Topic 19: Minimum Spanning Trees and Union-Find

(CLRS 23, 21)

CPS 230, Fall 2001

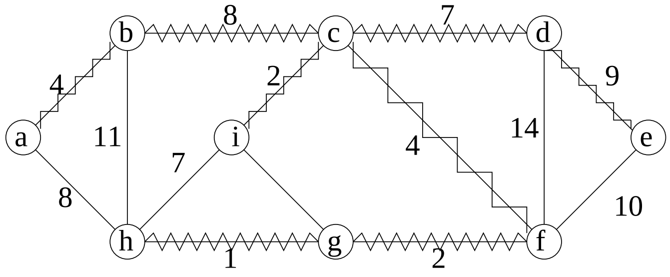
1 Graphs

- Last time we defined (weighted) graphs (either undirected or directed) and introduced basic graph vocabulary (vertex, edge, degree, path, connected components of undirected graphs, strongly connected components of directe graphs, ...).
- We also discussed adjacency list and adjacency matrix representation
  - Unless stated otherwise, we will use adjacency list representation, using $O(|V| + |E|)$ space.
- We discussed $O(|V| + |E|)$-time breadth-first (BFS) and depth-first search (DFS) algorithms and how they can be used to compute, e.g., connected components, shortest path distances in unweighted graphs, and topological sorting.
- We will now start discussing more complicated problems/algorithms on weighted graphs.

2 Minimum Spanning tree (MST)

- Problem: Given a connected, undirected graph $G = (V, E)$, where each edge (u, v) has weight $w(u, v)$. Find a tree $T \subseteq E$ that connects all the vertices in V such that it has minimum weight $w(T) = \sum_{(u,v)\in T} w(u, v)$.
- Note: Problem is to find a *spanning tree* (acyclic set connecting all vertices) of *minimum weight*. (We use the term *minimum spanning tree* as shorthand for *minimum-weight spanning tree*).
- MST problem has many applications in networking, communications, scheduling, ... For example, think about connecting cities with the minimum amount of wire (cities are vertices, weight of edges are distances between city pairs).

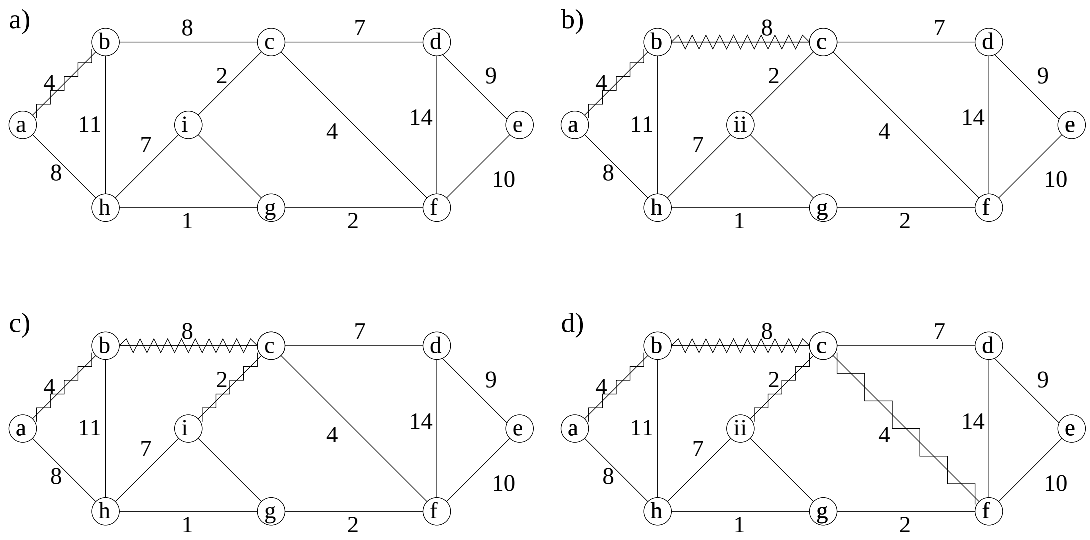
- Example:



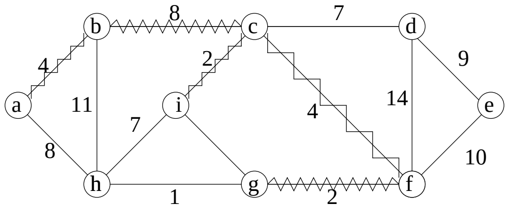
  - Weight of MST is $4 + 8 + 7 + 9 + 2 + 4 + 1 + 2 = 37$
  - MST is not unique: e.g. (b, c) can be exchanged with (a, h)

2.1 PRIM's algorithm

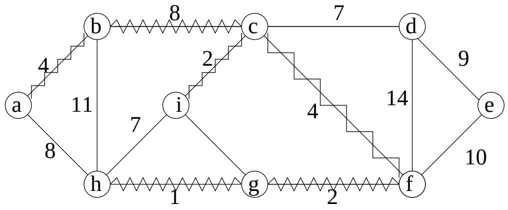
- *Greedy* algorithm for computing MST:
  - Start with spanning tree containing arbitrary vertex r and no edges.
  - Grow a spanning tree by repeatedly adding the minimum weight edge that connects some vertex in the current tree to some vertex not in the current tree.
- On the example, the greedy algorithm would work as follows (starting at vertex a):



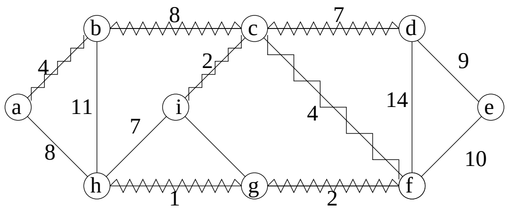
e)



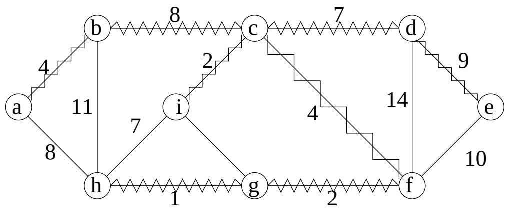
f)



g)



h)



- Implementation:
  - To find the minimum weight edge connected to the current tree, we maintain a priority queue on the vertices not in the tree. The priority of a vertex is the weight of the minimum weight edge that connects it to the current tree. (The priority is $+\infty$ if there is no connecting edge.)
  - We also maintain a pointer from the adjacency list entry of v to the entry for v in the priority queue.

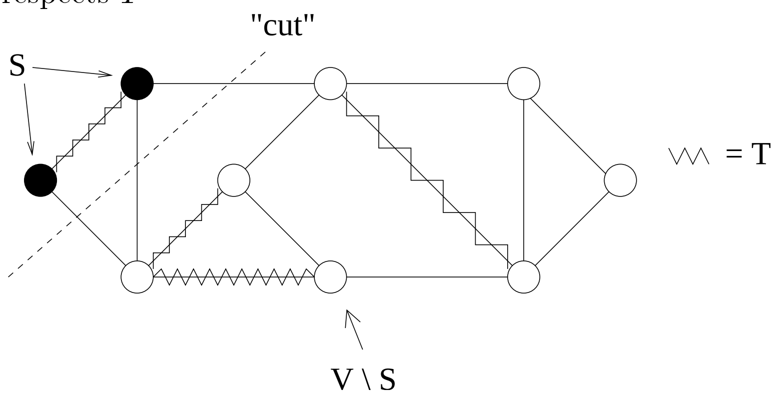
```
PRIM(r)

For each v ∈ V DO
    INSERT(Q, v, ∞)
OD
DECREASE-KEY(Q, r, 0)
WHILE Q not empty DO
    u = DELETE-MIN(Q)
    For each (u, v) ∈ E DO
        IF v ∈ Q and w(u, v) < key(v) THEN
            visit[v] = u
            DECREASE-KEY(Q, v, w(u, v))
        FI
    OD
OD
```

- Analysis:
  - While loop runs $|V|$ times $\Longrightarrow$ we perform $|V|$ DELETE-MIN's
  - We perform at most one DECREASE-KEY for each of the $|E|$ edges
    $\Longrightarrow$ $\mathit{Time} = \Theta(|V| \times T(\text{DELETE-MIN}) + |E| \times T(\text{DECREASE-KEY}))$

Priority Queue	T(Delete-Min)	T(Decrease-Key)	Total time
array	$\Theta(\|V\|)$	$\Theta(1)$	$\Theta(\|V\|^2)$
binary heap	$\Theta(\log \|V\|)$	$\Theta(\log \|V\|)$	$\Theta(\|E\| \log \|V\|)$
Fibonacci heap	$\Theta(\log \|V\|)$	$\Theta(1)$	$\Theta(\|V\| \log \|V\| + \|E\|)$

- Correctness:
  - As discussed previously, when we design a greedy algorithm, the hard part is often to prove that it works correctly.
  - We will prove a theorem below that allows us to prove the correctness of a general class of greedy algorithms. First some definitions:
    * A *cut* S is a partition of V into sets S and $V \setminus S$
    * A *edge* (u, v) *crosses a cut* S if $u \in S$ and $v \in V \setminus S$ *or* $v \in S$ and $u \in V \setminus S$
    * A *cut* S *respects a set* $T \subseteq E$ if no edge in T crosses the cut

    Example: Cut S respects T

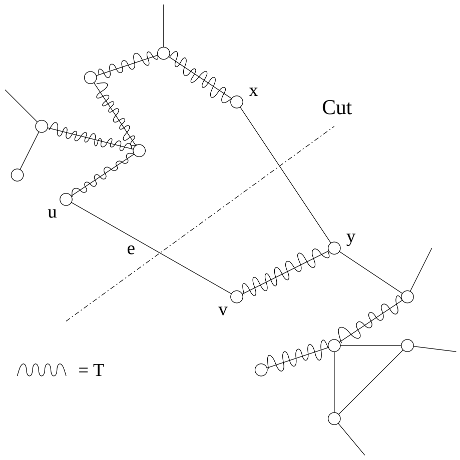


- *Theorem*: If $G = (V, E)$ is a graph such that $T \subseteq E$ is subset of some MST of G, and S is a cut respecting T **then** there is an MST for G that contains T and the minimum weight edge $e = (u, v)$ that crosses S.

- Note: The correctness of Prim's algorithm follows from the Theorem by induction: At each step, let the cut S consist of the edges T in the current tree. Therefore, at every step of the way in Prim's algorithm, the current set T of edges chosen is a subset of some MST. Thus, at the end, when T has $n - 1$ edges, T must be an MST.

- Proof of theorem:
  - Let T^* be MST containing T. Let $e = (u, v)$ be the minimum weight edge that crosses S.
  - If $e \in T^*$ we are done.
  - If $e \notin T^*$:
    * Since T^* is a spanning tree and is therefore connected, there must be (at least) one other edge $(x, y) \in T^*$ crossing the cut S such that there is a unique path

from u to v in T^*.



  * This path together with e forms a cycle
  * If we remove edge (x, y) from T^* and add e instead, we still have spanning tree
  * The new spanning tree must have the same weight as T^* since T^* is an MST and $w(u, v) \leq w(x, y)$
    $\Longrightarrow$ There is an MST containing T and e.

- The Theorem allows us to describe a very abstract greedy algorithm for MST:

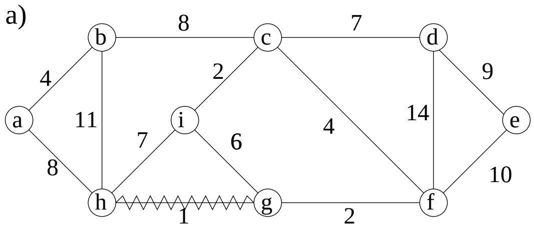
```
T = ∅
While |T| ≤ |V| − 1 DO
    Find cut S respecting T
    Find the minimum weight edge e that
    crosses S
    T = T ∪ {e}
OD
```

  - Prim's algorithm follows this abstract algorithm.

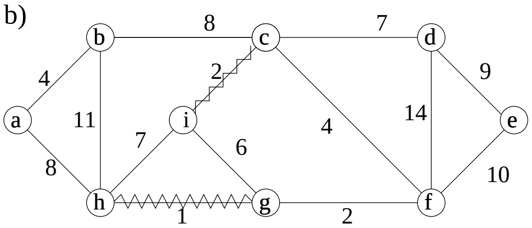
3 Kruskal's Algorithm

- Kruskal's algorithm is another implementation of the abstract algorithm.
- Idea in Kruskal's algorithm:
  - Start with $|V|$ trees (one for each vertex).
  - Consider edges E in increasing order; add edge if it connects two trees (in which case we merge the two trees into one).
- Example:

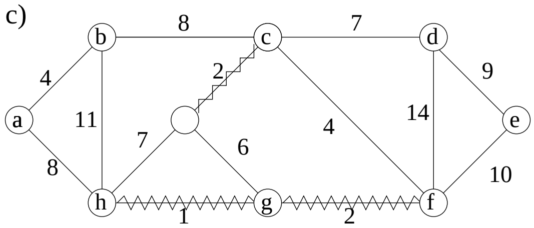
a)



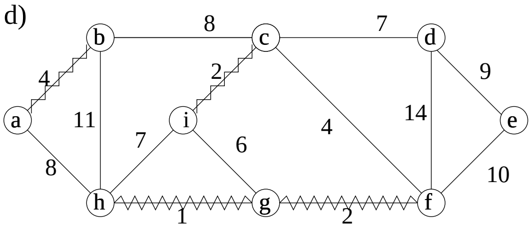
b)



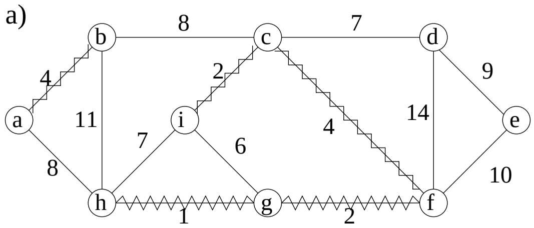
c)



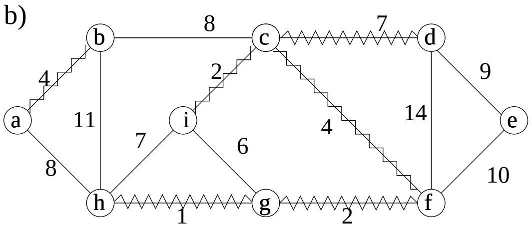
d)



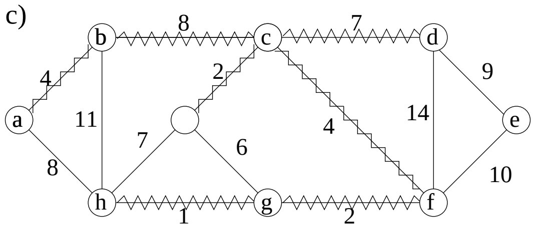
a)



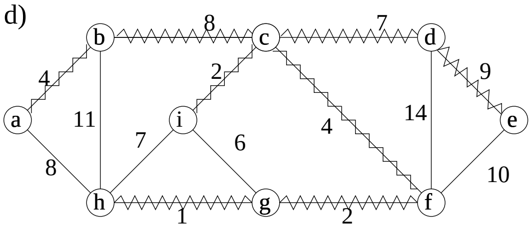
b)



c)



d)



- Correctness of Kruskal's algorithm follows from Theorem: Suppose the minimum weight edge connects two trees. Let the cut S consist of the vertices in one of the two trees. Let T be the current set of edges chosen. The cut S respects T. By the theorem, since the minimum weight edge crosses the cut S, there is an MST that contains T and the minimum weight edge.

- Therefore, at every step of the way in Kruskal's algorithm, the current set T of edges chosen is a subset of some MST. Thus, at the end, when T has $n-1$ edges, T must be an MST.

- Implementation:

KRUSKAL

$T = \emptyset$
FOR each vertex $v \in V$ DO

  MAKE-SET(v)

OD
Sort edges of E in increasing order by weight
FOR each edge $e = (u, v) \in E$ in order DO

  IF FIND-SET(u) $\neq$ FIND-SET(v) THEN

    $T = T \cup \{e\}$

    UNION-SET(u, v)

  FI

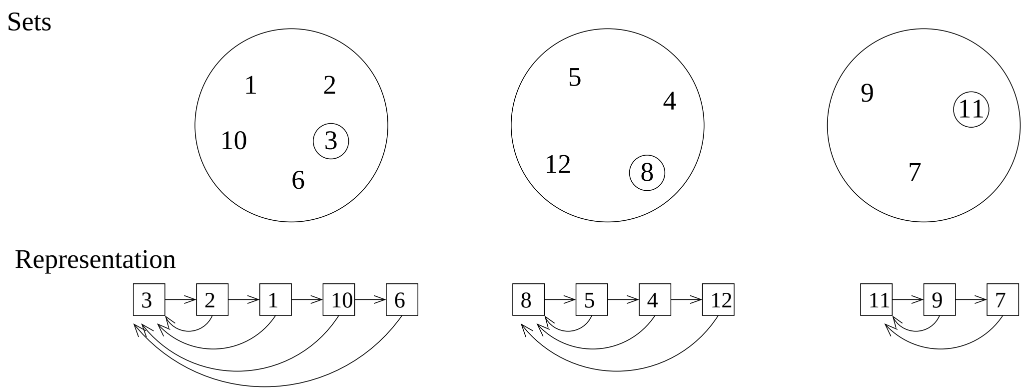
OD

- We need (Union-Find) data structure that supports:
  - MAKE-SET(v): Create set consisting of v
  - UNION-SET(u, v): Unite set containing u and set containing v
  - FIND-SET(u): Return unique representative for set containing u
- We use $O(|E| \log |E|)$ time to sort edges, and we perform $|V|$ MAKE-SET, $|V| - 1$ UNION-SET, and $2|E|$ FIND-SET operations.

3.1 Union-Find

- Simple solution to Union-Find problem (maintain set system under FIND-SET and UNION-SET)
  - Maintain elements in same set as a linked list with each element having a pointer to the first element in the list (unique representative)

    Example:



  - MAKE-SET(v): Make a list with one element $\Longrightarrow$ $O(1)$ time.
  - FIND-SET(u): Follow pointer and return unique representative $\Longrightarrow O(1)$ time.

– UNION-SET(u, v): Link first element in list with unique representative FIND-SET(u) after last element in list with unique representative FIND-SET(v)
$\Longrightarrow$ $O(|V|)$ time (since we have to update all unique representative pointers in the list containing u)

- With this simple implementation, the $|V| - 1$ UNION-SET operations in Kruskal's algorithm may take $O(|V|^2)$ time.

- We can improve the performance of UNION-SET with a very simple modification: always link the smaller list after the longer list (i.e., update the pointers of the smaller list)

  – One UNION-SET operation can still take $O(|V|)$ time, but the $|V|-1$ UNION-SET operations collectively take a total of $O(|V| \log |V|)$ time.

    * Total time is proportional to number of unique representative pointer changes.
    * Consider any element u:
    After pointer for u is updated, u belongs to a list of size at least double the size of the list it was in previously.
    $\Longrightarrow$ After k pointer changes, u is in list of size at least 2^k.
    $\Longrightarrow$ u's pointer can be changed at most $\log |V|$ times.

- With improvement, Kruskal's algorithm runs in time $O(|E| \log |E| + |V| \log |V|) = O((|E|+|V|) \log |E|) = O(|E| \log |V|)$, like Prim's algorithm with the standard priority queue implementation.

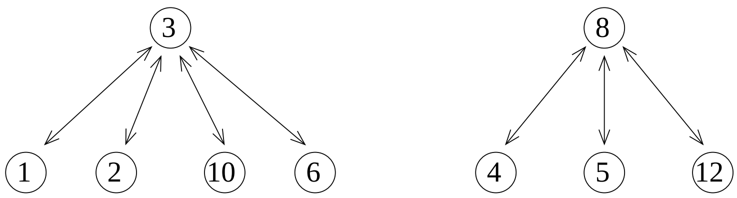
- Note: Very recently an $O(|V| + |E|)$ randomized minimum spanning tree algorithm has been developed.

4 Improved Union-Find

- It turns out that Union-Find can be improved (but without leading to an improvement of Kruskal's algorithm, unless the edges are provided for free in sorted order).

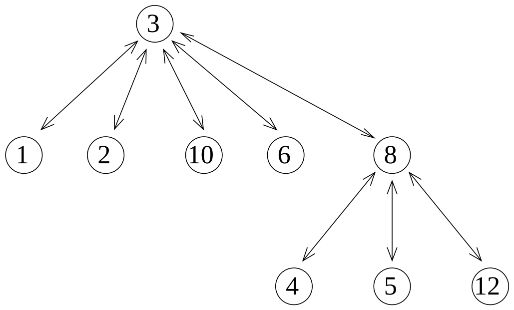
  – Linked list representation can also be viewed as trees of height 1.

  Example:



  – Instead of updating root pointers when performing UNION-SET, we could just link one tree below the root of the other.

  Example: UNION-SET(2,5)

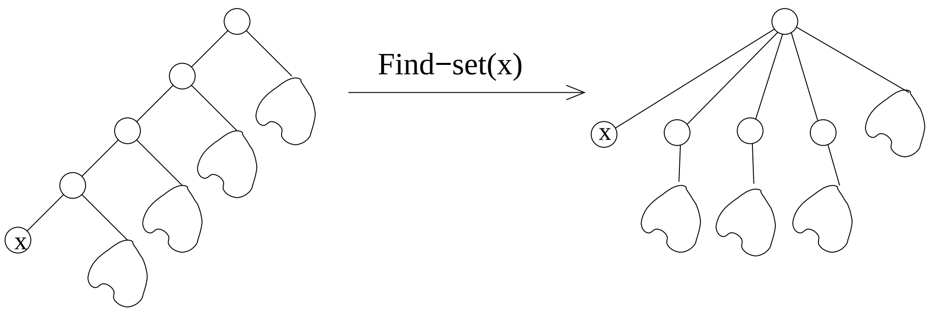


UNION-SET and FIND-SET take $O(\log |V|)$ if we always insert the smaller tree below the larger tree, since every tree will have height $O(\log |V|)$ (prove it!)
$\Longrightarrow$ $|E|$ FIND-SET operations takes $O(|E| \log |V|))$ time.

– If we furthermore perform *path compression*, $|E|$ Find-set operations can be performed even faster

  Path-compression: During each FIND-SET, as we follow the path to the root (the set's unique representative), we link the traversed nodes directly to the root:

  Example:



  Note that a lot of paths are shortened (decreasing time spent on future FIND-SET operations) without using extra time

  It can be shown that $O(|E| \log^* |V|)$ is the total time used on the $|E|$ FIND-SET operations

4.1 $\log^* n$, an extremely slowly growing function

- Consider $g(n) = \begin{cases} 2^1 & \text{if } i = 0 \\ 2^2 & \text{if } i = 1 \\ 2^{g(n-1)} & \text{if } i \geq 2 \end{cases}$

$$\begin{aligned} g(0) &= 2 \\ g(1) &= 2^2 = 4 \\ g(2) &= 2^{2^2} = 2^4 = 16 \\ g(3) &= 2^{2^{2^2}} = 2^{16} = 65536 \\ &\vdots \\ g(i) &= 2^{2^{\cdot^{\cdot^{2}}}} \qquad \text{(Stack of 2s of height } i+1\text{)} \end{aligned}$$

$g(n)$ is an *extremely* fast growing function.

- Define $\log^{(i)} n = \begin{cases} n & \text{if } i = 0 \\ \log \log^{(i-1)} n & \text{otherwise} \end{cases}$

- $\log^* n = \min\{i \geq 0 \mid \log^{(i)} n \leq 1\}$.
  $\Longrightarrow$ $\log^* n$ is minimum number of times we need to take log to get below 1.
  $\Longrightarrow$ $\log^* n$ is inverse of $g(n)$.
  $\Longrightarrow$ $\log^* n$ *extremely* slow growing function.

- $\log^* n \leq 5$ for all practical values of n.

- One can even prove that with path-compression the total time spent on $|E|$ FIND-SET operations is $O(|E| \cdot \alpha(|V|))$, where $\alpha(n)$ (a functional inverse of Ackermann's function) is a function growing even more slowly than $\log^* n$

  - $\alpha(n) < 4$ for all practical values of n.