Greedy algorithms
and Union-Find

CS218, Fall 2016

Outline

• Intro
• Activity selection
• Dijkstra (single source shortest path)
• Prim and Kruskal (minimum spanning tree)
• Union-Find
Intro

Greedy method

- Typically applied to *optimization problems*, that is, problems that involve searching through a set of *configurations* to find one that minimizes/maximizes an *objective function* defined on these configuration

- *Greedy strategy*: at each step of the optimization procedure, choose the configuration which seems the best between all of those possible
Greedy method

- There are problems for which the globally optimal solution can be found by making a series of locally optimal (greedy) choices
  - Make whatever choice seems best at the moment and then solve the sub-problem arising after the choice is made
  - The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to sub-problems
- The greedy strategy does not always lead to the global optimal solution

Elements of Greedy Strategy

- Two ingredients that are exhibited by most problems that lend themselves to a greedy strategy
  - Greedy-choice property: a globally optimal solution can be reached by making a locally optimal choice
  - Optimal substructure: optimal solution to the problem consists of optimal solutions to sub-problems
An activity-selection problem

(aka, “task scheduling” problem)

An Activity Selection Problem

- **Input**: A set of activities \( S = \{a_1, \ldots, a_n\} \)
- Each activity has start time and a finish time \( a_i = (s_i, f_i) \)
- Two activities are compatible if and only if their interval does not overlap
- **Output**: a maximum-size subset of mutually compatible activities
An Activity Selection Problem

- Here are a set of tasks (start time, finish time):

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>s_i</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>f_i</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

- What is the maximum number of activities that can be completed?
  - \( \{a_3, a_9, a_{11}\} \) can be completed
  - But so can \( \{a_4, a_8, a_{11}\} \) which is a larger set
  - But it is not unique, consider \( \{a_2, a_4, a_9, a_{11}\} \)
“Greedy” Strategies

1. Longest first
2. Shortest first
3. Early start first
4. Early finish first
5. None of the above
Early Finish Greedy strategy

- Sort the activities by finish time
- Schedule the first activity
- Then, schedule the next activity (in sorted list) which starts after previous activity finishes (first non-conflicting task)
- Repeat until no more activities
Activity selection in Python

```python
def greedy_activity_selection(A):
    A.sort(key=itemgetter(1))  # Remark: sort A by finish time
    result = [A[0]]  # Remark: first activity in the solution
    i = 0
    for j in range(1, len(A)):
        if A[j][0] >= A[i][1]:
            result.append(A[j])  # Remark: start[j] >= finish[i]
            i = j
    return result
```

Time complexity? $O(n \log n)$ to sort, the rest is linear.

Why it is Greedy?

- Greedy in the sense that it leaves as much opportunity as possible for the remaining activities to be scheduled

- The greedy choice is the one that maximizes the amount of unscheduled time remaining
Correctness (optimality)

• We will show that
  • the problem has the optimal substructure property
  • the algorithm satisfies the greedy-choice property
  • Thus, the algorithm always finds the optimal solution

Greedy-Choice Property

• We want to show there is an optimal solution that begins with a greedy choice (i.e., with activity 1, which has the earliest finish time)
Greedy-Choice Property

• Suppose \( A \subseteq S \) is an optimal solution
  – Order the activities in \( A \) by finish time
    Let \( k \) be the first activity in \( A \)
    • If \( k = 1 \), the schedule \( A \) begins with a greedy choice
    • If \( k \neq 1 \), show that there is another optimal solution \( B \) that begins with the greedy choice (activity 1)
  – Let \( B = A - \{k\} \cup \{1\} \)
    • Activities in \( B \) are non-conflicting because activities in \( A \) are non-conflicting, \( k \) is the first activity to finish and \( f_1 \leq f_k \)
    • \( B \) has the same number of activities as \( A \) thus, \( B \) is optimal

Optimal Substructure

• Once the greedy choice of the first activity is made, the problem reduces to finding an optimal solution for the activity-selection problem over those activities in \( S \) that are compatible with the first activity
  – Optimal Substructure: if \( A \) is optimal to \( S \), then \( A' = A - \{1\} \) is optimal to \( S' = \{i \in S: s_i \geq f_1\} \)
  – Why? If we could find a solution \( B' \) to \( S' \) with more activities than \( A' \), adding activity 1 to \( B' \) would yield a solution \( B \) to \( S \) with more activities than \( A \) contradicting the optimality of \( A \)
Optimal Substructure

- After each greedy choice is made, we are left with an optimization problem of the same form as the original problem.

- By induction on the number of choices made, making the greedy choice at every step produces an optimal solution.

Dijkstra (single-source shortest path)
Shortest Path

• Let $G$ be a weighted graph ($w(e)$ is the weight of the edge $e$)

• The length of a path $P$ is the sum of the weights of the edges of $P$

• If $P=e_0, e_1, ..., e_{k-1}$ then the length of $P$ is $\sum w(e_i)$

Single-Source Shortest Path

• The distance from a vertex $u$ to vertex $v$, denoted by $\delta(u,v)$ is the length of a minimum length path (also called shortest-path) from $u$ to $v$, if such a path exists

• If the path does not exists, $\delta(u,v)=+\infty$

• Note that if there is a negative cycle, then the distance may not be defined
Optimal Substructure

• **Fact:** subpaths of shortest paths are shortest paths
• **Proof:** decompose a shortest path 
  \( p = \langle v_1, v_2, ..., v_k \rangle \) into \( v_i \rightarrow v_j \rightarrow v_k \). Then 
  \( w(p) = w(v_i, v_j) + w(v_j, v_k) \).
  If \( v_i \rightarrow v_j \) is not optimal, then we could make the path 
  \( v_i \rightarrow v_k \) shorter, which contradicts the optimality of \( p \).

Shortest-Path Problems

• **Single-source (single-destination):** Find a shortest path from a given source (vertex \( s \)) to all the other vertices → **greedy**
• **All-pairs:** Find shortest-paths for every pair of vertices → **dynamic programming**
• **Special cases**
  - Unweighted shortest-paths → **BFS**
  - Shortest path on a DAG → **topological sorting**
Dijkstra’s Algorithm

• Computes shortest paths from a start vertex $s$ to all the other vertices
• Works on a simple graph with non-negative weights
• Computes for each vertex $u$ the distance to $u$ from the start vertex $s$, that is, the weight of a shortest path between $s$ and $u$
• Keeps track of the set of vertices for which the distance has been computed, called the cloud $S$

Dijkstra’s Algorithm

• Every vertex has a label associated with it
• For any vertex $u$, we can refer to its “d label” as $d[u]$
• $d[u]$ stores an approximation of $\delta(s,u)$
• The algorithm will update a $d[u]$ value when it finds a shorter path from $s$ to $u$
Dijkstra’s Algorithm

• When a vertex \( u \) is added to the cloud, its label \( d[u] \) is equal to the actual (final) distance between the starting vertex \( s \) and vertex \( u \)
• Initially, we set
  - \( d[s]=0 \) ...the distance from \( s \) to itself is 0...
  - \( d[u]=\infty \) for \( u \neq s \) ...these will change...

Edge relaxation

• For each vertex \( v \) in the graph, we maintain in \( d[v] \) the estimate of the shortest path from \( s \)
• Relaxing an edge \((u, v)\) means testing whether we can improve the shortest path to \( v \) found so far by going through \( u \)

Observe that after the relaxation of \((u,v)\), \( d[v] \leq d[u] + w(u,v) \)
Expanding the Cloud

• Repeat until all vertices have been put in the cloud
  – let \( u \) be a vertex not in the cloud that has smallest \( d[u] \)
    (on the first iteration, the starting vertex will be chosen)
  – we add \( u \) to the cloud \( S \)
  – we update \( d[.] \) of the adjacent vertices of \( u \) as follows
    (edge relaxation)
    
    for each vertex \( z \) adjacent to \( u \) do
      if \( z \) is not in the cloud \( S \) then
        if \( d[u] + \text{weight}(u,z) < d[z] \) then
          \( d[z] \leftarrow d[u] + \text{weight}(u,z) \)

Dijkstra’s

Algorithm ShortestPath\((G,v)\):

Input: A simple undirected weighted graph \( G \) with nonnegative edge weights,
       and a distinguished vertex \( v \) of \( G \)

Output: A label \( D[u] \), for each vertex \( u \) of \( G \), such that \( D[u] \) is the distance from
         \( v \) to \( u \) in \( G \)

Initialize \( D[v] \leftarrow 0 \) and \( D[u] \leftarrow +\infty \) for each vertex \( u \neq v \).

Let a priority queue \( Q \) contain all the vertices of \( G \) using the \( D \) labels as keys.

while \( Q \) is not empty do
  {pull a new vertex \( u \) into the cloud}
  \( u \leftarrow Q.\text{removeMin()} \)
  for each vertex \( z \) adjacent to \( u \) such that \( z \) is in \( Q \) do
    {perform the relaxation procedure on edge \((u,z)\)}
    if \( D[u] + \text{weight}(u,z) < D[z] \) then
      \( D[z] \leftarrow D[u] + \text{weight}(u,z) \)
      Change to \( D[z] \) the key of vertex \( z \) in \( Q \).
return the label \( D[u] \) of each vertex \( u \)
Time complexity

- Use a heap-based priority queue $Q$ to store the vertices not in the cloud, where $d[u]$ is the key of a vertex $u$ in $Q$
- Insert all vertices in $Q$, takes $O(n \log n)$
- Each iteration of the while, we spend $O(\log n)$ time to remove vertex $u$ from $Q$ and $O(\deg(u) \log n)$ to perform the relaxation step
- Overall, $O(n \log n + \sum_v (\deg(v) \log n))$ which is $O((n+m) \log n)$ [using binary heaps]
- FYI: using Fibonacci heaps, Dijkstra runs in $O(m+n \log n)$

Greedy choice

- **Theorem:** In Dijkstra’s algorithm, whenever a vertex $u$ is pulled into $S$, the label $d[u]$ is equal to $\delta(s,u)$ (the length of a shortest path from $s$ to $u$), and that equality is maintained thereafter
Upper-bound property

• Lemma: For all $v$ in $V$, $d[v] \geq \delta(s,v)$
• Proof: by induction on the number of relaxation steps.
• Base case: true at initialization (zero relaxations).
• Induction step: Let us consider the relaxation of edge $(u,v)$. By inductive hypothesis we have $d[x] \geq \delta(s,x)$ for all the nodes $x$ prior to the relaxation step. If $d[v]$ changes, we have $d[v] = d[u] + w(u,v) \geq \delta(s,u) + w(u,v) \geq \delta(s,v)$ thus the invariant is maintained (middle inequality due to the inductive hypothesis, the last one is due to triangle inequality).

Convergence property

• Lemma: If $s \rightarrow (u,v)$ is a shortest path and $d[u] = \delta(s,u)$, when we relax edge $(u,v)$ we have $d[v] = \delta(s,v)$.
• Proof: By the upper-bound property if $d[u] = \delta(s,u)$ at some point before relaxing $(u,v)$, then this equality holds thereafter. After relaxing edge $(u,v)$ $d[v] \leq d[u] + w(u,v) = \delta(s,u) + w(u,v) = \delta(s,v)$ (the first inequality is due to the RELAX code, the last equality is due to optimal substructure).

Since $d[v] \geq \delta(s,v)$ we must have $d[v] = \delta(s,v)$. 

51

52
Proof of Theorem (by contradiction)

• By the upper bound lemma the only way Dijkstra can be “wrong” is that $d[u] > \delta(s,u)$
• Let $u$ be the first vertex pulled in $S$ such that there is a path shorter than $d[u]$, i.e., $d[u] > \delta(s,u)$
• We will show that this leads to a contradiction

Proof of Theorem

• Let $y$ be the first vertex outside $S$ on the actual shortest path from $s$ to $u$ ($y$ could be $u$)
• Let $x$ be the predecessor of $y$ ($x$ could be $s$)
• Then it must be that $d[y] = \delta(s,y)$ because
  – the label $d[x]$ is set correctly because $x$ is in $S$ and $u$ is the first vertex for which $d$ is set incorrectly
  – when the algorithm pulled $x$ into $S$, the algorithm relaxed the edge $(x,y)$, setting $d[y]$ to the correct value (due to Convergence lemma)
Proof of Theorem

\[ d[u] > \delta(s,u) \] (initial assumption)
\[ = \delta(s, y) + \delta(y, u) \] (optimal substructure)
\[ = d[y] + \delta(y, u) \] (correctness of \( d[y] \))
\[ \geq d[y] \] (no negative weights)

- But if algorithm has chosen \( u \) to be next in \( S \), not \( y \) then \( d[u] \leq d[y] \)
- Thus, \( d[y] = \delta(s, y) = \delta(s, u) = d[u] \) at time of insertion of \( u \) into \( S \) (contradicts \( d[u] > \delta(s, u) \))
- Dijkstra’s algorithm is correct

Kruskal (minimum spanning tree)
Minimum Spanning Tree

- Given a weighted undirected graph $G$, find a tree $T$ that spans all the vertices of $G$ and minimizes the sum of the weights on the edges, that is
  \[ w(T) = \sum_{e \in T} w(e) \]

- We want a spanning tree of minimum cost

Example

\[ w(T) = 4 + 8 + 7 + 9 + 2 + 4 + 2 + 1 = 37 \]

Note that the MST is not necessarily unique

For example, add $(a,h)$, delete $(b,c)$
Growing a MST: Generic algorithm

- Grow MST one edge at a time
- Manage a set of edges $A$, maintaining the following invariant
  - prior to each iteration, $A$ is a subset of some MST
- At each iteration, we determine an edge $(u,v)$ that can be added to $A$ without violating this invariant
- If $A \cup \{(u,v)\}$ is also a subset of a MST, then $(u,v)$ is called a safe edge for $A$

Generic MST algorithm

```plaintext
GENERIC-MST(G, w)
1  A ← ∅
2  while A does not form a spanning tree
3      do find an edge $(u, v)$ that is safe for A
4          A ← A ∪ {(u, v)}
5  return A
```

- Loop in lines 2-4 is executed $|V| - 1$ times because any MST tree contains $|V| - 1$ edges
- The overall execution time depends on how to find a safe edge (step 3)
Greedy Choice

- **Definitions**
  - **Cut** $(S, V-S)$: a partition of $V$
  - **Crossing edge**: one endpoint in $S$ and the other in $V-S$
  - A cut respects a set of $A$ of edges if no edges in $A$ crosses the cut
  - A **light edge** crossing a partition if its weight is the minimum of any edge crossing the cut

- **Theorem.** Let $A$ be a subset of $E$ that is included in some MST of $G=(V,E)$. Let $(S, V-S)$ be any cut of $G$ that respects $A$, and let $(u,v)$ be a light edge crossing $(S, V-S)$. Then, edge $(u,v)$ is safe for $A$.

---

**Examples of Cuts and light edges**

**Figure 23.2** Two ways of viewing a cut $(S, V-S)$ of the graph from Figure 23.1. (a) The vertices in the set $S$ are shown in black, and those in $V-S$ are shown in white. The edges crossing the cut are those connecting white vertices with black vertices. The edge $(d, c)$ is the unique light edge crossing the cut. A subset $A$ of the edges is shaded; note that the cut $(S, V-S)$ respects $A$, since no edge of $A$ crosses the cut. (b) The same graph with the vertices in the set $S$ on the left and the vertices in the set $V-S$ on the right. An edge crosses the cut if it connects a vertex on the left with a vertex on the right.
Proof

• Let $T$ be a MST that includes $A$, and assume $T$ does not contain the light edge $(u, v)$
• First, we construct another MST $T'$ that includes $(u, v)$
  – Adding $(u, v)$ to $T$ induces a cycle
  – Let $(x, y)$ be the edge on the cycle crossing $(S, V - S)$, then $w(u, v) \leq w(x, y)$, hence $w(u, v) - w(x, y) \leq 0$
  – $T' = T - (x, y) \cup (u, v)$
  – $T'$ is also a MST since
    $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$
• Second, we prove that $(u, v)$ is a safe edge for $A$
  – Since $A \subseteq T$ and $(x, y)$ is not in $A$ then $A \subseteq T'$. Therefore $A \cup \{(u, v)\} \subseteq T'$. Since $T'$ is a MST, $(u, v)$ is safe for $A$

Optimal substructure property

• Let $T$ be an MST of $G$ and $(u, v)$ be an edge in $T$
• Removing $(u, v)$ partitions $T$ into two trees $T_1$ and $T_2$
• Let $(S, V - S)$ be a cut that respects $T_1$ and $T_2$
• Let $E_1$ be the subset of edges incident to $S$, and $E_2$ be the subset of edges incident to $V - S$
• Claim: $T_1$ is an MST of $G_1 = (S, E_1)$, and $T_2$ is an MST of $G_2 = (V - S, E_2)$
  – Note that $w(T) = w(u, v) + w(T_1) + w(T_2)$
  – A spanning tree “cheaper” than $T_1$ or $T_2$ cannot exists for $G_1$ or $G_2$, otherwise $T$ would not be optimal
Generic MST algorithm

\begin{algorithm}
\caption{GENERIC-MST($G$, $w$)}
\begin{algorithmic}[1]
\State $A \leftarrow \emptyset$
\While{$A$ does not form a spanning tree}
\State do find an edge $(u, v)$ that is safe for $A$
\State $A \leftarrow A \cup \{(u, v)\}$
\EndWhile
\Return $A$
\end{algorithmic}
\end{algorithm}

The Algorithms of Kruskal and Prim

- **Kruskal’s algorithm**
  - $A$ is a forest
  - The safe edge added to $A$ is always a minimum-weight edge in the graph that connects two distinct trees in $A$

- **Prim’s algorithm**
  - $A$ is a single tree
  - The safe edge added to $A$ is always a minimum-weight edge connecting the tree to a vertex not in the tree
Prim’s Algorithm

- The edges in the set $A$ always form a single tree.
- The tree starts from an arbitrary vertex and grows until the tree spans all the vertices in $V$.
- At each step, a light edge is added to the tree $A$ that connects $A$ to an isolated vertex of $G_A=(V, A)$.
- “Greedy” because the tree is augmented at each step with an edge that contributes the minimum amount possible to the tree’s weight.

Prim vs. Dijkstra

- Prim’s strategy similar to Dijkstra’s.
- Grows the MST $T$ one edge at a time.
- “Cloud” covers $A$, that is, the portion of $T$ already computed.
- Label $D[u]$ associated with each vertex $u$ outside the cloud (distance to the cloud).
Prim’s algorithm

• For any vertex $u$, $D[u]$ represents the weight of the current best edge for joining $u$ to the rest of the tree in the cloud (as opposed to the total sum of edge weights on a path from start vertex to $u$)

• Use a priority queue $Q$ whose keys are $D$ labels, and whose elements are vertex-edge pairs

Prim’s algorithm

• Any vertex $v$ can be the starting vertex

• We still initialize $D[v] = 0$ and all the other $D[u]$ values to $+\infty$

• We can reuse code from Dijkstra’s, just change a few things
Prim’s algorithm

**Algorithm PrimJarník(G):**

*Input:* A weighted connected graph G with n vertices and m edges  
*Output:* A minimum spanning tree T for G

Pick any vertex v of G  
D[v] ← 0  
for each vertex u ≠ v do  
D[u] ← +∞  
Initialize T ← 0.  
Initialize a priority queue Q with an item ((u, null), D[u]) for each vertex u, where (u, null) is the element and D[u]) is the key.

while Q is not empty do  
(u, e) ← Q.removeMin()  
Add vertex u and edge e to T.  
for each vertex v adjacent to u such that z is in Q do  
[perform the relaxation procedure on edge (u, z)]  
if w((u, z)) < D[z] then  
D[z] ← w((u, z))  
Change to (z, (u, z)) the element of vertex z in Q.  
Change to D[z] the key of vertex z in Q.

return the tree T

Time complexity

- Initializing the queue takes O(n log n) [binary heap]
- Each iteration of the while, we spend O(log n) time to remove vertex u from Q and O(deg(u) log n) to perform the relaxation step
- Overall, O(n log n + Σ_v(deg(v) log n)) which is O((n+m) log n) [if using a binary heap]

FYI: using Fibonacci heaps, Prim runs in O(m+n log n)
Kruskal’s Algorithm

- Initialization: $A$ is a forest of trees, where each node is a tree (with no edges)
- Sort the edges in increasing weight
- While $A$ is not a spanning tree of $G$
  - Consider the next edges $(u,v)$ in increasing order
  - Add $(u,v)$ to $A$ if it connects two distinct trees

---

**Algorithm** Kruskal($G$):

**Input:** A simple connected weighted graph $G$ with $n$ vertices and $m$ edges

**Output:** A minimum spanning tree $T$ for $G$

**for** each vertex $v$ in $G$ **do**

- Define an elementary cluster $C(v) \leftarrow \{v\}$.
- Initialize a priority queue $Q$ to contain all edges in $G$, using the weights as keys.

**T** $\leftarrow \emptyset$ \quad \{$T$ will ultimately contain the edges of the MST\}

**while** $T$ has fewer than $n - 1$ edges **do**

- $(u,v) \leftarrow Q\text{.removeMin}()$
- Let $C(v)$ be the cluster containing $v$, and let $C(u)$ be the cluster containing $u$.
- **if** $C(v) \neq C(u)$ **then**
  - Add edge $(v,u)$ to $T$.
  - Merge $C(v)$ and $C(u)$ into one cluster, that is, union $C(v)$ and $C(u)$.

**return** tree $T$
Data Structure for Kruskal Algorithm

- The data structure maintains a forest of trees
- We need a data structure that maintains a partition, i.e., a collection of disjoint sets, with the following operations
  - $\text{find}(u)$: return the set storing $u$
  - $\text{union}(u,v)$: replace the sets storing $u$ and $v$ with their union
Union-Find Abstract Data Type

• Let $S = \{S_1, S_2, \ldots, S_k\}$ be a dynamic collection of disjoint sets

• Each set $S_i$ is identified by a representative member (some member of the set)

Union-Find Abstract Data Type

• Operations
  
  **Make-Set(x):** create a new set $S_x$, whose only member is $x$
  (assuming $x$ is not already in one of the sets)
  
  **Union(x, y):** replace two disjoint sets $S_x$ and $S_y$ represented by $x$ and $y$ by their union
  
  **Find-Set(x):** find and return the representative of the set $S_i$ that contains $x$

• We will analyze the running time in terms of $(n,m)$ where $n =$ # of Make-Set and $m =$ # Make-Set + #Union + #Find-Set $(m \geq n)$

• Note that each Union operation reduces the number of sets by one, so the number of Union is at most $n-1$
Disjoint sets: tree representation

- Each set is a tree, and the representative is the root
- Each element points to its parent in the tree
- The root points to itself

Example: disjoint sets tree representation

```
{c,h,e,b}  {f,g,d}  Union(e,g)
```

```plaintext
{c,h,e,b}   {f,g,d}   Union(e,g)
```

- In the diagram, the set `{c,h,e,b}` is represented by the tree with root `c`, and the set `{f,g,d}` is represented by the tree with root `f`.
- The `Union(e,g)` operation merges the trees, resulting in a single tree with root `f`.
- The final tree combines elements `c`, `d`, `e`, `g`, `h`, and `b` under root `f`.
Disjoint sets: tree representation

- **Make-Set**: takes $O(1)$
- **Find-Set**: takes $O(h)$ where $h$ is the height of the tree
- **Union**: is performed by finding the two roots, and choosing one of the roots, to point to the other. This takes $O(h)$

- The complexity depends on how the trees are maintained

Disjoint sets: tree representation

- Two heuristics allow us to achieve a running time with is “almost linear” in the total number of operations $m$ (that is, almost $O(1)$ amortized)
  1. Union by rank
  2. Path compression
Union by rank

- Goal: make trees as shallow as possible
- Track the estimated size of each sub-tree by storing the *rank* of each node (upper bound on the height of the subtree, or the log of the subtree size)
- **Union by rank**: the root with small rank is made to point to the root with larger rank
- When a *Union* is performed, the rank of the root might need to be updated
Path compression

- Goal: make trees as shallow as possible
- During a Find-Set operation, make each node on the find path point directly to the root
- Find-Set is a two-pass method: one pass to find the root, and a second pass to update each node in the path
- Path compression does not change any rank
Union-Find: pseudocode

<table>
<thead>
<tr>
<th>Make-Set(x)</th>
<th>Union(x,y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x.p \leftarrow x )</td>
<td>Link(Find-Set(x), Find-Set(y))</td>
</tr>
<tr>
<td>( x.rank \leftarrow 0 )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Link(x,y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ( x.rank &gt; y.rank ) then ( y.p \leftarrow x ) /* x is the root */</td>
</tr>
<tr>
<td>else ( x.p \leftarrow y ) /* y is the root */</td>
</tr>
<tr>
<td>if ( x.rank = y.rank ) then ( y.rank \leftarrow y.rank + 1 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Find-Set(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ( x \neq x.p ) then ( x.p \leftarrow \text{Find-Set}(x,p) )</td>
</tr>
<tr>
<td>return ( x.p )</td>
</tr>
</tbody>
</table>
Observations about ranks

• Ranks satisfy the following properties
  – Longest path on the subtree rooted at $x \leq \text{rank}[x]$
  – For each node $u$, rank[$u$] is initially 0 then it increases monotonically with more and more Union until $u$
    becomes a non-root (at that time its rank is fixed)
  – The difference between the rank[$u$] and the rank[p[$u$]]
    increases monotonically with time
  – Along each path from a node to a root, the ranks are
    strictly increasing, i.e., rank[$u$] < rank[p[$u$]] if $u$ non-root
• All properties above can be proven by induction

Union by rank and path compression

• When both heuristics are used, the worst-case time complexity is $O(m \alpha(n))$ where
  $\alpha(n)$ is the inverse of the Ackerman function
• Proof: too technical 😎
• The inverse Ackerman function grows so slowly that for all practical purposes
  $\alpha(n) \leq 4$ for very very large $n$
An alternative bound …

• We prove a slightly weaker bound
• Define the iterated logarithm as $\log^{(i)} n = n$ and $\log^{(i)} n = \log(\log^{(i-1)} n)$
• Define: $\log^* n = \min \{ i : \log^{(i)} n \leq 1 \}$ (log base 2)
• For example, $\log^* 2 = 1$, $\log^* 4 = 2$, $\log^* 16 = 3$, $\log^* 65536 = 4$, $\log^* (2^{65536}) = 5$
• Define the tetration (iterated exponentiation) as $2^{<1>} = 2$ and $2^{<i+1>} = 2^{2^{<i>}}$
• Fact: $\log^* n = i$ iff $2^{<i-1>} < n \leq 2^{<i>}$

Analysis

• First note that each Union requires two Find-Set

• We just need to find a bound on the time needed to perform $m$ Find-Set
Properties of rank (1)

• **Lemma**: For all root nodes \( x \) of rank \( k \), the size of the tree rooted at \( x \) is at least \( 2^k \).

**Proof**: by induction on the number of Union. Based on the fact that a root node with rank \( k \) is created by merging two trees with roots of rank \( k-1 \)

Properties of rank (2)

• **Lemma**: If there are \( n \) elements overall, at most \( n/2^k \) elements have rank in the range \((k, 2^k]\).

**Proof**: Prove first that there are at most \( n/2^k \) elements of rank \( k \). From the previous lemma the maximum number of nodes of rank \( k \) is reached when each node with rank \( k \) is the root of a tree that has exactly \( 2^k \) nodes. In this case, the number of nodes of rank \( k \) is \( n/2^k \). Then,

\[
\sum_{r=k+1}^{2^k} \frac{n}{2^r} < n \sum_{r=k+1}^{\infty} \frac{1}{2^r} = \frac{n}{2^k} \sum_{r=1}^{\infty} \frac{1}{2^r} = \frac{n}{2^k}
\]
Properties of rank (3)

- **Corollary**: Every node has rank at most \( \lfloor \log_2 n \rfloor \)

**Proof**: There at most \( n/2^r \) nodes of rank \( r \). If \( r > \log_2 n \) then \( n/2^r < 1 \). Since ranks are natural numbers, the corollary follows.

Thus, the height of all trees is bounded by \( \log n \)

---

**Analysis**

- Partition the nodes according to their final rank. Put rank \( r \) nodes in block number \( \log^* r \) (for \( r = 0, 1, \ldots, \lfloor \log n \rfloor \))
  - Group 0 contains nodes of rank \((-1,2^0] = \{0,1\}\)
  - Group 1 contains nodes of rank \((1,2^1] = \{2\}\)
  - Group 2 contains nodes of rank \((2,2^2] = \{3,4\}\)
  - Group 3 contains nodes of rank \((4,2^3] = \{5,6,7,\ldots,16\}\)
  - Group 4 contains nodes of rank \((16,2^{16}] = \{17,18,\ldots,65536\}\)
  - Group 5 contains nodes of rank \((65536,2^{65536}] = \{65537,\ldots,2^{65536}\}\)
  - …
  - Group \( i \) contains nodes of rank \((2^{i-1},2^i]\)
  - …
- There are no more than \( \log^* n \) groups because the highest numbered block is \( \log^* (\log n) = \log^* n - 1 \)
Amortized Analysis

• Assign to each node $u$ a fixed amount of dollars *(credit)*, each of which is worth $O(1)$ time

• **Rule:** A node $u$ receives its credit as soon as it ceases to be a root, at which point its rank is fixed. If its rank is in the range $(k, 2^k]$ the node receives $2^k$ dollars of credit.

Analysis

• **Lemma:** We distribute at most $n \log^* n$ dollars of credit overall

  **Proof:** We are giving $2^k$ dollars to nodes of rank $(k, 2^k]$, and there are at most $n/2^k$ nodes in that group, so we give a total of $n$ dollars for that group. Since there are at most $\log^* n$ groups, the conclusion follows.
Analysis

• We will show that each Find-Set costs \( \log^* n \) time plus the some additional time which is paid using the credit

• There are \( m \) Find-Set, overall time \( m \log^* n \)

• We distributed \( n \log^* n \) credit dollars

• Overall \( O((m+n) \log^* n) \)

• **Lemma:** Each Find-Set operation can be completed in \( O(\log^* n) \) time [plus additional cost using credit]

**Proof:** The cost of Find-Set is proportional to the number of pointers traversed until we get to the root.

When we move from \( u \) to \( p[u] \)

– (Block-charges) if (1) \( u \) and \( p[u] \) belong to different groups, or (2) \( u \) is the root, or (3) \( p[u] \) is the root, then we charge the Find-Set

– (Path-charges) otherwise (\( u \) and \( p[u] \) belong to the same group) we charge \( u \)’s credit

Since there are at most \( \log^* n \) groups, the conclusion follows.
Credit is sufficient for path-charges

- **Lemma**: If $u$’s final rank belongs to the range group $(k, 2^k]$, then $u$ cannot be path-charged more than $2^k$ times.

**Proof**: When **Find-Set** path-charges $u$, $u$ will be assigned a new parent during path-compression. Moreover, $u$’s new parent will have a higher rank than $u$’s old parent. Thus, once a node (other than the root or its child) is assigned block-charges, it will never again be assigned path-charges.
Proof (continued)

• Suppose $u$ is in a group that has final rank in the range $(k, 2^k]$
• How many times can $u$ be assigned a new parent (i.e., be path-charged) before $u$ is assigned to a parent whose rank is in a different block?
• Worst-case: if $u$ has the lowest rank in its block $(k+1)$ and its parent’s ranks successively are $k+2, k+3, \ldots, 2^k$
• Then $u$ cannot be path-charged more than $2^k$ times, because after that parent of $u$ will move to another group; whereupon $u$ never has to pay path-charges again.

Summary

• For a sequence of $m>n$ Make-Set, Union, and Find-Set operations, of which $n$ are Make-Set
• Union by rank + path compression yields $O(m \alpha(n))$ complexity
  [here we proved $O((m+n) \log^* n )]$
Kruskal’s running time

• $m = \#$ edges, $n = \#$ nodes
• Cost of initializing the priority queue (or sorting) is $O(m \log m)$ which is $O(m \log n)$
• $O(m)$ Find-set and Union and $O(n)$ Make-set, overall $O(m \alpha(n))$
• Overall running time is $O(m \log n)$
• Sorting dominates the complexity, but there are cases in which Union-Find’s complexity becomes critical

Reading assignment

• Chapter 17, “Greedy algorithms”
• Section 24.3, “Dijkstra’s algorithm”
• Section 23.2, “Kruskal and Prim”
• Chapter 21, “Data Structures for Disjoint Sets”