11 Bellman-Ford Algorothm for Single-Source Shortest Paths

A graph is **weighted** if there are (possibly multiple types of) weights associatd with the edges. For an edge $e(u,v)$ in a weighted graph, we shall use $w(e)$, $w(u,v)$, $c(e)$, or $c(u,v)$ to represent its **weight** or **cost**. In some textbooks, a weighted graph is also referred to as a network.

Definition 10 *Given a weighted graph $G(V,E)$, and a vertex $s \in V$, find the shortest path from s to all other vertices.*

Before we present the Bellman-Ford algroithm for solving the single-source shortest paths problem, it is important to realize several key properties of shortest paths.

1. Subpaths of shortest paths are shortest paths. Precisely, let $v_0, e_1, v_1, e_2, v_2, ..., v_k$ be a shortest path from vertex v_0 to v_k, then for any i, j such that $0 \leq i \leq j \leq k$, the path $v_i, e_{i+1}, v_{i+1}, ..., v_j$ is a subpath and is a shortest path from v_i to v_j.
2. A shortest path doesn't contain a positive weight cycle.
3. If the weighted graph contains a negative cycle, then the shortest paths may not be well-defined. Assuming the graph has no negative cycles, the shortest path from any vertex to itself is 0.
4. For two vertices v, w, if there is no path from v to w, then the we say the shortest path length from v to w is ∞.
5. If the graph doesn't have negative cycles, then there is a shortest path from the source s to any vertex v using no more than $n-1$ edges, where n is the number of vertices.

The Bellman-Ford algorithm is a dynamic programming algorithm based on the induction of the number of edges k on the shortest path. In each iteration, the algorithm calculates the shortest paths from the source s to all other vertices using $\leq k$ edges. If the graph doesn't have any negative cycles reachable from the source s, then then the algorithm finds all single-source shortests paths in $n-1$ iterations.

B.C.: $k = 0$, i.e., the shortest paths from s to all other verteices using ≤ 0 edges. Obviously, the shortest path distance from s to itself is 0, and that from s to any other vertex is ∞.

I.H.: Assume that that we have the shortest path distnaces from s to any vertex in V using $\leq k-1$ edges. Let's assume that these shortest path distances are denoted by $SP_{k-1}(s,v)$.

I.S.: Based on the I.H., can we obtain the shortest path distances from s to any vertex in V using $\leq k$ edges?

Consider a vertex v. The shortest path using $\leq k$ edges from s to v has two possibilities. Either the path uses $< k$ edges, or it uses exactl k edges. If the path uses $< k$ edges, then this path is provided by the I.H. Otherwise, assume this path uses exactly k edges, and let the path be $v_0 = s, v_1, ..., v_{k-1}, v_k = v$. According to the structure of shortest paths, the subpath $v_0 = s, v_1, ..., v_{k-1}$ must be the shortest path from s to v_{k-1} using $\leq k-1$ edges, and hence must be available from the I.H. Thus, to obtain the shortest path of $\leq k$ edges, all we need to do is add edge (v_{k-1}, v_k) to the shortest path of $\leq k-1$ edges from s to v_{k-1}.

The difficulty here is that we don't know v_{k-1}. However, we can try every vertex in V as v_{k-1} and take the one that produces the shortest paths, i.e.,

$$SP_k(s,v) = \min_{v \in V}\{SP_{k-1}(s,u) + c(u,v)\}$$

Algorithm: Bellman-Ford

Input: The adjacency matrix of a weighted graph $G(V,E)$, a source vertex s. For ease of explanation, let $V = \{1, 2, ..., n\}$ and $s \in V$.

Output: The shortest path distance from s to all vertices in V.

```
1. Initialize an array $SP[n][n]$, where each entry $SP[k][v]$ stores the shortest path from $s$ to $v$ using
   $\leq k$ vertices
2. $SP[0][s] = 0$
3. For $j = 1$ to $n$ and $j \neq s$, $SP[0][j] = \infty$
4. For $k = 1$ to $n-1$ do:
5.     For $j = 1$ to $n$ do:
6.         $SP[k][j] = \min_{i \in V}\{SP[k][i] + w(i,j)\}$
```

What if the given graph has negative cycles? The following algorithm will detect negative cycles.

Algorithm: Bellman-Ford

Input: The adjacency matrix of a weighted graph $G(V,E)$, a source vertex s. For ease of explanation, let $V = \{1, 2, ..., n\}$ and $s \in V$.

Output: Return Yes if the graph doesn't have negative cycles and the shortest path distance from s to all vertices in V. Return No otherwise.

```
1. Initialize an array $SP[n+1][n]$, where each entry $SP[k][v]$ stores the shortest path from $s$ to $v$
   using $\leq k$ vertices
2. $SP[0][s] = 0$
3. For $j = 1$ to $n$ and $j \neq s$, $SP[0][j] = \infty$
4. For $k = 1$ to $n-1$ do:
5.     For $j = 1$ to $n$ do:
6.         $SP[k][j] = \min_{i \in V}\{SP[k][i] + w(i,j)\}$
7. For $j = 1$ to $n$ do:
```

```
8.      $SP[n][j] = \min_{i \in V}\{SP[n-1][i] + w(i,j)\}$
9.      If $SP[n][j] < SP[n][j]$, return No.
10. return Yes.
```

Now, let's prove that the added code will detect negative cycles in the given graph. Suppose the graph G has a negative cycle in $v_0, v_1, v_2, ..., v_k = v_0$ that is reachable from the source s. Clearly, $\sum_{j=0}^{k-1} w(v_j, v+j+1) < 0$. Assume that $SP[n][v_j] \geq SP[n-1][v_j]$ for $j = 1, 2, ..., k$. Since $SP[n][v_j] = \min_{v \in V}\{SP_{n-1}(s,u) + c(u,v)$, we have $SP[n-1][v_j] + c(v_j, v_{j+1}) \geq SP[n][v_{j+1}] \geq SP[n-1][v_{j+1}]$. This implies:

$$SP[n-1][v_0] + c(v_0, v_1) \geq SP[n-1][v_1]$$
$$SP[n-1][v_1] + c(v_1, v_2) \geq SP[n-1][v_2]$$
$$SP[n-1][v_2] + c(v_2, v_3) \geq SP[n-1][v_3]$$
$$\cdots$$
$$SP[n-1][v_{k-1}] + c(v_{k-1}, v_0) \geq SP[n-1][v_0]$$

Summing up all these inequalities, we have

$$\sum_{j=0}^{k-1} SP[n-1][v_j] + \sum_{j=0}^{k-1} c(v_j, v_{j+1}) \geq \sum_{j=0}^{k-1} SP[n-1][v_j]$$

i.e.,

$$\sum_{j=0}^{k-1} c(v_j, v_{j+1}) \geq 0$$

A contradiction! Thus, if there is a negative cycle, for at least one j, we will have $SP[n][j] < SP[n-1][j]$, and the algroithm will return a "No".

12 Floyd-Warshall Algorothm for All-Pair Shortest Paths

Definition 11 *Given a weighted graph $G(V,E)$, find the shortest path between every pair of vertices.*

Suppose that the vertices of G are labled as $1,2,...,n$, where n is the number of vertices. The Floyd-Warshall algorithm is a dynamic programming algorithm based on the induction of vertex-label on the shortest path. More precisely, consider a path $v_0, v_1, ..., v_m$, where $v_0, v_1, ..., v_k \in \{1,2,...,n\}$ are labelled vertices. Then the path $v_0, v_1, ..., v_k$ is called a $k-$path if $\max\{v_1, v_2, ..., v_{m-1}\} \leq k$.

In Floyd-Warshall algroithm, in each iteration, the algorithm calculates the shortest $k-$paths between every pair of vertices.

B.C.: $k = 0$, i.e., the shortest $0-$paths between every pair of vertices Obviously, the shortest $0-$path from any vertex to itself is 0. For a pair of vertices u, v, if there is an edge (u,v) connecting them, then the shortest $0-$path from u to v is simply $w(u,v)$; otherwise, the shortest $0-$path doesn't exist, and should be ∞. Clearly, the shortest $0-$paths is the adjacency matrix.

I.H.: Assume that that we have the shortest $(k-1)-$paths between every pair of vertices. Let's assume that these shortest paths are stored in a matrix $D_{k-1}[i][j]$.

I.S.: Based on the I.H., can we obtain the shortest $k-$paths between every pair of vertices.

Consider a pair of vertices i and j. The shortest $k-$path has two possibilities. Either the vertex k is on the path or the vertex k is not on the path. If the vertex k is on the path, then since sub-paths of shortest paths are all shortest paths, the shortest $k-$path must be $D_{k-1}[i][k]+D_{k-1}[j][k]$. On the other hand, if the vertex k is not on the path, then the shorest $k-$path must be the same as the shortest $(k-1)-$path, i.e., $D_{k-1}[i][j]$. Thus, to obtain the shortest $k-$path, we will have to take the minimum of these two scenarios, i.e., $D_k[i][j] = \min\{D_{k-1}[i][k] + D_{k-1}[j][k], D_{k-1}[i][j]$.

Algorithm: Floyd-Warshall

Input: The adjacency matrix of a weighted graph $G(V,E)$.

Output: All pair shotest path distances.

```
1. Initialize an array $D[n+1][n][n]$, where each entry $D[k][i][j]$ stores the shortest $k-$path from $i$ to $j$.
2. Set $D[0]$ to be the adjacency matrix of $G$.
3. For $k = 1$ to $n$ do:
4.     For $i = 1$ to $n$ do:
5.         For $j = 1$ to $n$ do:
6.             $D[k][i][j] = \min\{D[k-1][i][k] + D[k-1][k][j], D[k-1][i][k]\}$.
```

What if the given graph has negative cycles?

13 Topological Sort

Definition 12 *A directed acyclic graph (or in short DAG) is a directed graph with no directed cycles.*

Problem 10 *Given a weighted DAG G with non-negative edge costs, and a source vertex s, find the shortest paths from s to all other vertices.*

The most efficient algorithm for calculating single-source shortest paths in a DAG involves topological order of the graph. We will therefore present topological order first.

13.1 Topological Order

Definition 13 *A* **toplogical order** *of a DAG G is a linear ordering of the vertices such that if G contains an arc (u, v), then u appears before v in the ordering. An algorithm that produces the topological order of a DAG is called a* **topological sorting** *algorithm.*

Question 1 *Is the topological order of a given DAG unique?*

Answer 1 *No.*

Observation 3 *B.C.: The vertices with in-degree* 0 *should be sorted first.*

I.H.: Suppose we know how to find the first k vertices of the topological order of a graph G.

I.S.: Can we find the $(k+1)-th$ vertex of the topological order? Yes, by removing all the edges associated with the first k vertices, any remaining vertex with in-degree 0 *can be the $(k+1)-th$ vertex.*

Question 2 *How to implement the above idea?*

Answer 2 *We can use a queue to store all the vertices with in-degree* 0*. As edges are being "removed" from the graph, the in-degree of some other vetices will reduce to* 0 *and will be entered into the queue. The algorithm iterates until the queue is empty.*

Algorithm: Topological Sort

Input: A DAG $G(V, E)$

Output: The topological order of V. We will assume that each vertex $v \in V$ has an attribute called "label" that stores its topological order index.

```
1. For all verteices set $v.label$ to 0.
2. Create an empty queue $Q$.
3. Find all vertices in $V$ with in-degree 0 and enter them into $Q$.
4. $i = 1$
5. while $Q$ is not empty do:
```

```
6.     $v = deQueue(Q)$
7.     $v.label = i$, $i++$
8.     For all arcs $(v, w)$ do:
9.         $w.indegree--$
10.        if $w.indegree = 0$, enter $w$ into $Q$.
```

The running time for toplogical sort is $O(|V| + |E|)$.

13.2 Difference between Topological Order and Breadth First Search

An algorithm that is very similar to the above topological sorting algorithm is the **breadth first search (BFS)** algorithm. Recall that the iterative version of the BFS algorithm explores the graph using a **queue** as well. BFS explores the vertices and edges of a given graph G from a "source" vertex s. It "discovers" the vertices of G based on their shortest paths in terms of the number of edges from s. The vertices closer to s are discovered first, while the vertices farther away are discovered later. If the graph is undirected, the algorithm iterates from one connrected component to another until the entire graph is explored. The running time for BFS is $O(|V| + |E|)$.

The BFS explores vertices based on their shortest distance from the source. In topological sort, on the other hand, the vertices are explored based on the longest distance from the vertices with indegree 0.

```
Algorithm: BFS
    Input: A directed graph $G(V, E)$
   Output: Varies and depends on the application.

1.  Mark all vertices "unvisited".
2.  While there are still "unvisited" vertices do:
3.      Pick an unvisited vertex $v$
4.      Mark $v$ as "visited"
5.      Create an empty queue $Q$.
6.      EnQuene $(Q, v)$.
7.      While $Q$ is not empty do:
8.          $u = deQueue(Q)$
9.          For all arcs $(u, w)$ do:
10.             if $w$ is "unvisited"
11.                 Mark $w$ as "visited"
```

```
12.            EnQueue $(Q, w)$.
```

Another graph exploration algorithm is the the **depth first search (DFS)**. Recall that DFS explores a graph using a **stack**. As the name implies, DFS tries to search "**deep**". For any vertex v, DFS **recursively** search all unexplored edges leaving v. It **backtracks** only after all edges leaving v have been explored.

The pseudo-code for DFS is as follows:

```
Algorithm: DFS
    Input: A directed graph $G$
   Output: Depends. DFS can discover paths, cycles, connected components, etc, and produce a DFS tree.

 1. Mark all vertices "unvisited".
 2. While there is still unvisited vertices, pick an unvisited vertex $v$ and do:
 3.     Mark $v$ as "visited".
 4.     DFS-Visit $(G, v)$.
```

```
Algorithm: DFS-Visit
    Input: A directed graph $G(V, E)$ and a vertex $v \in V$.
   Output: Variies.

 1. For each edge $(v, u) \in E$ do:
 2.     if $u$ is "unvisited"
 3.         Mark $u$ as "visited".
 4.         DFS-Visit $(G, u)$.
```

The iterative version of DFS is as follows:

```
Algorithm: DFS
    Input: A directed graph $G(V, E)$
   Output: Varies and depends on the application.

 1. Mark all vertices and edges "unvisited".
 2. Create an empty stack $S$.
 3. While there are still "unvisited" vertices do:
 4.     Pick an unvisited vertex $v$
 5.     $Push(S, v)$
```

```
6.  While $S$ is not empty do:
7.      $u = Pop(S)$
8.      While there is an "unvisited" edge $(u, v)$ do:
9.          Mark $(u, v)$ as "visited".
10.         If $v$ is "unvisited"
11.             $Push(S, u)$
12.             $Push(S, v)$
13.     Mark $u$ as "visited"
```

The running time for DFS is $O(|V| + |E|)$.

13.3 Single-Source Shortest Paths on DAG

The single-source shortest paths on DAG calculates the shortest paths based on toplogical order. For ease of explanation, let $G(V, E)$ be a DAG, and $s \in V$ be the given source vertex. Clearly, any vertices notreachable from s will have a shortest path distance of ∞. Thus, WLOG, we shall assume that all vertices in G are reachable from s. Since there are no cycles, $s.label = 1$.

B.C.: The shortest path distance from s to itself is 0.

I.H.: Assume we have found the shortest path distnaces for all vertices with topological ordering label $< k$.

I.S.: Can we find the shortest paths from s to the vertex v with $v.label = k$?

Consider the incoming arcs $\{(u_1, v), (u_2, v), ..., (u_l, v)\}$ of v. From the I.H., we already have the shortest paths from s to $u_1, u_2, ..., u_l$. Obviously, the shortest path from s to v has to go through one of the vertices from $\{u_1, u_2, ..., u_l\}$. Since we don't know which vertex, we can simply take the one that gives us the shortest path, i.e., $\min_{1 \le j \le l} SP(s, u_j) + c(u_j, v)$.

Algorithm: Single Shortest Path on DAG

Input: A DAG $G(V, E)$ and a source vertex $s \in V$.

Output: Singles source shortest paths from s to all other vertices.

```
1. Topopolical sort $G$, and assume the topological order for each vertex $v$ is stored in $v.label$.
2. Assume that each vertex has an attribute called $SP$ that stores its shorest path distnace from $s$.
3. For all vertex $v \in V$, set $v.SP = \infty$.
4. $s.SP = 0$.
5. For vertices $v$ with topological label from $s.label + 1$ to $|V|$ do:
6.     $v.SP = \min_{1 \le j \le l} u_j.SP + c(u_j, v)$
```

The running time of the above algorithm is $O(|V| + |E|)$.

14 Dijkstra's Algorithm

Problem 11 *Given a weighted undirected graph G with non-negative edge costs, and a source vertex s, find the shortest paths from s to all other vertices.*

If you understand BFS and shortest paths for DAG algorithms, then Dijkstra's algroithm is very easy to understand. It can be viewed as combining these two algorithms. Dijstra's algorith aims to calculate the shortest paths from the source vertex s to all other vertices based on their distances to the source. In other words, the shortest paths of vertices closer to the source will be calculated first; the vertices farther away will be calculated afterwards.

B.C.: Since the graph doesn't have non-negative edge cost, the vertex closest to the source vertex is the source itself, and the shortest path distance is 0.

Note that, the Base Case also reveals why Disjkstra's algroithm can **NOT** be used for graphs with negative edge costs. This is because if the graph has negative edge costs, then the vertex cloeset to the source may not be the source itself, making the basis of the induction invalid.

I.H.: The algroithm will run in iterations. In each iteration, the next closet vertex is discovered and its shortest path distance obtained. Thus, the very first iteration discovers the vertex cloeset to the source, which is the base case. Assume that by the end of iteration $k-$th iteration, we have found the k vertices cloeset to the source s.

I.S.: Can we find the $(k+1)-$th cloeset vertex?

Let the k closest vertices be denoted by $v_1 = s, v_2, ..., v_k$. Assume that the $(k+1)-$ closest vertex is v_{k+1}. Let $\{(u_1, v), (u_2, v), ..., (u_l, v)\}$ be the set of incoming arcs to v. Obviously, the shortest path from s to v has to go through one of the verteices from $\{u_1, u_2, ..., u_l\} \cap \{v_1 = s, v_2, ..., v_k\}$. The difficulty here is that we don't know which vertex is v_k and we don't know which vertex v_{k+1} is, and which vertex u_j the shortest path to v_{k+1} goes through. However, we don't know that the vertx v_{k+1} is adjacent to some vertex from $v_1 = s, v_2, ..., v_k\}$. This immeidately implies that v_{k+1} is the vertex that minimizes:

$$\min_{v \in V - \{v_1 = s, v_2, \ldots, v_k\}} \left(\min_{v_j \in \{v_1 = s, v_2, \ldots, v_k\}} v_j.SP + c(v_j, v) \right)$$
$$= \min_{v_j \in \{v_1 = s, v_2, \ldots, v_k} \left(\min_{(v_j, v), v \in v_j \in \{v_1 = s, v_2, \ldots, v_k} v_j.SP + c(v_j, v) \right)$$

Since in every iteration, we are searching for a minimum, a priority queue (heap) will be used in the Dijkstra's algorithm.

Algorithm: Diskstra's Shortest Path using a Heap

Input: A non-negative weighted graph $G(V, E)$ and a source vertex $s \in V$.

Output: The single source shortest path distances from s to all vertices in V.

1. Create a min-heap H.

```
2.  For all vertices $v \in V - \{s\}$, $v.SP = \infty$.
3.  $s.SP = 0$
4.  For all vertices $v \in V$ insert $(H, v)$ using $v.SP$ as the key.
5.  while $H$ is not empty do:
6.      $v = remove - root(H)$.
7.      For all edges $(v, w)$ do:
8.          if $v.SP + c(v, w) < w.SP$
9.              $w.SP = v.SP + c(v, w)$.
10.             Update $w$ in $H$ based on its new key $w.SP$.
```

The running time of the algorithm is $O\left((|V| + |E|) \log |V|\right)$.

15 Minimum Spanning Tree

Problem 12 *Given an undrected weighted graph $G(V, E)$ with no negative edge weights, find the spanning tree of G with the minimum total edge weights.*

Clearly, the minimum spanning tree of G has $|V|$ vertices and $|V| - 1$ edges. We will use two inductive approaches to solve for the problem. In the first approach, we shall add vertices to the spanning tree one at a time, while in the second approach, we shall add edges to the spanning tree one at a time.

15.1 Prim's Algorithm

The Prim's algorithm is also called the algorithm for "growing" a minimum spanning tree. The algorithm maintains a sub-tree of the final minimum spanning tree. In each iteration, the algorithm will "grow" this sub-tree by adding in one more vertex. When the maintained tree has $|V|$ vertices, the algroithm terminates with the final minimum spanning tree.

B.C.: Clearly, any singleton tree consisting any individual vertex is a sub-tree of the final minimum spanning tree.

I.H.: Suppose we have grown a sub-tree of k vertices of the final minimum spanning tree.

I.S.: Can we expand this sub-tree to $k + 1$ vertices?

Since we are maintaining a tree in each iteration, clearly, the $(k+1)-$vertex must be adjacent to one of the k vertices that are already in the in the current sub-tree. Since our goal is to minimize the total edge weights of the spannig tree, naturally, we will greedily expand along the shortest edge.

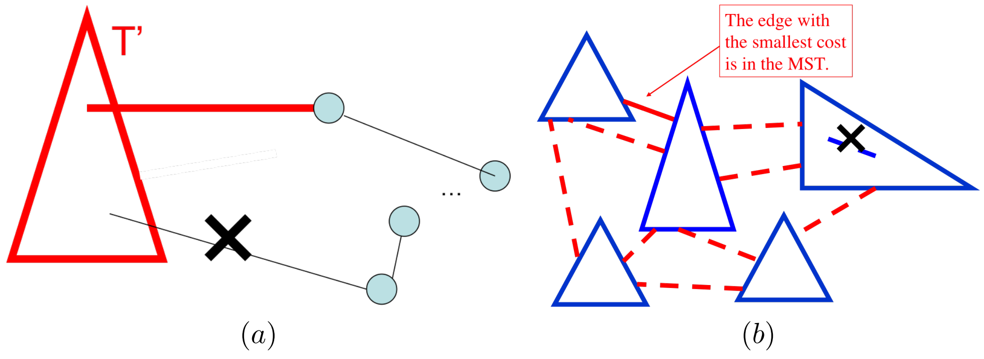

Figure 8: (a) Illustrating the correctness of Prim's Algorithm. (b) Illustrating the correctness of Kruskal's Algorithm.

Algorithm: Prim's Algorithm

Input: An undirected graph $G(V, E)$

Output: The MST of G. We assume each edge that is included in the MST is marked at the end.

```
1. Create an empty min-heap $H$.
2. Unmark all vertices and edges in $G$.
3. Pick a vertex $v$ and set $v$ as “marked”.
4. For edges $e$ adjacent to $v$, insert $e$ into $H$ using its weight as the key.
5. while $H$ is not empty do:
6.     $e = remove - root(H)$.
7.     Let $e = \{u, w\}$.
8.     If either $u$ or $w$ is “unmarked”
9.         Let $z$ be the unmarked vertex from $\{u, w\}$
10.        Mark $z$
11.        Mark $e$
12.        Insert all edges adjacent to $z$ that leads to an unmarked vertex into $H$.
```

15.2 Kruskal's Algorithm

The Kruskal's algorithm maintains a minimum spanning forest of the given graph. At the very beginning, this minimum spanning forest consists of $|V|$ trees, i.e., no edges. In each iteration, the algorithm will reduce the number of trees by adding in an edge. The algroithm terminates when there is only one tree left in the minimum spanning forest.

B.C.: Clearly, the forest of all vertices with no edges is a minimum spanning forest of the given graph.

I.H.: Suppose we have found a minimum spanning forest of k trees.

I.S.: Can we find a minimum spanning forest of $k - 1$ trees?

Since we need to reduce the number of trees, clearly, we need to add an edge between two trees in the forest. Since our goal is to minimize the total edge weights of the spannig forest, naturally, we will greedily add the shortest inter-tree edge.

Algorithm: Kruskal's Algorithm

Input: An undirected graph $G(V, E)$

Output: The MST of G. We assume each edge that is included in the MST is marked at the end.

```
1. Create an empty min-heap $H$.
2. Unmark all edges in $G$.
3. Insert all edges into $H$ based on their weight.
```

```
4. while $H$ is not empty do:
5.     $e = remove - root(H)$.
6.     Let $e = \{u, w\}$.
7.     If $u$ and $w$ **belong to different trees** in the forest
8.         Mark $e$
```

15.3 Union Find Data Structure

Problem 13 *Given a set of elements $S = \{x_1, x_2, ..., x_n\}$, how to support the following three operations?*

1. *MakeSet(x): if x is not in any set, create a singleton set $\{x\}$.*
2. *FindSet(x): return the identifier of the set containing x.*
3. *Union(x, y): merge the sets containing x and y.*

Recall that the Union-Find Data Structure uses the following key ideas:

1. Each set is represented as a tree. The name of the set is the root element of the tree.
2. MakeSet(x) will create a singleton tree with x being the root.
3. FindSet(x) will traverse from x to the root and return the root element.
4. Union(x, y) will combine two trees by connecting the root of one tree to the root of the other tree.
5. Union by Rank: we will maintain an attribute for node to represent the height of its subtree. When union two trees, we will always connect the root of the shorter tree to the root of the taller tree. Recall, with union-by-rank, the running time of union and find is bounded by $O(\log n)$.
6. Path Compression: when performing find operations, we need to traverse from the given element to the root. We will directly "hang" all the traversed element to the root. Recall with path compression, the amortized running time for m union find operations is bounded by $O((m + n) \log *n)$, where $\log *n$ is number of times the logarithm function is applied before the result is ≤ 1.