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 $K_n$
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: $O(\log n)$
Kruskal\((G = (V, E, w))\)

\(E_T = \emptyset\)

Sort\((E)\) by \(w\)

for \(u \in V\) do MakeSet\((u)\)
end for

for \((u, v) \in E\) by increasing \(w\) do
    if Find\((u) \neq \text{Find}(v)\) then
        \(E_T = E_T \cup \{(u, v)\}\)
        Union\((u, v)\)
    end if
end for

Running time: \(O((n + m) \log n)\)
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 tree data structure

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

- Nodes correspond to set elements (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*.

\[
\text{rank}(u) = \text{height of } u\text{'s subtree} = \# \text{ edges in longest path from a leaf to } u
\]
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 \texttt{MakeSet}(u), \texttt{Find}(u)

\texttt{MakeSet}(u)
\begin{align*}
\pi(u) &= u \quad \text{//}\pi(u) \text{ is the parent of } u \\
\text{rank}(u) &= 0
\end{align*}

\texttt{Find}(u) \quad \text{//returns the \textit{name} of the set where } u \text{ belongs}
\begin{align*}
\textbf{while } \pi(u) \neq u \text{ do} \\
&\quad u = \pi(u) \\
\textbf{end while} \\
\text{return } u
\end{align*}

Running time?
Operation \texttt{Union}(u, v) \text{ constructs the tree}

\begin{align*}
\text{Union}(u, v) & \quad \text{//merges the trees where } u, v \text{ belong} \\
r_u = \text{Find}(u) & \quad \text{//find the root of } u \text{'s tree} \\
r_v = \text{Find}(v) & \\
\textbf{if } r_u == r_v \textbf{ then} & \quad \text{//if } u, v \text{ in the same tree, do nothing} \\
& \quad \text{return} \\
\textbf{end if} \\
\textbf{if } \text{rank}(r_u) > \text{rank}(r_v) \textbf{ then} & \quad \text{//make the shorter tree point to the taller} \\
& \quad \pi(r_v) = r_u \\
\textbf{else} & \\
& \quad \pi(r_u) = r_v \\
& \quad \text{//if trees equally tall, increase height of resulting tree} \\
\textbf{if } \text{rank}(r_u) == \text{rank}(r_v) \textbf{ then} & \quad \text{rank}(r_v) = \text{rank}(r_v) + 1 \\
& \quad \text{end if} \\
\textbf{end if} \\
\end{align*}
Example: a sequence of \texttt{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 \texttt{Union} operations \texttt{Union}(x, v), \texttt{Union}(x, y), \texttt{Union}(z, u), \texttt{Union}(y, u), \texttt{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

\[
\text{Union}(x, v), \text{Union}(x, y), \text{Union}(z, u), \text{Union}(y, u), \text{Union}(x, w).
\]

(Break ties by making the alphabetically smaller root the new root.)
Properties of \textit{rank}

1. How do \textit{rank}(u), \textit{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 \textit{rank}

1. $\text{rank}(u) < \text{rank}(\pi(u))$ by construction
2. Every element has \textbf{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.
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 $\text{max tree height} = \log_2 n$

Hence worst-case running time for $\text{Find}$, $\text{Union} = O(\log_2 n)$, 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
Goal: maintain short trees since the time for $\text{Find}(u)$ corresponds to $u$’s depth in the tree

Heuristic idea: when performing $\text{Find}(u)$, update the parent pointers of every node $x$ on the $u$-$r$ path to point to $r$.

Why? All future $\text{Find}(x)$ 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 $\text{Find}$ and $\text{Union}$ operations, and look at the average time spent per operation (amortized analysis).
Find($x$)
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 π(u) ≠ u do
        π(u) = Find(π(u))
    end while

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

2. Recall that \textbf{Find}(u) \textbf{traverses} a sequence of pointers from \textit{u} 
to the root \textit{r}. \textbf{Think of each pointer as belonging to one of} 
two different \textbf{types of pointers}.

\begin{itemize}
  \item \textit{The type of the pointer from} \textit{x} \textbf{to} \textit{π(x)} \textbf{will be determined by} 
    \textbf{the groups of} \textit{x} \textbf{and} \textit{π(x)}, \textbf{for} \textit{x} \textbf{on the} \textit{u-r} \textbf{path}.
  \item Directly bound \textit{the time} \textit{t_1} \textbf{spent by a single Find operation} 
    \textbf{on} \textbf{pointers of type 1}.
    \Rightarrow \textit{Total time spent by all Find’s on pointers of type 1 is} \textit{mt_1}.
  \item Carefully bound \textbf{the total time} \textbf{spent by all} \textit{m Find’s} \textbf{on} 
    \textbf{pointers of type 2}.
\end{itemize}
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 \([k + 1, 2^k]\)

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

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

- Group \(i\) is of the form \((2^{i-1}, 2^{2i-1}]\) (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 \([k + 1, 2^k]\).*
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 \textbf{Find} operations that follow pointers between nodes whose ranks belong to the same group.
Let $v = \pi(u)$. Recall that $\text{Find}(u)$ 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 $\text{Find}$ operation.
2. **Type 2** pointers are charged to $u$, who pays using its pocket money.
Suppose $u$ belongs to group $[k + 1, 2^k]$. Let $v = \pi(u)$.

1. **Type 1** pointers: charged directly to the $\text{Find}$ operation
   At most $t_1 = \log^* n$ pointers of Type 1 in each $\text{Find}$ 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 $m$ $\text{Find}$ operations?*
Recall that

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

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

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

Fewer than $2^k$. 

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 $K_n$
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 $T_n$. 
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 $\nu$ of different *sequences of directed edges* that can be added to an empty graph on $n$ vertices to yield a rooted tree.
1. Start with a spanning tree on the empty graph \((T_n \text{ choices})\)
2. Pick a root for the tree \((n \text{ choices})\)
3. Given the root, the direction of every edge is fully determined \((why?)\)
\[\Rightarrow \] There are \(n - 1\) directed edges to insert in \textbf{any order} in our graph \(((n - 1)! \text{ ways to order them})\)

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 $\nu$ 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 number of possible directed edges from which to choose the edge to add.

2. Then the number of 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
   \[ \Rightarrow \alpha_1 = n(n - 1) \text{ ways} \text{ to choose the 1st edge} \]
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?**)  
   \[ \Rightarrow \alpha_2 = n(n - 2) \text{ ways to choose the 2nd edge} \]
**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
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 \( \nu \), we obtain:
\[
T_n = n^{n-2}
\]

Arbitrary graphs: \#spanning trees computable in polynomial time