **??**. So you can use the similar arguments to show that the algorithm is correct and returns the minimum spanning tree of the graph.

Algorithm 1: Kruskal Algorithm

```
Input: $G = (V, E)$, and weights of edges $W = \{w_e \in \mathbb{Z}_0 : e \in E\}$
Output: A minimum spanning tree $T \subseteq E$ of $G$
begin
    if $G$ is not connected then
        return None                                  // Use DFS or BFS
    $T \longleftarrow \emptyset$
    Sort the edges in $E$ in non-decreasing order of their weights so that $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$
    for $i = 1, \ldots, m$ do
        Let $e_i = (u, v)$
        if $T \cup \{e_i\}$ is acyclic then
            $T \longleftarrow T \cup \{e_i\}$
        if $|T| = |V| - 1$ then
            return $T$
```

Now in the algorithm the checking of $T \cup \{e_i\}$ is acyclic can be done by checking if both the end points are in same component or not. And if they are not then we need to combine those to components. But there comes a question:

Question 1.1

hat it means to give a component?

We will use some vertex to represent the component. We keep a pointer *v.parent* for each vertex which points to representative of component v is in. Hence, we need a data structure that can do the following two operations efficiently:

- Find(u): Returns the component u is in.
- Union(u, v): Merges the components of u and v into a single component.

So we can use the updated algorithm to implement the Kruskal algorithm using proper data structure: The Kruskal Algo-

Algorithm 2: Kruskal Algorithm

```
Input: $G = (V, E)$, and weights of edges $W = \{w_e \in \mathbb{Z}_0 : e \in E\}$
Output: A minimum spanning tree $T \subseteq E$ of $G$
begin
    if $G$ is not connected then
        return None                                  // Use DFS or BFS
    $T \longleftarrow \emptyset$
    Sort the edges in $E$ in non-decreasing order of their weights so that $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$
    for $i = 1, \ldots, m$ do
        Let $e_i = (u, v)$
        if Find$(u) \neq$ Find$(v)$ then
            $T \longleftarrow T \cup \{e_i\}$
            Union$(u, v)$
        if $|T| = |V| - 1$ then
            return $T$
```

rithm calls m times the FIND operation and n times the UNION operation.

1.2 Data Structure 1: Linear Array

We create an n length array A which hold the parent pointer of each vertex. Initially for all vertices $A[v] = v$. So ARRAY-FIND(u) will just return $A[u]$. Hence FIND takes $O(1)$ time. For UNION(u, v) we use the following: Therefore,

Algorithm 3: ARRAY-UNION(u, v)

```
if $A[u] \neq A[v]$ then
    for $i = 1, \dots, n$ do
        if $A[i] == A[v]$ then
            $A[i] \longleftarrow A[u]$
```

ARRAY-UNION(u, v) takes $O(n)$ time. Hence, the time complexity of the Kruskal algorithm using this data structure is $m \cdot O(1) + n \cdot O(n) = O(m + n^2) = O(n^2)$.

1.3 Data Structure 2: Left Child Right Siblings Tree

Using an array is not efficient enough. One place we can optimize is if given the components is there a faster way to get the vertices in the component? We can use the following tricks to optimize:

1. For every representative of a component, store pointers to all vertices in that component.
2. Change representative for the smaller component while doing UNION(u, v).

1.3.1 Construction

So now every representative of a component we point to one vertex which is also in the component. And from that vertex we can iterate through all the vertices in that component. So basically we can immagine a 2 level tree where all the children point towards the root which is the representative of the component. The root points to one of the children take the left most child. And then all the other children points to the immediate right child of the root.

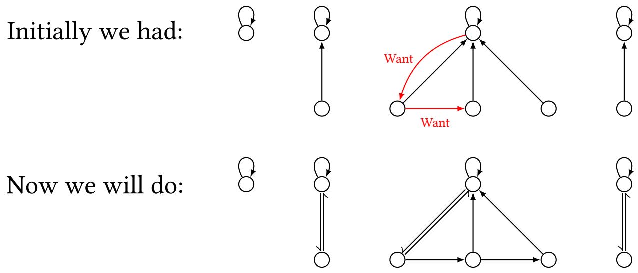


Figure 1.1: Left Child Right Sibling

We can also store a variable to store the number of vertices in the component so that we can use it to compare the size of two components and then update for the smaller one. Therefore, the data structure now stores:

- $v.parent$ for each v which points to the vertex representing the component v is in.
- $v.size$ for size of the component for each component representative v.
- $v.left$ for the left most child for each component representative v.
- $v.right$ for the immediate right sibling of v for all vertices in a component which are not representatives of the components.

This data structure is called Left Child Right Sibling. So in this data structure the LCRS-FIND(u) just returns the value of $u.parent$. Hence, LCRS-FIND takes $O(1)$ time.

1.3.2 LCRS-Union Function

For the LCRS-Union function we do the following

Algorithm 4: LCRS-Union(u, v)

```
$up \longleftarrow u.parent$
$vp \longleftarrow v.parent$
if $up \neq vp$ then
    if $up == u$ then
        $u.parent \longleftarrow vp$
        $u.right \longleftarrow vp.left$
        $vp.left \longleftarrow u$
        $vp.size \longleftarrow vp.size + 1$
    else if $up$.size $\leq vp$.size then
        $up.right \longleftarrow u$
        $x \longleftarrow up$
        while $x$.right == $None$ do
            $x.parent \longleftarrow vp, x \longleftarrow x.right$
        $x.right \longleftarrow vp.left$
        $vp.left \longleftarrow up.left$
        $vp.left \longleftarrow up$
        $vp.size \longleftarrow vp.size + up.size$
    else
        $vp.right \longleftarrow v$
        $x \longleftarrow vp$
        while $x$.right == $None$ do
            $x.parent \longleftarrow up, x \longleftarrow x.right$
        $x.right \longleftarrow up.left$
        $up.left \longleftarrow vp.left$
        $up.left \longleftarrow vp$
        $up.size \longleftarrow up.size + vp.size$
```

Below we have shown how the LCRS-Union function works. This way we can unite two components and update the corresponding component representative in the vertices of the smaller component. In the next section we will analyze the amortized time complexity of the LCRS-Union function

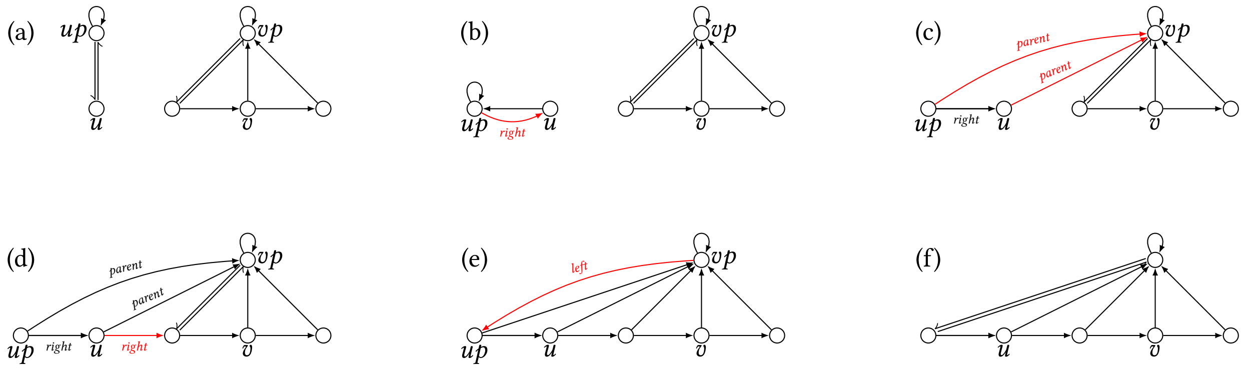


Figure 1.2: A run of LCRS-Union(u, v)

1.3.3 Amortized analysis of LCRS-UNION

Lemma 1.3.1

For any vertex $v \in V$, $v.parent$ can change at most $O(\log n)$ times.

Proof: Initially size of v's component is 1. Each time $v.parent$ is changed the size of the component v is in becomes at least double. Therefore, at most $O(\log n)$ times $v.parent$ can change. ■

Now since there are n vertices at most $O(n \log n)$ times change of *parent* for any vertex happens. Now change of *parent* for any vertex happens only in LCRS-UNION function. Total time taken by all the LCRS-UNION operations is $O(n \log n)$ time. Since LCRS-UNION was called n times the amortized cost of LCRS-UNION is $O(\log n)$.

1.3.4 Time Complexity Analysis of Kruskal

We have shown above that LCRS-FIND takes $O(1)$ time and amortized cost of LCRS-UNION is $O(\log n)$. Since LCRS-FIND is called m times and LCRS-UNION is called n times the total run time of Kruskal Algorithm using the Left Child Right Sibling data structure is $O(m + n \log n)$.

1.4 Data Structure 3: Union Find

We will now give up on the idea of height 1 trees to optimize more. Representative of a component is still vertex at root, but we will do the following: changes:

- UNION just changes parent pointers of root nodes, but it takes $O(1)$.
- Instead of size, we will maintain a variable *rank* of a component roughly which can be thought of as height of the component.
- For UNION root of smaller rank will be changed to point to root of the component of larger rank.
- The FIND operation does something called path compression which we will explain later.

To implement this data structure we will use a tree for each component and every node has a *parent* pointer. And there we will use the FIND and UNION operations.

Our goal is to run Kruskal algorithm in almost $O(n + m)$ time. More precisely $O((n + m) \log^* n)$ time where $\log^* n$ is the number of times we need to compose log on n to get 1.

1.4.1 FIND Operation

The FIND operation does something called path compression i.e. for any vertex $v \in V$, if FIND(v) is called then it starts from v changes it's parent to the root of the component v is in and then it moves to its parent and changes his parent to be the root and move to his parent and keep on doing like that till it reaches the root. In other words in path from the root to v, for every vertex in that path the FIND operation changes the *parent* pointer to the root.

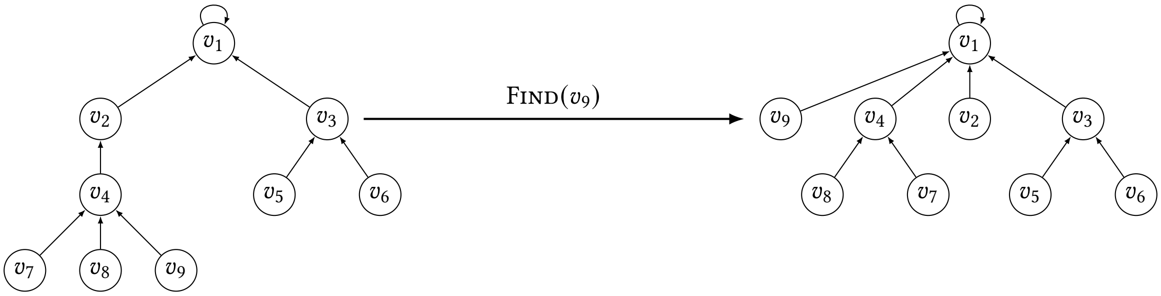


Figure 1.3: Path compression during FIND(v_9) operation

Here is the pseudocode of the FIND operation below: We will not discuss the amortized cost of this operation. Instead, we will do it in subsection 1.4.3.

Algorithm 5: FIND(v)

```
if $v$.parent $\neq v$ then
    $v.parent \longleftarrow$ FIND($v.parent$)
return $v$.parent
```

1.4.2 UNION Operation

For the UNION operation we change the root of smaller rank to point to the root of the component of larger rank. So for the UNION(u,v) operation we assume that u, v are the roots of their respective components. Since in UNION(u,v) we assume u, v are component representatives before using UNION(u,v) we use FIND on u and v to get the roots of their components, and then we apply UNION on the roots. Hence, UNION(u,v) takes $O(1)$ time.

Algorithm 6: UNION(u,v)

```
if $u$.rank > $v$.rank then
    $v.parent \longleftarrow v$
else if $v$.rank > $u$.rank then
    $u.parent \longleftarrow v$
else
    $v.parent \longleftarrow u$
    $v.rank \longleftarrow v.rank + 1$
```

1.4.3 Analyzing the Union-Find Data-Structure

We call a node in the union-find data-structure a *leader* if it is the root of the (reversed) tree.

Lemma 1.4.1

Once a node stop being a leader (i.e. the node in top of a tree). It can never become a leader again.

Proof: A node x stops being a leader only because of the UNION operation which made x child of a node y which is a leader of a tree. From this point on, the only operation that might change the parent pointer of x is the FIND operation which traverses through x. Since path-compression only change the parent pointer of x to point to some other node y. Therefore, the parent pointer of x will never become equal to itself i.e. x can never be a leader again. Hence, once x stops being a leader it can never be a leader again. ■

Lemma 1.4.2

Once a node stop being a leader then its rank is fixed.

Proof: The rank of a node changes only by a UNION operation. But the UNION operation only changes the rank of nodes that are leader after the operation is done. Therefore, once a node stops being a leader it's rank will not being changed by a UNION operation. Hence, once a node stop being a leader then its rank is fixed. ■

Lemma 1.4.3

Ranks are monotonically increasing in the reversed trees, as we travel from a node to the root of the tree.

Proof: To show that the ranks are monotonically increasing it suffices to prove that for all edge $u \longrightarrow v$ in the data structure we have $\text{rank}(u) < \text{rank}(v)$. ■

Lemma 1.4.4

When a node gets rank k than there are at least $\geq 2^k$ elements in its subtree.

Corollary 1.4.5

For all vertices v, $v.rank \leq \lfloor \log n \rfloor$

Corollary 1.4.6

Height of any tree $\leq \left\lfloor \log_2 n \right\rfloor$

Lemma 1.4.7

The number of nodes that get assigned rank k throughout the execution of the Union-Find data-structure is at most $\frac{n}{2^k}$.

Define $N(r) =$ #vertices with rank at least k. Then by the above lemma we have $N(r) \leq \frac{n}{2^k}$.

Lemma 1.4.8

The time to perform a single find operation when we perform union by rank and path compression is $O(\log n)$ time.

We will show that we can do much better. In fact, we will show that for m operations over n elements the overall running time is $O((n+m)\log^* n)$

Lemma 1.4.9

During a single $\textsc{Find}(x)$ operation, the number of jumps between blocks along the search path is $O(\log^* n)$.

Lemma 1.4.10

At most $|Block(i)| \leq Tower(i)$ many FIND operations can pass through an element x which is in the i^{th} block (i.e. $\textsc{index}_B(x) = i$) before $x.parent$ is no longer in the i^{th} block. That is $\textsc{index}_B(x.parent) > i$.

Lemma 1.4.11

There are at most $\frac{n}{Tower(i)}$ nodes that have ranks in the i^{th} block throughout the algorithm execution.

Lemma 1.4.12

The number of internal jumps performed, inside the i^{th} block, during the lifetime of UNION-FIND data structure is $O(n)$.

Theorem 1.4.13

The number of internal jumps performed by the UNION-FIND data structure overall $O(n\log^* n)$.

Theorem 1.4.14

The overall time spent on m FIND operations, throughout the lifetime of a Union-Find data structure defined over n elements is $O((n+m)\log^* n)$.