Analysis of Algorithms, I
CSOR W4231.002

Eleni Drinea
Computer Science Department

Columbia University

The Union Find data structure

Outline

1. Recap: Kruskal's algorithm for MSTs
2. A union-find data structure for disjoint sets
3. Fun combinatorics: #spanning trees in K_n

Today

1. Recap: Kruskal's algorithm for MSTs
2. A union-find data structure for disjoint sets
3. Fun combinatorics: #spanning trees in K_n

Recap

- Minimum Spanning Trees (MSTs)
- The Cut Property and greedy algorithms for MSTs
  - Prim's algorithm
  - Kruskal's algorithm
  - Counting the #MSTs in K_n

Kruskal's algorithm: detailed description

Short description: at every step, add to E_T the lightest edge that does not create a cycle with the edges already in E_T

Alternative view of the algorithm: let $T(v)$ be the tree where vertex v belongs; initially, every vertex forms its own tree.

1. Initialize $E_T = \emptyset$
2. `Sort` the edges by increasing weight
3. For every edge $e = (u, v)$ in order of **increasing weight**:
   - If u and v belong to the same tree, discard e
   - Else $E_T = E_T \cup \{e\}$; merge $T(u)$, $T(v)$ into a single tree

Implementing Kruskal's algorithm

Need a data structure that maintains a **collection of disjoint sets** (trees) and allows

1. to check if u, v belong to the same set (tree);
2. for updates to reflect the merging of two sets (trees) into one

Operations:

1. `MakeSet`(u): Given an element u, create a new set containing only u. Target worst-case time: $O(1)$
2. `Find`(u): Given an element u, find which set u belongs to. Target worst-case time: $O(\log n)$
3. `Union`(u, v): Merge the set containing u and the set containing v into a single set.
   Target worst-case time: $O(\log n)$

Pseudocode

```
Kruskal(G = (V, E, w))
   E_T = ∅
   Sort(E) by w
   for u ∈ V do MakeSet(u)
   end for
   for (u, v) ∈ E by increasing w do
      if Find(u) ≠ Find(v) then
         E_T = E_T ∪ {(u, v)}
         Union(u, v)
      end if
   end for
```

Running time: $O((n+m)\log n)$

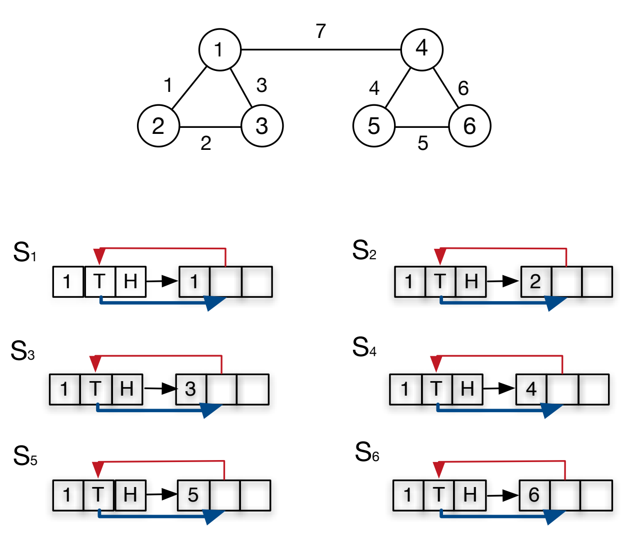
Today

1. Recap: Kruskal's algorithm for MSTs
2. A union-find data structure for disjoint sets
3. Fun combinatorics: #spanning trees in K_n

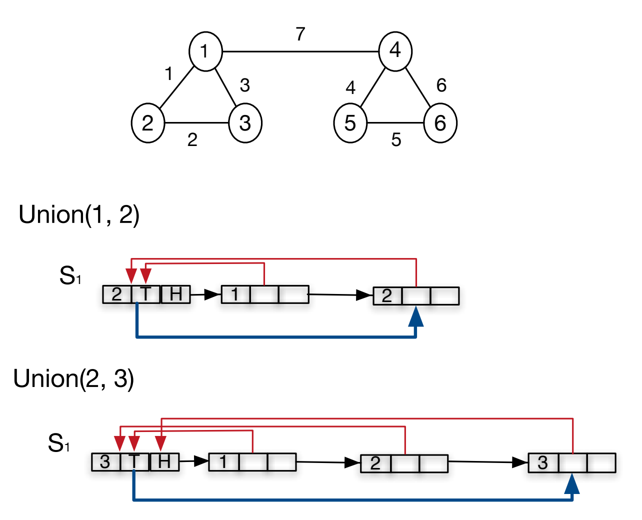
A Union-Find data structure that uses linked lists to represent each set (tree)

- Each set is represented by a linked list.
- The **name of the set** is the name of the first element in the list.
- The **set object** contains:
  1. the **size** of the set: this allows for faster `Union`: assign as the name of the set resulting from `Union`(a, b) the name of the *larger* set (*weighted union heuristic*);
  2. the **T**ail pointer pointing to the last element in the list (this also allows for faster `Union`—*how?*);
  3. the **H**ead pointer pointing to the first element in the list (allows for `Find` in $O(1)$ time).
- Each node in the linked list contains an item, a Head pointer pointing to the set object and a Tail pointer, pointing to the next item in the linked list.

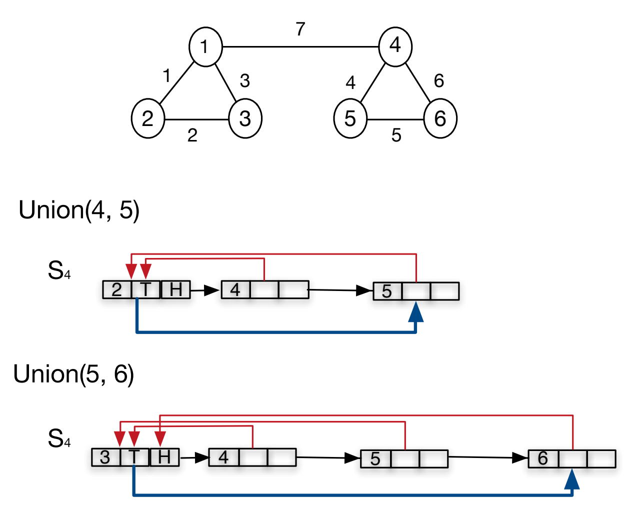
Example: maintaining the data structure during Kruskal's algorithm



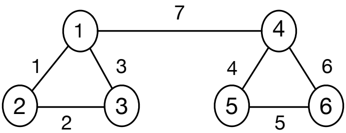
Example: maintaining the data structure during Kruskal's algorithm



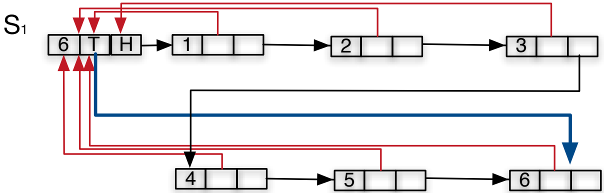
Example: maintaining the data structure during Kruskal's algorithm



Example: maintaining the data structure during Kruskal's algorithm



Union(1, 4)



Amortized analysis

Worst-case analysis: A single `Union` may still require $\Omega(n)$ time *(why?)*.

Idea: Instead of bounding the max time spent on *individual* `Find` and `Union` operations, bound the time spent on a *sequence* of $2m$ `Find` and $n-1$ `Union` operations.

Amortized analysis: consider the entire sequence of $2m$ `Find` and $n-1$ `Union` operations.

- $O(n)$ time to update Tail pointers and size of sets
- $O(n \log n)$ updates of Head pointers for all elements over entire sequence of operations

A tree data structure

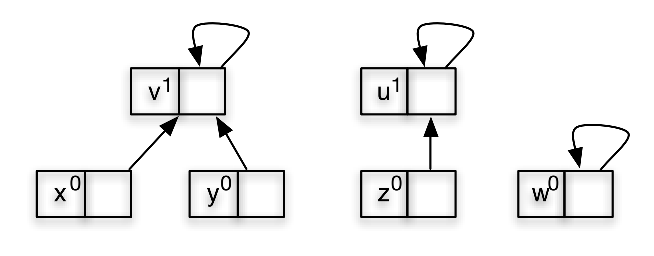
Store a set of elements as a **directed tree**.

- Nodes correspond to elements of the set (no particular order)
- Each node has a *parent* pointer
- If the root of the tree is element r, then the set is assigned the *name* r.
  - The root's *parent* pointer is a self-loop.
- Every node has a *rank*.

$$\begin{aligned} rank(u) &= \text{height of } u\text{'s subtree} \\ &= \textit{\# edges in longest path from a leaf to } u \end{aligned}$$

A set represented as a forest of directed trees

A set of 6 elements $\{u, v, x, y, z, w\}$ maintained by 3 disjoint sets. Here elements $\{x, y\}$ belong to tree v, while z belongs to tree u. The superscript next to each element is its rank.



Operations `MakeSet`(u), `Find`(u)

`MakeSet`(u)

$\pi(u) = u$ $//\pi(u)$ is the parent of u

$rank(u) = 0$

`Find`(u) //returns the *name* of the set where u belongs

while $\pi(u) \neq u$ **do**

$u = \pi(u)$

end while

`return` u

Running time?

Operation `Union`(u, v) constructs the tree

```
Union(u, v)                    //merges the trees where u, v belong
  r_u = Find(u)                //find the root of u's tree
  r_v = Find(v)
  if r_u == r_v then           //if u, v in the same tree, do nothing
    return
  end if
  if rank(r_u) > rank(r_v) then
    π(r_v) = r_u               //make the shorter tree point to the taller
  else
    π(r_u) = r_v
    //if trees equally tall, increase height of resulting tree
    if rank(r_u) == rank(r_v) then
      rank(r_v) = rank(r_v) + 1
    end if
  end if
```

Example: a sequence of `Union` operations

Starting from an empty data structure, make a set for each of the six elements $\{u, v, x, y, z, w\}$ and then perform a sequence of 5 `Union` operations

`Union`(x, v), `Union`(x, y), `Union`(z, u), `Union`(y, u), `Union`(x, w).

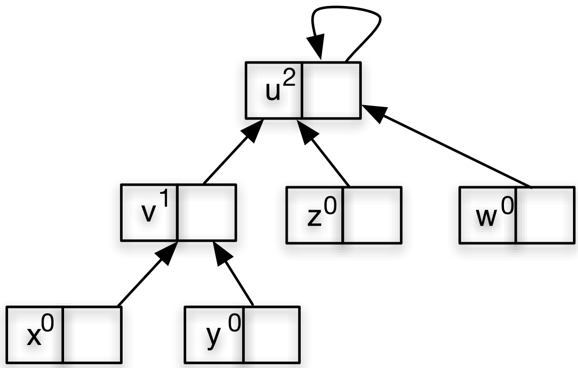
(Break ties by making the alphabetically smaller root the new root.

Example: a sequence of `Union` operations

Starting from an empty data structure, make a set for each of the six elements $\{u, v, x, y, z, w\}$ and then perform a sequence of 5 **`Union`** operations

`Union`(x, v), `Union`(x, y), `Union`(z, u), `Union`(y, u), `Union`(x, w).

(Break ties by making the alphabetically smaller root the new root.



Properties of $rank$

1. How do $rank(u)$, $rank(\pi(u))$ compare?
2. How many ancestors of rank k does an element have?
3. Can subtrees of different rank k nodes overlap?
4. Lower bound on # nodes in a tree whose root has rank k?
5. Lower bound on # nodes in subtree of a node of rank k?
6. How many nodes of rank k can exist?

Properties of $rank$

1. $rank(u) < rank(\pi(u))$ by construction
2. Every element has at most one ancestor of rank k.
3. Subtrees of different rank k nodes are disjoint. (by 1.$\Rightarrow$)
4. # nodes in a tree whose root has rank k: $\geq 2^k$ (by induction)
5. # nodes in the subtree of a node of rank k: $\geq 2^k$
6. If x nodes of rank k, then $\geq x \cdot 2^k$ nodes in the x subtrees.

Max tree height and worst-case running time

Therefore, if we have n elements in total,

- $x \cdot 2^k \leq n \Rightarrow$ at most $\frac{n}{2^k}$ nodes of rank k
- the maximum rank is $\log_2 n$

Thus *max tree height* $= \log_2 n$

Hence worst-case running time for `Find`, `Union` $= O(\log_2 n)$, and Kruskal's algorithm takes $O((n+m)\log_2 n)$ time.

What if edges are already sorted?

What if edge weights are already sorted?

Or, they are small enough, e.g., $w(e) < m$ *for all* $e \in E$*, so they can be sorted in linear time (e.g., using Bucketsort)?*

Then the data structure is the *bottleneck* for the performance of Kruskal's algorithm.

Goal: design a data structure that allows for linear (or almost linear) running time

Maintenance operations that pay off in the long run

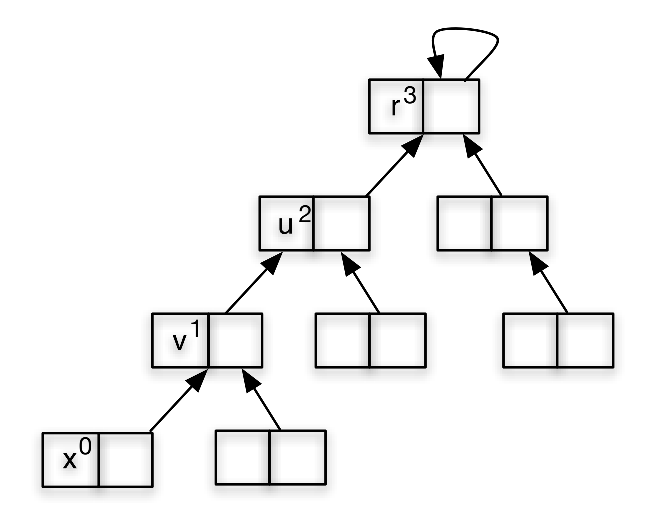
Goal: maintain *short* trees since the time for `Find`(u) corresponds to u's depth in the tree

Heuristic idea: when performing `Find`(u), update the parent pointers of **every** node x on the u-r path to point to r.

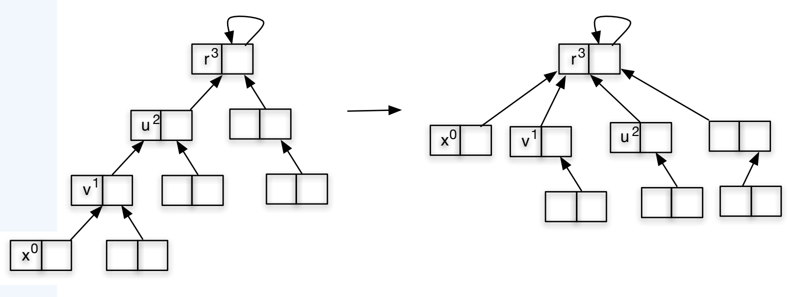
Why? All future `Find`(x) will start from much closer to the root (although x might not point to the root anymore –*why?*).

This motivates a different kind of analysis: consider sequences of `Find` and `Union` operations, and look at the **average** time spent per operation (amortized analysis).

Find(x)



$\texttt{Find}(x)$ with path compression



`Find` with **path compression**

//returns the *name* of the set where u belongs
//sets every node on the u-r path to point to r

`Find`(u)

    while $\pi(u) \neq u$ **do**

        $\pi(u) = \texttt{Find}(\pi(u))$

    end while

    `return` $\pi(u)$

Remark 1.

1. *This procedure makes* **two passes** *on the find path.*
2. *It does not change the ranks of the nodes. However, the rank of a node no longer corresponds to the height of its subtree.*

Roadmap

- A `Find` may still take $O(\log n)$ time (exercise).
- Instead of bounding the max time spent on *individual* `Find` operations, bound the time spent on a *sequence* of m `Find` operations.
- If we perform a total of m `Find` operations, we want to spend linear or almost linear time for all of them.

Roadmap cont'd

1. Partition the nodes in a small number of carefully designed groups, depending on their ranks.

2. Recall that `Find`(u) traverses a sequence of pointers from u to the root r. Think of each pointer as belonging to **one of two** different types of pointers.

   - *The type of the pointer from x to $\pi(x)$ will be determined by the groups of x and $\pi(x)$, for x on the u-r path.*
   - Directly bound the time t_1 spent by a single `Find` operation on pointers of type 1.
     - $\Rightarrow$ Total time spent by all `Find`'s on pointers of type 1 is mt_1.
   - Carefully bound the total time spent by all m `Find`'s on pointers of type 2.

1. Partitioning nodes into groups

If there are n nodes, their ranks range from 0 to $\log n$.

Divide the nonzero ranks into groups as follows:

1. Group 0: $[1]$ $[0+1, 2^0]$
2. Group 1: $[2]$ $[1+1, 2^1]$
3. Group 2: $[3, 4]$ $[2+1, 2^2]$
4. Group 3: $[5, 16]$ $[4+1, 2^4]$
5. Group 4: $[17, 2^{16}]$ $[16+1, 2^{16}]$
6. Group 5: $[65537, 2^{65537}]$ $[65536+1, 2^{65537}]$
7. ...

#nodes in group $[k+1, 2^k]$

$\log^* n$ = #iterations of the $\log_2$ function on n until we get a number less than or equal to 1

Examples: $\log^* 4 = 2, \log^* 16 = 3$

- Group i is of the form $(2^{i-1}, 2^{2^{i-1}}]$ (except for group 0).
- For simplicity, denote groups by $[k+1, 2^k]$.
- Total # groups: $\leq \log^* n$ (*why?*)
- For all practical purposes, $\log^* n \leq 5$ —else, $n \geq 2^{65537}$!

Fact 1.

There are at most $\frac{n}{2^k}$ nodes in group $[k+1, 2^k]$.

Creative accounting

Idea: assign 2^k dollars (corresponding to units of time) to every node in group $[k+1, 2^k]$.

By Fact 1, we are spending at most extra $n \log^* n$ dollars for all nodes (this amount is "linear" in n).

We will spend these dollars to pay for the work required by `Find` operations that follow pointers between nodes whose ranks belong to the same group.

Types of pointers in a `Find` operation

Let $v = \pi(u)$. Recall that $\texttt{Find}(u)$ follows a sequence of pointers. We distinguish between two types of pointers.

1. Type 1: a pointer is of Type 1 if u and v belong to different groups, or if v is the root.
2. Type 2: a pointer is of Type 2 if u and v belong to the same group.

We *account* for the two types of pointers in two different ways:

1. Type 1 pointers are charged directly to the `Find` operation
2. Type 2 pointers are charged to u, who pays using its pocket money

Counting the work spent on `Find`(u) operations

Suppose u belongs to group $[k+1, 2^k]$. Let $v = \pi(u)$.

1. Type 1 pointers: charged directly to the `Find` operation
   At most $t_1 = \log^* n$ pointers of Type 1 in **each `Find`** operation.

2. Type 2 pointers: recall that every node with rank in group $[k+1, 2^k]$ is given 2^k dollars (units of time).
   u pays a dollar for each of them using its pocket money.

Does u have enough money to pay for the Type 2 pointers in all m `Find` *operations?*

u's allowance suffices for all m operations

Recall that

- both u, v are in group $[k+1, 2^k]$;
- each `Find`(u) causes u to pay a dollar.

Key observation: each `Find`(u) causes $\pi(u)$ to point to the root of u's tree; so $rank(v)$ increases by at least 1 (of course, $rank(u)$ does not change).

How many times can v's rank increase before u and v are in different groups?

Fewer than 2^k.

Summary

Suppose we perform $2m$ `Find` operations.

- Each `Find` is charged at most $t_1 = \log^* n$ dollars.
  Hence all $2m$ `Find` require at most $O(m \log^* n)$ time.
- We spend at most extra $n \log^* n$ dollars.

$\Rightarrow$ The total amount of time spent for a sequence of $2m$ `Find` and $n-1$ `Union` operations is

$$O((n+m)\log^* n).$$

$\Rightarrow$ **On average**, every `Find` operation takes $\log^* n$ time.

Final remarks

- **Amortized analysis:** a tighter *worst-case* analysis
- This is **not** average case analysis: no probability is involved; rather, the *average* cost of an operation is shown small, *averaged* over a sequence of operations (so a few individual operations may still be costly)
- Other uses of Union-Find: maintain SCCs in dynamic graphs

Today

1. Recap: Kruskal's algorithm for MSTs
2. A union-find data structure for disjoint sets
3. Fun combinatorics: #spanning trees in K_n

Cayley's formula

- Let K_n be the complete graph on n vertices.
- Let $T_n = \#$ spanning trees in K_n
- Cayley's formula: $T_n = n^{n-2}$
- Proof: via computing a quantity in two different ways to derive an expression for T_n.

Proof of Cayley's formula: counting in 2 different ways

Recall that a directed tree is a **rooted** graph that has a simple path from the root to every vertex in the graph

- A tree has $n - 1$ edges.

The quantity we will compute in two different ways is the number ν of different *sequences of directed edges* that can be added to an empty graph on n vertices to yield a rooted tree.

1. A formula for ν that directly involves T_n

1. Start with a spanning tree on the empty graph (T_n choices)
2. Pick a root for the tree (n choices)
3. Given the root, the direction of every edge is fully determined *(why?)*

$\Rightarrow$ There are $n-1$ directed edges to insert in **any order** in our graph ($(n-1)!$ ways to order them)

In total, there are $T_n \cdot n \cdot (n-1)!$ different sequences of directed edges to add in a graph so as to form a directed rooted tree; so

$$\nu = T_n \cdot n \cdot (n-1)!$$

2. Computing ν directly

Start with an empty graph on n nodes.
We will add $n-1$ directed edges **one by one** so that we construct a rooted tree spanning the n nodes.

1. At every step $i = 1, 2, \ldots, n-1$, let n_i be the #possible directed edges from which to choose the edge to add.

2. Then the #different sequences of directed edges that yield a rooted tree is simply the product of all n_i.

Adding the first directed edge

#possible directed edges from which to choose at every step:

- An edge is completely defined when its tail and head are picked.
- Hence the #possible directed edges at every step is

$$(\#\text{ways to choose a tail}) \cdot (\#\text{ways to choose a head})$$

Initially, we have a forest of n empty rooted trees.

1. Adding the 1st edge:
   - tail: pick **any** of the n vertices
   - head: direct the edge to **any** of the $n-1$ *other* roots

   $\Rightarrow$ $\alpha_1 = n(n-1)$ ways to choose the 1st edge

Adding the second directed edge

The graph is now a forest with $n-1$ rooted trees.

2. Choosing the 2nd edge:
   - tail: pick **any** of the n vertices
   - direct the edge to the **root** of any tree (so that the resulting graph remains a rooted tree) **except** for the tree where the tail belongs (*why?*)

   $\Rightarrow$ $\alpha_2 = n(n-2)$ ways to choose the 2nd edge

Adding the k-th edge

k. k-th edge: the reasoning is entirely similar. After addition of the $(k-1)$-st edge, there are $n-(k-1)$ rooted trees in the forest (by construction, every edge we add reduces the number of trees by 1).

- pick **any** of the n vertices as the tail of the edge
- direct the edge to the **root** of any tree in the **except** for the tree where the tail belongs

$\Rightarrow$ $\alpha_k = n(n-k)$ ways to choose the k-th edge

n-1. $n-1$-st edge: $\alpha_{n-1} = n \cdot 1$ ways to choose the $n-1$-st edge

Conclusion

- In total, there are $\prod_{i=1}^{n} \alpha_i = n^{n-1}(n-1)!$ ways to add the edges. Hence
$$\nu = n^{n-1}(n-1)!$$
- Equating the two expressions for ν, we obtain:
$$T_n = n^{n-2}$$
- Arbitrary graphs: #spanning trees computable in polynomial time