Weighted graphs

- A weighted graph is a graph that has a number $w(e)$ associated with each edge.
- Weights can be integers, rationals, reals, although in specific problems there may be restrictions.
- Weights can be positive, negative or 0, although in specific problems there may be restrictions.
- Weights can represent distance, cost, affinity, etc.

We will focus on two problems:

- Shortest path problem: Find path of minimum total weight, subject to specifications of the path
- Minimum spanning tree problem: Find spanning tree of minimum total weight

Shortest Paths

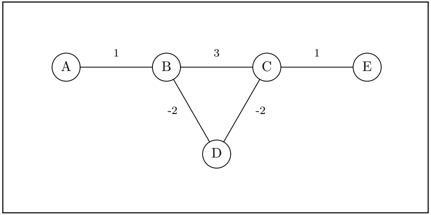
- Let G be a weighted graph. The length (or weight) of a path $P = e_0, \ldots, e_{k-1}$ is the sum of the weights of the edges:

$$w(P) = \sum_{i=0}^{k-1} w(e_i)$$

- Let u, v be two vertices in V. A shortest path (or minimum-length path or minimum-weight path) from u to v is a path from u to v of minimum total weight.
- The distance from vertex u to vertex v, denoted $d(u, v)$, is the length of the shortest path from u to v if such a path exists.
- Note on Terminology
  - The weight of an edge is sometimes called its cost or its length
  - Similarly, the weight (or length) of a path is sometimes called its cost

Shortest paths: Notes on definitions

- If no path exists from u to v, we will say that $d(u, v) = +\infty$.
- Even if there is a path from u to v, there may not be a shortest path. (Because of negative-weight cycles, sometimes called negative cycles.)



Single Source Shortest Path Problem

- Problem: Given graph G and a vertex $v \in V(G)$, find the shortest path from v to every other vertex in $V(G)$.
- We will discuss two algorithms:
  1. Dijkstra's algorithm
  2. The Bellman-Ford Algorithm

Dijkstra's Algorithm

- G an undirected graph in which every edge weight is nonnegative
- "Weighted BFS" starting at v
- Grow a "cloud" (actually a tree) of points:
  - Initially, the cloud is empty
  - At each iteration:
    1. Choose the vertex outside the cloud that is closest to v.
    2. Add the chosen vertex to the cloud
- Example of a greedy algorithm

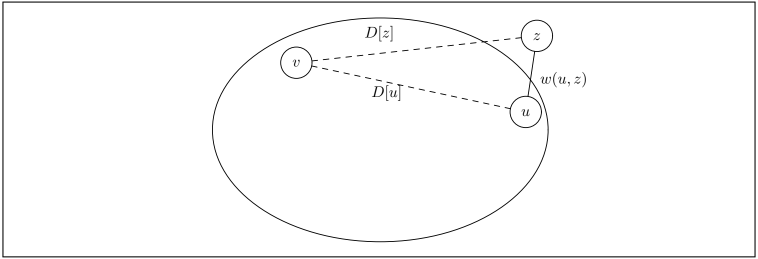
Overview of Dijkstra's algorithm

- Every vertex u has a label, $D[u]$.
- $D[u]$ stores the length of the best path from v to u that we have found so far
- Initially:
  - $D[v] = 0$ for our start vertex v
  - $D[u] = +\infty$ for all other vertices $v \neq u$
  - The vertex cloud C is empty (i.e., $C = \emptyset$)
- On each iteration:
  - Select a vertex u not in C with smallest $D[u]$ label
  - Put selected vertex u into C. (On first iteration, $u = v$).
  - Update $D[z]$ for each neighbor of u that is outside C (Because there may be a better path from v to z, via u, than we knew about before.)

```
if D[u] + w(u,z) < D[z] then
   D[z] ← D[u] + w(u,z)
```

This is called edge relaxation

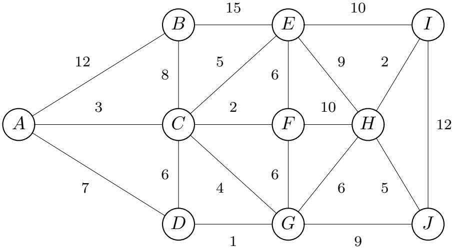
Edge relaxation, shortest-path tree

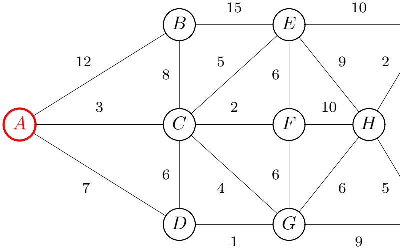



```
if D[u] + w(u,z) < D[z] then
   D[z] ← D[u] + w(u,z)
   z.parent = u;
```

- Setting `z.parent` organizes the points as a tree (the shortest-path tree), which allows us to find the shortest path after the algorithm completes.

Example of Dijkstra's Algorithm (Start vertex = A)





Pseudocode for Dijkstra's Algorithm

```
Algorithm DijkstraShortestPath(G,v)
  for each vertex u in G such that u ≠ v do
    D[u] = +∞
  D[v] = 0
  for each vertex u in G
    u.parent = null

  Put all vertices of G in a priority queue Q, using
      the labels D[] as keys
  while Q is not empty
    u ← Q.removeMin() // Priority Queue Operation
    for each z adjacent to u such that z is in Q do
       if D[u] + w(u,z) < D[z] then
          D[z] ← D[u] + w(u,z)
          Q.updateValue(z) to reflect new D[z] value
              // Priority Queue Operation
          z.parent = u

  return the collection of labels D[] and the parent values
```

Correctness of Dijkstra's algorithm

Follows from

- Lemma: When a vertex u is put in the "cloud" of known vertices (i.e., when u is removed from the priority queue), $D[u]$ represents the true distance from v to u

Correctness depends on there not being any negative edge weights.

Analysis of Dijkstra's Algorithm

Depends on implementation of priority queue

- Priority queue is abstract data type that supports removing the minimum, changing a value.
- Two possible implementations of a priority queue:
  1. Heap
     - Find/Remove minimum: $O(\log n)$
     - Update value: $O(\log n)$
  2. Array
     - Find/Remove minimum: $O(n)$
     - Update value: $O(1)$

Analysis of Dijkstra's Algorithm (continued)

- Operation counts:
  - Q.removeMin() performed n times
  - Q.updateValue() performed $O(m)$ times
- Standard implementation using heap:
  - Running time is $O((m+n)\log n)$
  - This simplifies to $O(m \log n)$ if G is connected
  - In terms of n only, this is $O(n^2 \log n)$ if G is simple
- Alternate implementation using an array:
  - Running time is $O((m+n^2)$
  - In terms of n only, this is $O(n^2)$ if G is simple
- Generally, heap implementation is better. However, if G is dense (in particular, if $m = \Omega(n/\log n)$), then the alternative implementation may be slightly better.
- One-sentence summary: Dijkstra's algorithm, using a heap, running on a simple connected graph, runs in $O(m \log n)$ time.

Finding the shortest path

After we have run Dijkstra's algorithm and computed the shortest-path tree, how do we find the actual shortest path from v to u?

- We can read off the path in reverse order with the following loop:

```
while (u.parent ≠ null)
   outputEdge(u,u.parent)
   u = u.parent
```

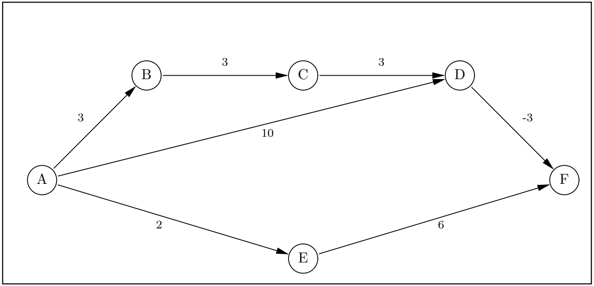
- We can read off the path in forward order by recursively calling `outputPath(u)`:

```
outputPath(u)
   if u.parent ≠ null then
      outputPath(u.parent)
      outputEdge(u.parent,u)
```

Bellman-Ford Shortest-path Algorithm

- Directed graphs
- Allows negative edges (unlike Dijkstra)
- Does not allow negative cycles
- Based on "iterative relaxation" of edge weights
- Repeatedly cycle through edges

Bellman-Ford example



Pass 1:Pass 2:Pass 3:Pass 4:Pass 5:Pass 5: Done

		u	D[u]
$\Rightarrow$	DF	A	0
$\Rightarrow$	CD	B	$+\infty$33
$\Rightarrow$	AE	C	$+\infty$66
$\Rightarrow$	BC	D	$+\infty$101099
$\Rightarrow$	AD	E	$+\infty$22
$\Rightarrow$	AB	F	$+\infty$87766
$\Rightarrow$	EF		

Pseudocode for Bellman Ford Algorithm

```
Algorithm BellmanFordShortestPath(G,v)
  for each vertex u in G such that u ≠ v do
    D[u] = +∞
  D[v] = 0
  for i ← 1 to n-1 do
    for each edge e=(u,z) in G do
       if D[u] + w(u,z) < D[z] then
          D[z] ← D[u] + w(u,z)
  if D[z] <= D[u] + w(u,z) for all edges e = (u,z) then
    return the collection of labels D[]
  else
    return "G contains a negative-weight cycle"
```

Correctness of Bellman-Ford algorithm

- For any i: After i iterations, $D[u]$ is $\leq$ the length of the shortest path from v to u that has at most i edges.
- So after $n-1$ iterations, $D[u]$ is $\leq$ the length of the shortest path from v to u that has at most $n-1$ edges.
- Since no simple path can have more than $n-1$ edges, the Bellman-Ford algorithm finds the shortest path provided there are no negative-weight cycles.

Analysis of Bellman-Ford algorithm

- $n - 1$ (i.e. $O(n)$) iterations of outer loop
- Each loop requires $O(m)$ relaxation tests
- Hence, total running time is $O(m \cdot n)$.

An optimization:

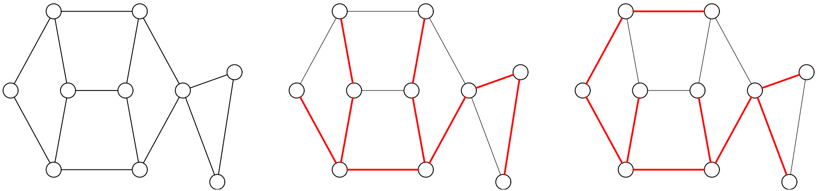
- We can stop after an iteration where no $D[u]$ values change. (So we don't necessarily need to run $n - 1$ iterations.

The Minimum Spanning Tree Problem

A spanning tree in a graph G is a free tree T such that

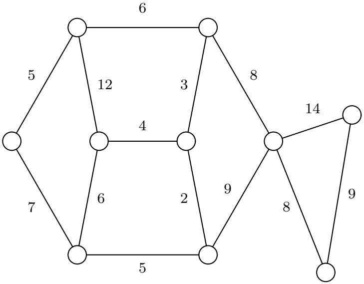
- T is a subgraph of G
- Every vertex of G is also a vertex of T.

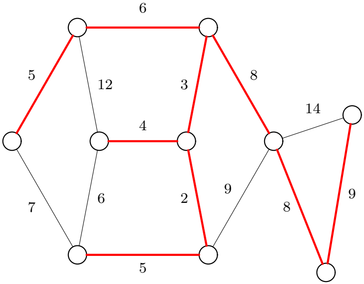
A graph may have many different spanning trees



Minimum Spanning Tree

A minimum spanning tree (MST) in a weighted undirected graph is a spanning tree of minimum total edge weight.

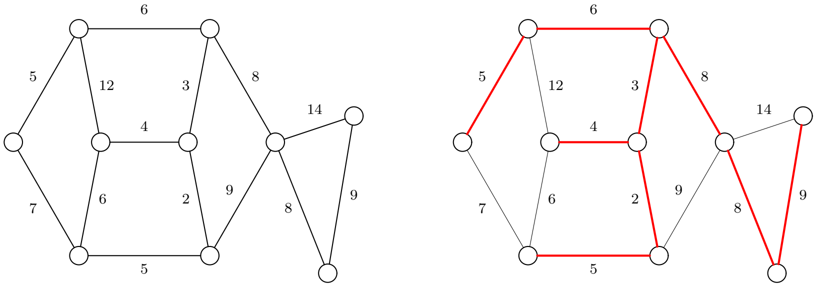




Minimum Spanning Tree

If we assume that the weights represent costs:

A minimum spanning tree represents the cheapest way to connect all nodes of a graph, using edges of the graph.



Applications:

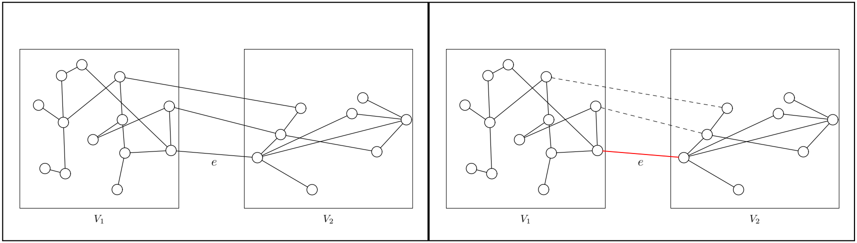
- Network layout
- Heuristics, subroutines for more complicated graph problems
  - Example: 2-Approximation for Traveling Salesman Problem with triangle inequality.

Uniqueness of Minimum Spanning Tree

- If all edge weights are distinct, the MST is unique.
- If two edge weights are the same, the MST may be unique, but it may not be.

A Crucial Fact about MST's

Let G be a weighted connected graph, and let V_1 and V_2 be a partition of $V(G)$ into two disjoint nonempty sets. Let e be an edge of minimum weight from among those edges with one endpoint in V_1 and the other endpoint in V_2. Then there is a minimum spanning tree that has e as one of its edges.



Minimum Spanning Tree algorithms

We will discuss three algorithms for computing the MST:

- Prim-Jarník algorithm: grow MST as a tree from a "root vertex"
- Kruskal's algorithm: Process edges in order of length
- Barůvka's algorithm: Each connected component selects the smallest edge connecting it with another connected component

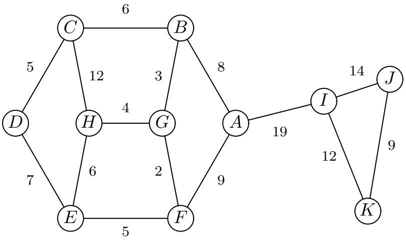
Prim-Jarník algorithm

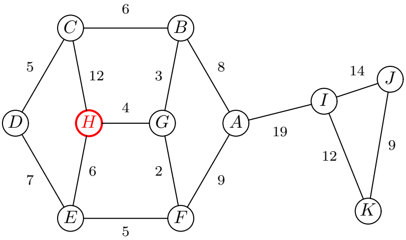
- Grow a MST from a single cluster, starting from some "root vertex"
- Similar to Dijkstra's algorithm (in fact, sometimes called the Prim-Dijkstra algorithm)
- Cloud of vertices C
- At each step:
  - Find smallest edge connecting a cloud vertex with a vertex not in the cloud.
  - Add this edge to the tree
  - Add endpoint outside cloud to the cloud.

Implementing the Prim-Jarník algorithm

- Each vertex u outside the cloud has a label $D[u]$ and another field, `u.parent`
- $D[u]$ stores the weight of the best edge connecting u to a cloud vertex.
- `u.parent` stores the other endpoint of this edge.
- When a vertex u is added to the cloud, the edge `(u,u.parent)` is added to the tree.

Prim-Jarník Example (Root vertex = H)





Done!

Pseudocode for Prim-Jarník algorithm

```
Algorithm PrimJarnikMST(G)
  pick an arbitrary root vertex v, set D[v] = 0
  for each vertex u in G such that u != v set D[u] = +∞
  for each vertex u in G set u.parent = null
  Initialize T ← ∅
  Put all vertices of G in a priority queue Q, using
      the labels D[] as keys
  while Q is not empty
    u ← Q.removeMin() // Priority Queue Operation
    add vertex u to T
    if u.parent != null add edge (u,u.parent) to T
    for each z adjacent to u such that z is not in T do
       if w(u,z) < D[z] then
          D[z] ← w(u,z)
          z.parent = u
          Q.updateValue(z) to reflect new D[z] value
              // Priority Queue Operation

  return the tree T
```

Analysis and Correctness of Prim-Jarník algorithm

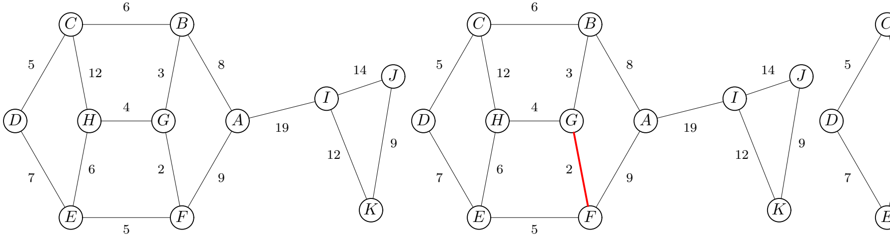
- Correctness: Follows from Crucial Fact stated earlier
  - V_1 = vertices in the cloud
  - V_2 = vertices not in the cloud (i.e., in the priority queue Q)
- Analysis:
  - Runs in $O(m \log n)$ time, assuming we use a heap for the priority queue and the graph is connected

Kruskal's MST algorithm

Basic idea:

- Build MST in clusters
- Initially, each vertex is in its own cluster
- Process edges of graph in nondecreasing order of weight (from smallest to largest)
- Add an edge if its endpoints are in two different clusters
- Continue until $n - 1$ edges have been added

Example



	Edge	Wt.	Tree?
1	*FG*	2	YY
2	*BG*	3	YY
3	*GH*	4	YY
4	*CD*	5	YY
5	*EF*	5	YY
6	*EH*	6	NN
7	*BC*	6	YY
8	*DE*	7	NN

	Edge	Wt.	Tree?
9	*AB*	8	YY
10	*AF*	9	NN
11	*JK*	9	YY
12	*CH*	12	NN
13	*IK*	12	YY
14	*IJ*	14	NN
15	*AI*	19	YY

Done!

Implementation of Kruskal's algorithm

- Store edges in a priority queue, using the edge weight as the key
- We have to manage the clusters. We need to implement:
  - Query: Are vertex u and vertex v currently in the same cluster?
  - Operation: Merge two clusters

Pseudocode for Kruskal's algorithm

```
Algorithm KruskalMST(G)

   for each vertex v in G
   Define an elementary cluster C(v), containing {v} //(*)
   Initialize a Priority Queue Q, storing the edges,
     using the weights as keys
   T ← 0
   while T has fewer than n − 1 edges do
      (u,v) = Q.removeMin();
      Let C(u) = the cluster containing u //(*)
      Let C(v) = the cluster containing v //(*)
      If C(u) != C(v) then //(*)
         Add edge (u,v) to T
         Merge C(u) and C(v) into a single cluster //(*)
```

- How do we implement "cluster management" functions at lines marked with //(∗) ?

Cluster Management

- Maintain each cluster as a linked list of vertices.
- Each vertex has an additional field `v.cluster`, indicating which cluster it belongs to
- When we merge two clusters, move elements of the smaller cluster into the larger cluster.

Analysis of Kruskal's Algorithm

- Priority queue operations: $O(m \log m)$, assuming heap implementation
  - Construct heap: $O(m)$
  - `removeMin()`: $O(m \log m)$ for all operations
    - $\leq m$ operations performed
    - Cost of each operation is $O(\log m)$ operation (which is actually $O(\log n)$ because G is simple),
- Cluster operations
  - Initializing clusters: $O(n)$
  - Merging clusters: $O(n \log n)$
    - Because each vertex gets moved between clusters at most $\log n$ times

So Kruskal's Algorithm runs in $O((m + n) \log n)$ time, which simplifies to $O(m \log n)$ time if G is connected.

Notes on Performance of Kruskal's Algorithm

- Limiting step is $O(m \log n)$ for heap operations. Cost of cluster operations match that step.
- If edges are already sorted, we can do better by using a faster but more complicated algorithm for cluster management, the Union-Find algorithm.
- We will come back to this later in these notes.

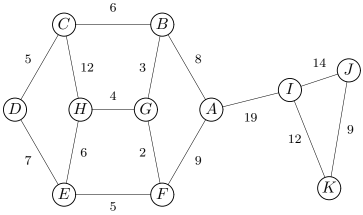
Correctness of Kruskal's Algorithm

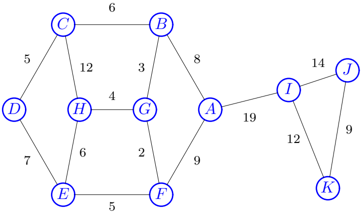
- Follows from "Crucial Fact"
  - V_1 = cluster containing u
  - V_2 = all other vertices

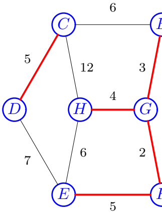
Barůvka's MST algorithm

- Proceeds in "rounds."
- Initially, each vertex is in its own cluster
- In each round:
  - Each cluster C selects the smallest edge in with one endpoint in C, the other endpoint outside C
  - All selected edges are added to MST.
  - Clusters are merged
- Continue until only one cluster remains

Barůvka's algorithm example







Round 1:

Cluster	Selected Edge
$\{A\}$	AB AB
$\{B\}$	BG BG
$\{C\}$	CD CD
$\{D\}$	CD CD
$\{E\}$	EF EF
$\{F\}$	FG FG
$\{G\}$	FG FG
$\{H\}$	GH GH
$\{I\}$	IK IK
$\{J\}$	KJ KJ
$\{K\}$	KJ KJ

Round 2:

Cluster	Selected Edge
$\{A, B, E, F, G, H\}$	BC BC
$\{C, D\}$	BC BC
$\{I, J, K\}$	AI AI

Sele

DONE!

Pseudocode for Barůvka's algorithm

```
Algorithm BarůvkaMST

   Let T be a subgraph initially containing
     just the vertices of G (and no edges)
   while T has fewer than n-1 edges do
      for each connected component C of T
         find the smallest weight edge e=(u,v)
           with u in C and v not in C
         add e to T (unless e is already in T)
   return T
```

Implementation of Barůvka's algorithm

- Store T as an adjacency list.
- In each round:
  1. Label connected components (each vertex gets a label indicating to which component it belongs)
  2. Scan edges to find edge that will be selected by each component.

  This takes $O(m)$ time per round

Analysis of Barůvka's algorithm

- Performance:
  - Since the number of components decreases by a factor of at least 2 each round, the number of rounds in $\leq \log n$,
  - The computation in each round is performed in $O(m)$ time
  - So running time of Barůvka's algorithm is $O(m \log n)$.
- Correctness:
  - follows from "Crucial Fact", with $V_1 = C$, $V_2 =$ everything else.
  - Technical issue: avoiding cycles. Number edges, use edge numbers to break ties.

The Union-Find Algorithm

Kruskal's algorithm

- Builds MST in clusters
- As we process each edge, we need to
  - Determine whether the two endpoints are in the same cluster
  - If not, add the edge to the MST and merge the two clusters
- Continue until only one cluster remains

We describe a very efficient algorithm for managing these clusters.

Set merging problem

The cluster management problem that arises in Kruskal's algorithm is a special case of the dynamic equivalence problem:

- Objects
- Sets of objects, with membership changing over time
- Queries: are two objects in the same set?

The special case that we need for cluster management in Kruskal's algorithm is called the set merging problem. The sets change as follows:

- Initially, each object is in its own set
- Over time, sets get merged. Once two sets get merged, they never get separated.

The set merging problem arises in other contexts. For example

- We are given some of the equivalence pairs in an equivalence relation, and we want to determine the equivalence classes.
- We are given the edges of a graph in some arbitrary order, and we want to construct the connected components without building the entire graph.

Example:

Suppose we have three sets:

set 1: $\{A, B, E\}$
set 2: $\{D, G\}$
set 3: $\{C\}$

A and C are not in the same set. But if we merged set 1 and set 3 and asked the question again, they would be in the same set.

The Union-Find algorithm

To solve the set merging problem, we will implement two primitive operations, called `union` and `find`.

1. `find(x)`: returns a temporary name for the set to which the object x currently belongs.

   More precisely, `find(x)` returns a special element in the set to which `x` belongs. If `x` and `y` are in the same set, then `find(x)` and `find(y)` will return the same value.
2. `union(s,t)`: If `s` and `t` are the temporary names of two different sets, this command causes the two sets to be merged.

The Union-Find algorithm

Once we have the operations `union` and `find`:

- To determine whether objects x and y are in the same set:

```
if find(x) == find(y) ...
```

- To determine whether x and y are in the same set and, if not, to merge them:

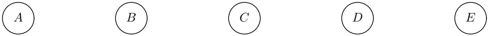
```
s = find(x);
t = find(y);
if s ≠ t
  union(s,t);
```

Data representation

- Each set is a tree of objects (not necessarily a binary tree). Each object has a parent pointer, but no child pointer.
- A `find(x)` command returns the object at the root of the tree to which `x` currently belongs.
- A `union(x,y)` command is only valid if `x` and `y` are both roots of trees. It combines the two trees by either
  - making `x` the parent of `y`, or
  - making `y` the parent of `x`

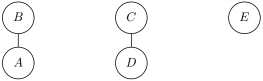
  We will discuss shortly how to make the choice.

Example

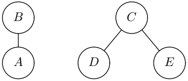


union(A,B)

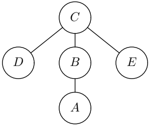
union(C,D)



union(C,E)



union(B,C)



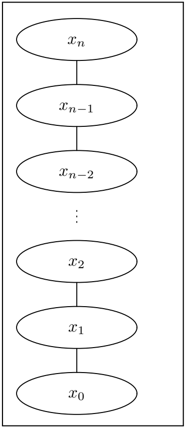
Union-Find

The algorithm as just stated is not very efficient.

For example, suppose there are $n+1$ objects $x_0, x_1, \ldots, x_n$. Suppose we first issue the following `union()` commands

union(x_0,x_1)
union(x_1,x_2)
...,
union(x_{n-1},x_n)

and then call find(x_0) repeatedly.



One possible result of the `union()` operations is as shown. Each call to `find()` would then require $\Theta(n)$ work.

Two simple improvements to Union-Find

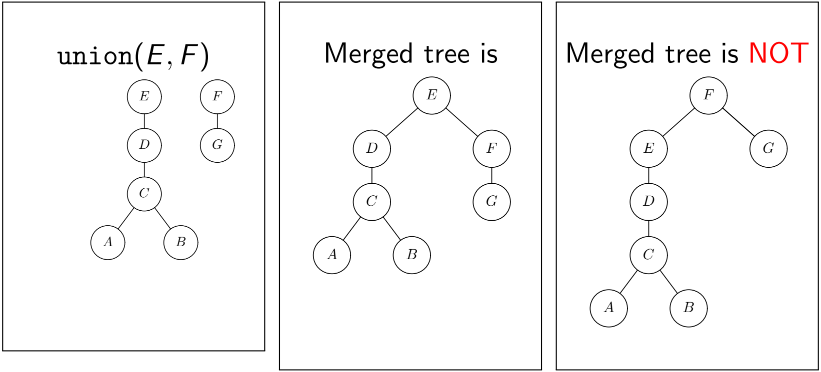
We can make the Union-Find algorithm extremely efficient with two simple improvements:

1. Weight balancing on `union`.
2. Path compression on `find`.

Improvement 1: Weight balancing on `union`

When two trees are merged, the root of the larger tree (the tree with more nodes) becomes the root of the new merged tree.

Example:



Implementation note

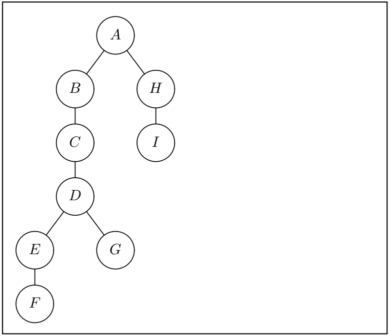
Implementation of weight-balancing:

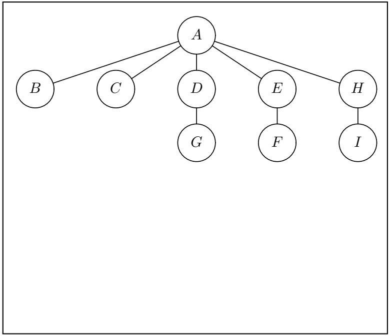
- Each node has a `weight` field.
- For a root note, the `weight` field contains the number of nodes in the tree rooted at that node.
  - Initially, each `weight` field is 1.
  - When two trees are merged, the `weight` field of the non-surviving root node is added to the `weight` field of the surviving root node.

Improvement 2: Path compression on `find`

On a `find(x)` operation, make every non-root node encountered on the path from `x` to the root a child of the root.

Example: `find(E)`:





With these two simple improvements, the union-find algorithm runs in "almost constant time per operation"

Analysis of the Union-Find Algorithm

- To describe the running time of the Union-Find algorithm, we first need to introduce a new function, called $\alpha(n)$ ("alpha"), that approaches ∞ as n gets large but does so extremely slowly.
- The function $\alpha(n)$ is the inverse of another function, $A(n)$, which approaches ∞ very quickly. $A(n)$ is defined as follows:

$$\begin{cases} A(0) & = & 1 \\ A(n) & = & 2^{A(n-1)} \qquad \text{if } n > 0 \end{cases}$$

$$\begin{aligned} A(0) &= 1 \\ A(1) &= 2 \\ A(2) &= 4 \\ A(3) &= 16 \end{aligned}$$

$$\begin{aligned} A(4) &= 65536 \\ A(5) &= 2^{65536} \approx 2 \times 10^{19728} \\ A(6) &= 2^{2^{65536}} \end{aligned}$$

The function $\alpha(n)$

$\alpha(n)$ = the smallest i such that $A(i) \geq n$

$$
\begin{aligned}
\alpha(1) &= 0 \\
\alpha(2) &= 1 \\
\alpha(n) &= 2 \qquad (2 < n \leq 4) \\
\alpha(n) &= 3 \qquad (4 < n \leq 16) \\
\alpha(n) &= 4 \qquad (16 < n \leq 65536) \\
\alpha(n) &= 5 \qquad (65536 < n \leq 2^{65536}) \\
\alpha(n) &= 6 \qquad (2^{65536} < n \leq 2^{2^{65536}})
\end{aligned}
$$

$\alpha(n)$ grows very slowly, but

$$\lim_{n \to \infty} \alpha(n) = \infty$$

Performance of Union-Find

It can be shown that

1. Any sequence of k union and find operations on a set of n objects, each initially in its own set, has a total cost of

$$O\left(k\alpha(n)\right)$$

2. There are sequences of k union and find operations on a set of n objects, each initially in its own set, that do in fact require

$$\Omega\left(k\alpha(n)\right)$$

So union-find has a cost per operation that is asymptotically worse than $\Theta(1)$, but is "almost" constant.

Note: This material can be found in [GT 4.2] and [CLRS 21]. [GT] proves a weaker analysis result. A proof of the analysis result stated here can be found in [CLRS].