Minimum Spanning Tree

References:

1. Algorithms, Jeff Erickson, Chapter 7
2. Algorithm Design Manual, Skiener, Chapter 6

Intro

Suppose we have a **connected**, **undirected**, ***weighted*** graph:

$$G = (V, E), w(e) \in R$$

for every edge $e \in E$ it has a weight $w(e)$. The weight could be positive, negative, or zero.

This lecture describes several algorithms to find the **minimum spanning tree** of G, that is the spanning tree T that minimizes the function

$$w(T) = \sum_{e \in T} w(e)$$

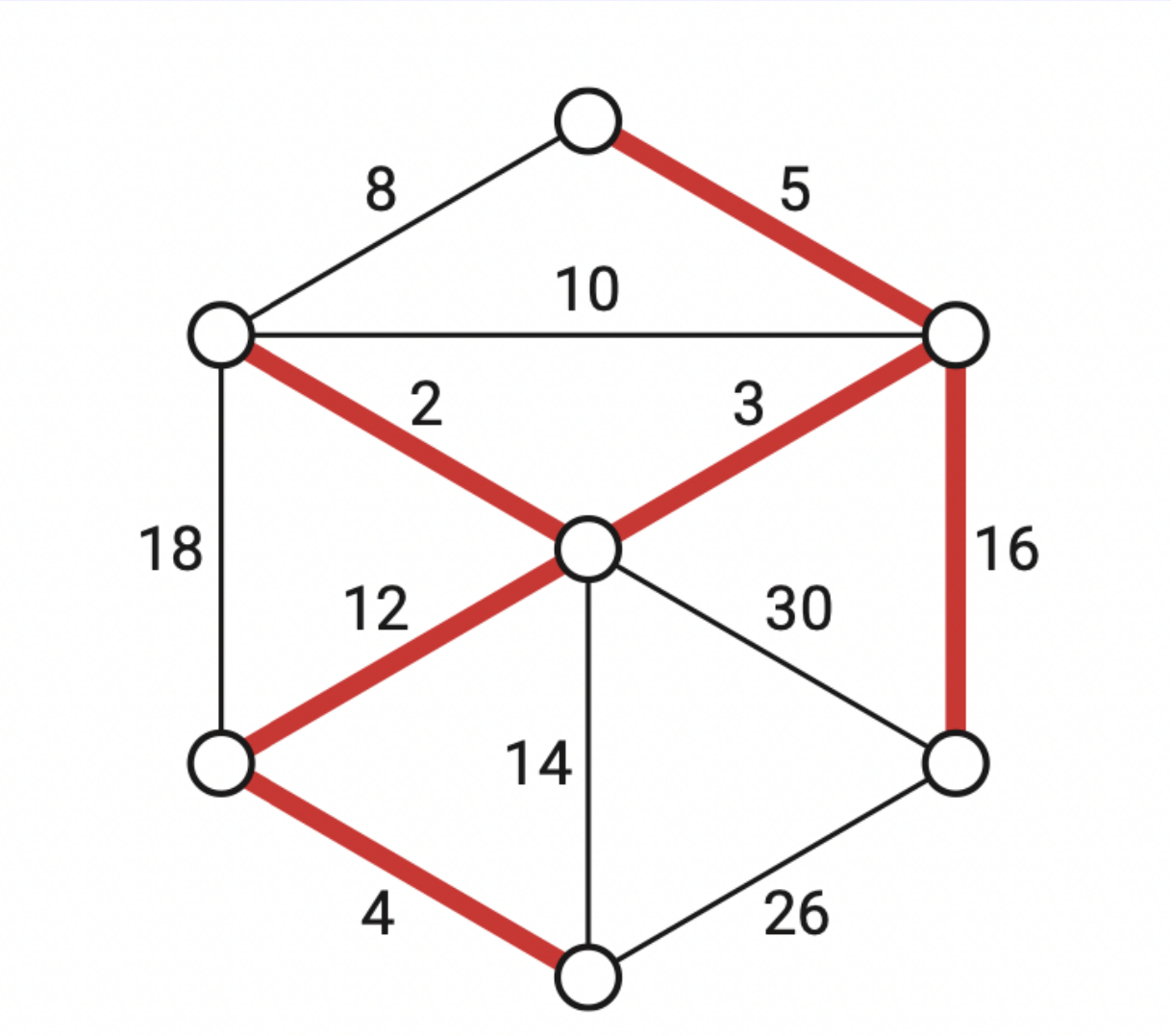


Figure. A weighted graph and its minimum spanning tree.

Uniqueness of MST

It would greatly simplify our discussion if the MST is unique in a given graph G. Fortunately,

if all edges in G have distinct weights, then G has unique MST.

Sketch proof: by contradiction. Goal: If G has two distinct MST, then G must have two edges which have the same weight. Proof is like "exchange argument" of greedy algorithm.

1. Suppose we have two distinct MST T, T'
2. Let e, e' be the minimum weight edges in the two MST $T \setminus T'$, and $T' \setminus T$ respectively. W.l.o.g, assume $w(e) \leqslant w(e')$.
3. $T' \cup \{e\}$ contains exactly 1 cycle C. Let e'' be any edge in C that is **not** in T (e'' may or may not be e'). Because $e'' \notin T$, we have $e'' \neq e$, therefore $e'' \in T' \setminus T$. So $w(e'') \geqslant w(e') \geqslant w(e)$

4. Consider new spanning tree $T'' = T' + e - e''$ (might be equal to T). We have $w(T'') = w(T') + w(e) - w(e'') \leqslant w(T')$. But T' is minimum spanning tree! We must have $=$ instead of $\leqslant$, which means $w(e) = w(e'')$.

To simplify discussion, we make all edges "distinct", by forcing a "tie breaking rule" of equal weight edges, such as: (details do not matter; any systematic tie breaking rule is good enough)

```
ShorterEdge(i, j, k, l)
  if w(i, j) < w(k, l)          then return (i, j)
  if w(i, j) > w(k, l)          then return (k, l)
  if min(i, j) < min(k, l)      then return (i, j)
  if min(i, j) > min(k, l)      then return (k, l)
  if max(i, j) < max(k, l)      then return (i, j)
  ⟨⟨if max(i,j) > max(k,l) ⟩⟩    return (k, l)
```

From now on, we can assume weights are distinct, and therefore the MST is unique.
We can talk about **the** MST.

The only MST algorithm

Just like Whatever-First-Search, there is one generic MST algorithm that can be considered as "mother" of the three popular MST algorithms.

The general MST algorithm maintains a **intermediate spanning forest** F at all times, such that

F is a subgraph of **the** minimum spanning tree of G

Initially, F consists of all the nodes and 0 edges (a forest of one-node trees). The generic algorithm connects trees in F by adding certain edges between them. When the algorithm halts, the forest becomes a single spanning tree (which is the MST).

Which edges to add?

At any time point, the intermediate spanning forest F induces two special types of edges in the rest of the graph:

- An edge is ***useless*** if it is not an edge of F, but both of its endpoints are in the same component of F.
- An edge is ***safe*** if it is the minium weight edge with exactly one endpoint in some component of F. (An edge could be safe for two different components)
- Otherwise, the edge is ***undecided***.

All MST algorithms are based on two observations:

Lemma 1. *(Prim) The minimum spanning tree of G contains every* ***safe*** *edge.*

(Yes, safe edge at **any** stage of the generic algorithm)

We can actually prove a stronger statement:

For any subset $S \subset V$, the MST contains the minimum-weight edge with exact one endpoint in S.

The proof is similar to greedy exchange argument:

Say e is the lightest such edge, and a spanning tree T does not include e, we prove that T is definitely not MST.

Since T is connected, there must be path from one endpoint of e to the other endpoint. Since this path starts with vertice in S and ends with vertice not in S, there must be an edge with exactly one endpoint in S. Let e' be such an edge. We replace e' with e: we get another tree, with smaller cost! (because e is lightest: $w(e) < w(e')$)

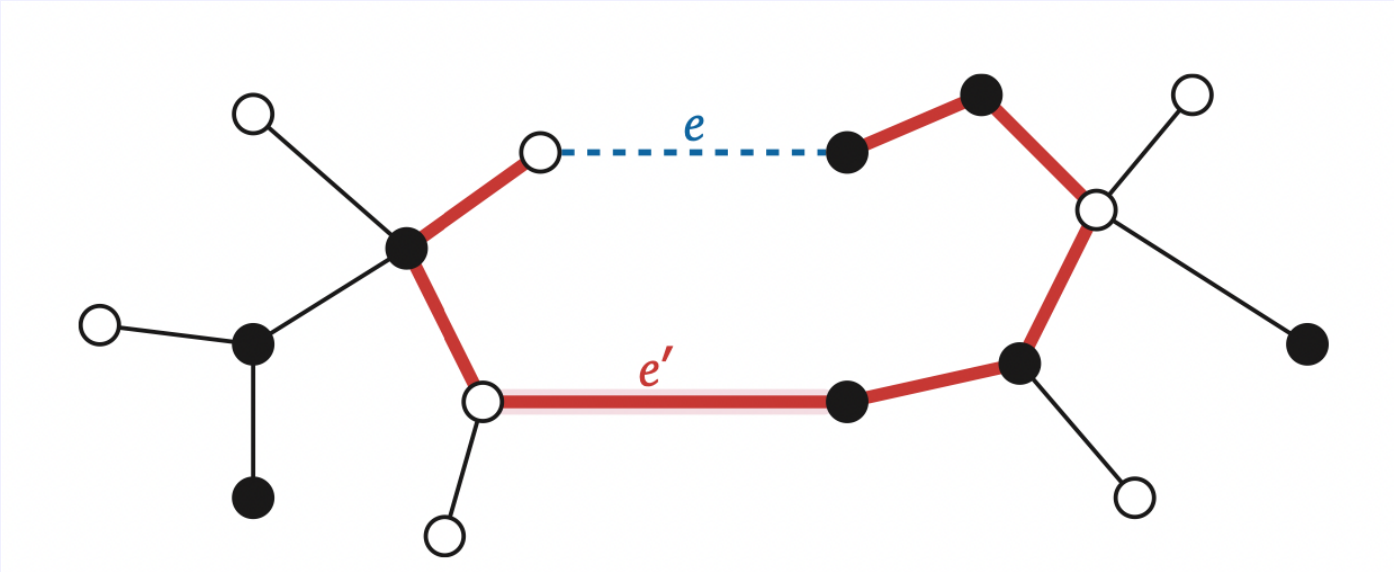


Figure 1. Black vertices are the subset S. e is the lightest edge with exactly one endpoint in S.

Lemma 2. *The MST contains no useless edge.*

Proof: adding any useless edge to F would introduce a cycle. (by definition, useless edges are edges in the same component). □

Generic MST Algorithm:

Repeatedly add **safe** edges to the evolving forest F. If F is not yet connected, there must be at least one safe edge.

Our generic MST algorithm eventually connects F. By induction, lemma 1 says that the resulting tree is the MST.

When we add an edge to F, some undecided edges become useless, and some undecided edges become safe.

To fully specify an algorithm, we must describe **which** safe edges to add in each iteration, and **how** to find such edges.

Boruvka's Algorithm

The simplest MST algorithm: (Boruvka, 1926)

Add **ALL** safe edges and recurse.

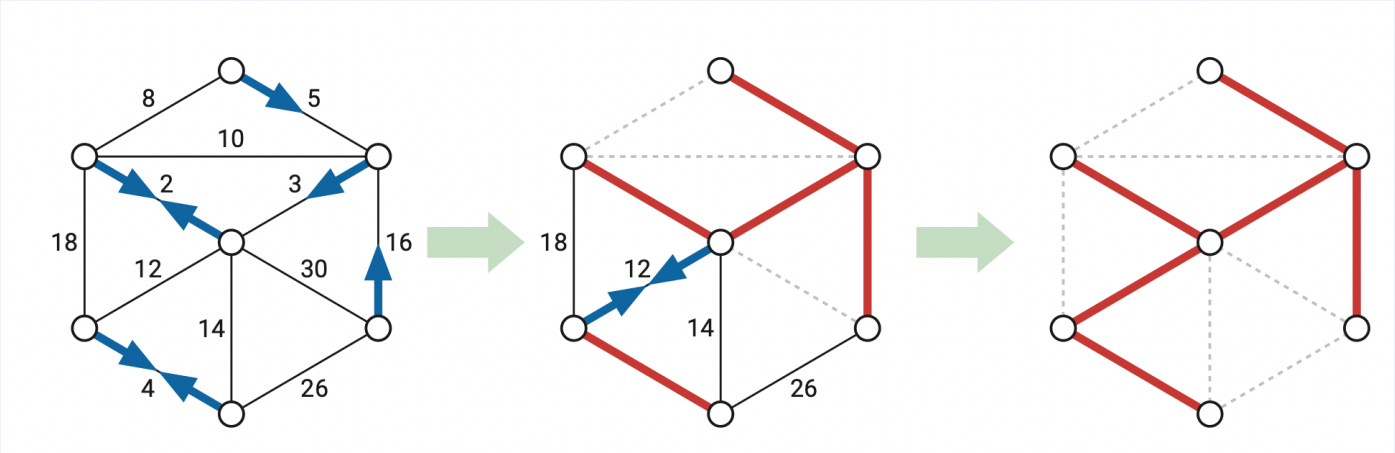


Figure 2. Red edges are in F; dashed lines are useless. Arrows are safe edges.

In more detail, Boruvka's algorithm works like this.

```
BORŮVKA(V, E):
    F = (V, ∅)
    count ← COUNTANDLABEL(F)
    while count > 1
        ADDALLSAFEEDGES(E, F, count)
        count ← COUNTANDLABEL(F)
    return F
```

1. It relies on the CountAndLabel() algorithm to count the number of connected components in a graph, and label each node with its component number:

```
CountComponents(G):
    count ← 0
    for all vertices v
        unmark v
    for all vertices v
        if v is unmarked
            count ← count + 1
            WhateverFirstSearch(v)
    return count
```

2. How do we find all safe edges of F? Suppose F has more than 1 component. The following subroutine computes an array safe[1..v] of safe edges, where safe[i] is the minimum weight edge with one endpoint in the i-th component of F. We brute-force trough all edges and find the safe edges. For an edge uv, if they belong to the same component, then it's either useless or already an edge in F. Otherwise, compare the weight of uv to safe[component(u)] and safe[component(v)] and update the entries if needed.

```
AddAllSafeEdges(E, F, count):
  for i ← 1 to count
      safe[i] ← Null
  for each edge uv ∈ E
      if comp(u) ≠ comp(v)
          if safe[comp(u)] = Null or w(uv) < w(safe[comp(u)])
              safe[comp(u)] ← uv
          if safe[comp(v)] = Null or w(uv) < w(safe[comp(v)])
              safe[comp(v)] ← uv
  for i ← 1 to count
      add safe[i] to F
```

What's the time complexity?

- Each CountAndLabel() costs $O(V+E)$, because it's basically a graph traversal of forest F. Since F is a forest, $E<V$. Thus $O(V+E)=O(V)$
- Each AddAllSafeEdges() costs $O(V+E)$ (again, it's graph traversal, with constant time work per vertice/edge). Since graph G is connected, we have $V<E$. Thus it costs $O(V+E)=O(E)$.
- How many iterations? Each iteartion reduce the #components of F by at least 2x (worst case: two components coalesce into one; best case: every component coalesce into one).

  Thus the number of iterations is $O(\log V)$.
- In total, the Boruvka algorithm costs $O(E \log V)$

Features of Boruvka algorithm for MST:

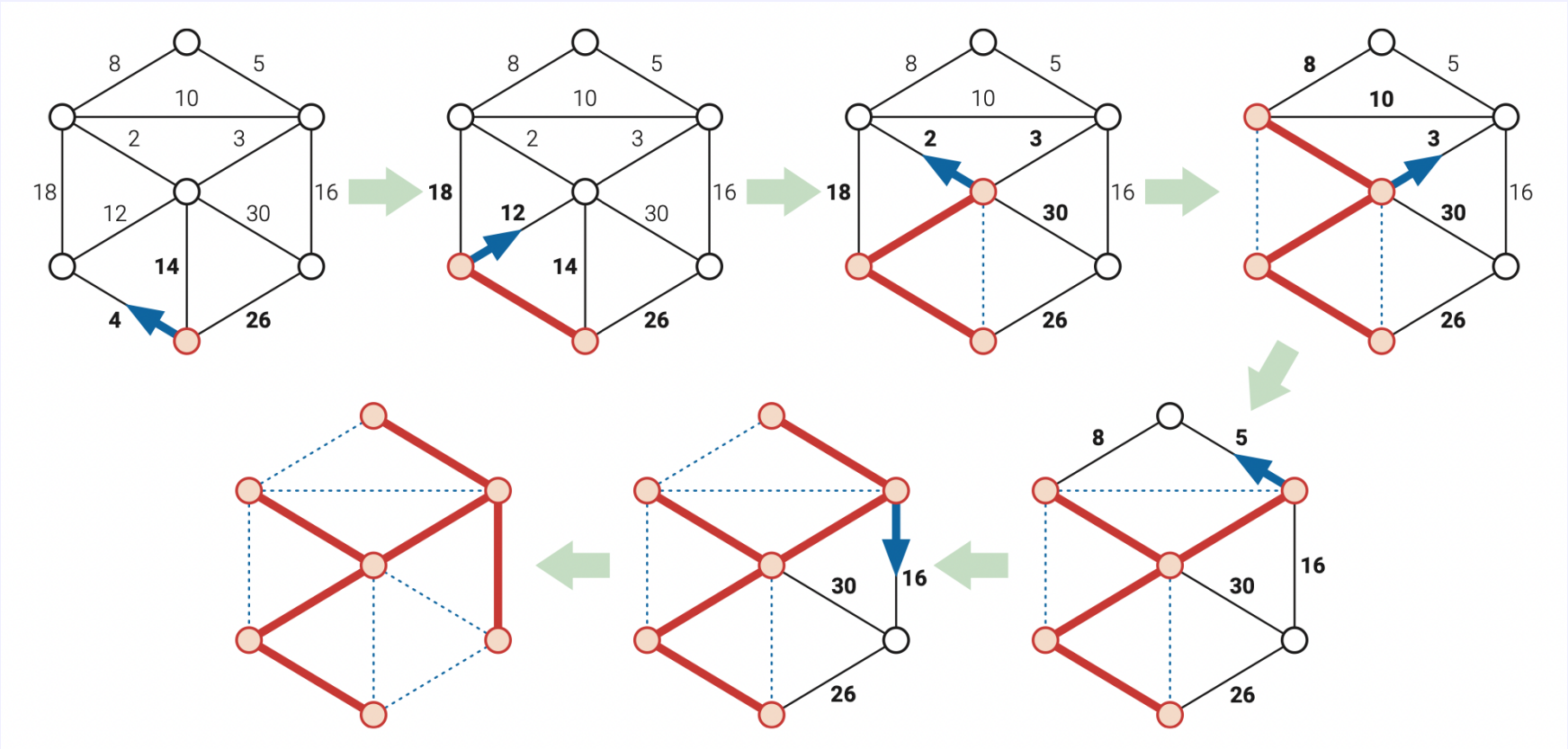
- It's fast; worst case is $O(E \log V)$ but for a lot of graphs it's much faster than that.
- It allows significant parallelism (each component can be processed in parallel)
- Recent very fast MST algorithms are generalization of Boruvka's.
- This should be the choice for implementation of MST.

Jarnik's (Prim's) Algorithm

In Jarnik's (Prim's) algorithm, the intermediate forest F has only one non-trivial component; all other components are isolated single vertices.

The algorithm repeats the following step until T spans the whole graph:

Jarnik: repeatedly add T's safe edge to T.

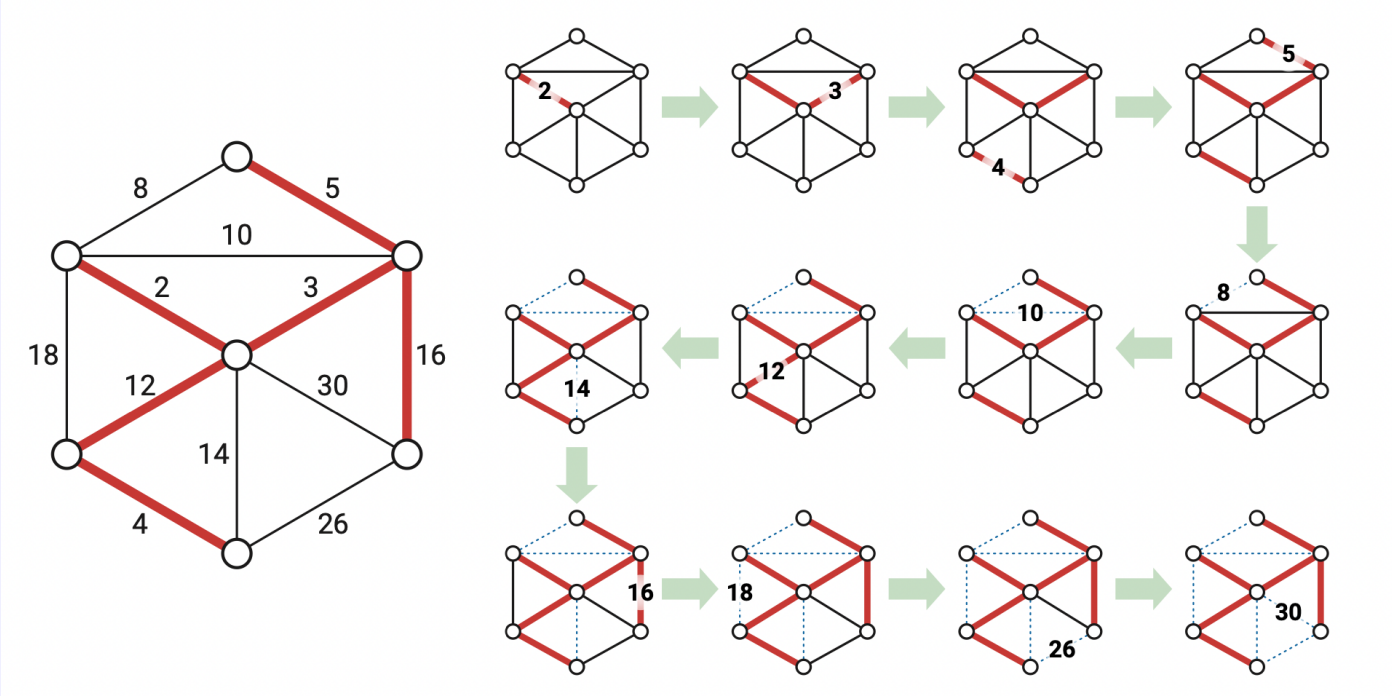


To implement Jarnik's algorithm we keep all the edges adjacent to T in a priority queue. When we take an edge e from the priority queue, we check whether both endpoints are in T. If not, we add the vertice at the other end of e to T, and all its edges into the priority queue. This is the best-first search!.

Time complexity: $O(E \log E) = O(E \log V)$.

Kruskal's Algorithm

Kruskal: Scan all edges by increasing weight; if an edge is safe, add it to F.



We can first sort the edges in increasing order which costs $O(E \log E) = O(E \log V)$. [Turns out this dominates all other costs].

```
KRUSKAL(V, E):
    sort E by increasing weight
    F ← (V, ∅)
    for each vertex v ∈ V
        MAKESET(v)
    for i ← 1 to |E|
        uv ← ith lightest edge in E
        if FIND(u) ≠ FIND(v)
            UNION(u, v)
            add uv to F
    return F
```

A little detour:

We need a **set-partition** data structure in Kruskal that can efficiently do:

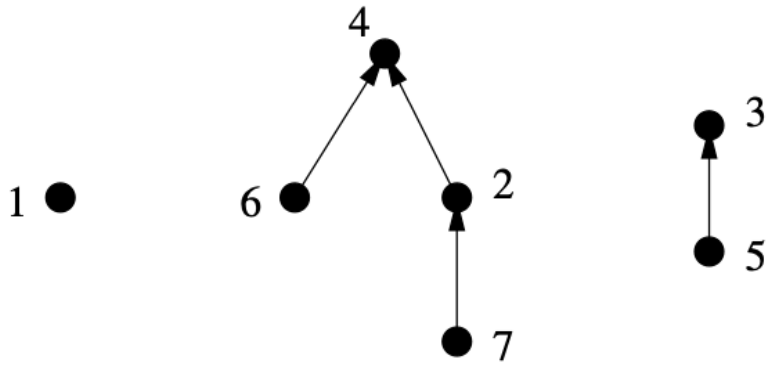
1. MakeSet(v): make a set out of a single node
2. Find(v): return the set that v belongs to.
3. Union(u,v): union the sets that u and v belong to.

Naive data structure candidates:

1. We just have labels attached to every node. MakeSet(v) and Find(v) takes constant time. However Union(u,v) takes linear time.
2. Graph: connected component is a set. How to Find & Union? Need full graph traversal to identify connected components.

This is actually quite common requirement for many applications. A particularly efficient and simple data structure called **union-find** fits the bill.

Union-find partitions elements into disjoint sets. Each element is in exactly one set. Union-find represents each set as a backward **tree**, with pointer to its parent. Each node contains an element. The root is the representative of the set: e.g. find(u) will return the root of the set that u belongs to.



(l)

1	2	3	4	5	6	7
0	4	0	0	3	4	2

(r)

Now, MakeSet(v) is pretty simple; just assign parent of v to 0.

For Find(v), we can just trace back using the parent array.

For Union(u,v): link one of u,v to the other (e.g. u adopting v as child).

How efficient are these operations? Find(v) is proportional to the height of tree; so we would like short trees. So when we Union(u,v), to make it as short as possible, we let the taller tree adopt the shorter tree.

Analysis: we can easily prove the height of any tree is at most $O(\log n)$. So the most time consuming operation is Find(v) which costs $O(\log n)$.

In fact this can be made even faster by collapsing the tree when we Find(v): just relinking the parent of all nodes in path v->root(v) to directly to root. The tree becomes very flat.