CS 255, Spring 2014, SJSU

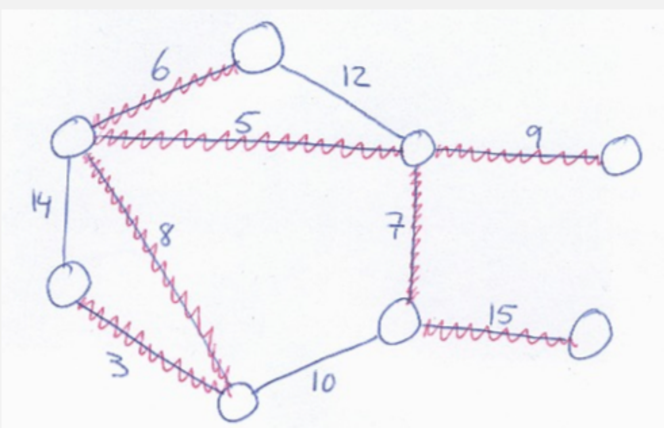
Minimum Spanning Trees

Fernando Lobo

Minimum Spanning Tree (MST)

- Input: a connected undirected graph $G = (V, E)$. Each edge $(u, v) \in E$ has a weight $\rightarrow w(u, v)$.
- Output: a set of edges $A \subseteq E$ such that:
  1. A is a tree that connects all the vertices of the graph (A is a spanning tree), and
  2. $w(A) = \sum_{(u,v) \in A} w(u, v)$ is minimum.
- Has many practical applications.

Example of a MST



Cost: $6 + 5 + 9 + 7 + 15 + 8 + 3 = 53$

Properties of a MST

- Has $|V| - 1$ edges.
- Has no cycles.
- May not be unique.

We shall see two algorithms for obtaining a MST. Both are greedy algorithms.

Generic greedy algorithm for obtaining a MST

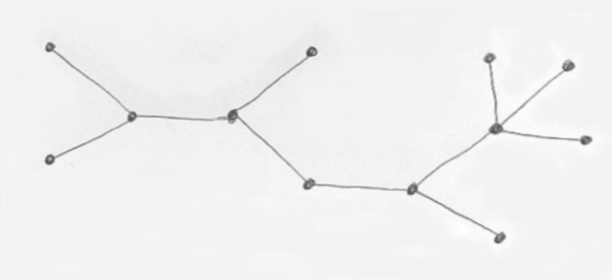
- Incrementally build a set of edges A that is a subset of some MST.
- Initially $A = \emptyset$
- At each iteration we add an edge to A keeping the invariant that A remains a subset of a MST. We call those edges *safe*.
- When $|A| = |V| - 1$, A will be a MST.

Pseudocode

Generic-MST(G)
  $A = \emptyset$
  while A is not a spanning tree
    find an edge (u, v) that is safe for A
    $A = A \cup \{(u, v)\}$
  return A

Problem has optimal substructure

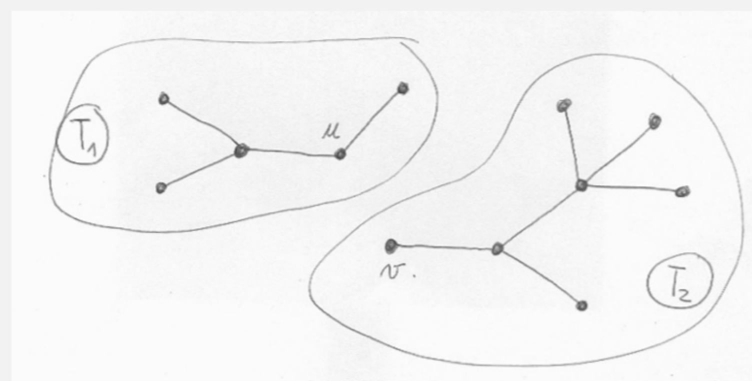
MST T:



(other graph edges not shown)

Problem has optimal substructure

If we remove an edge $(u, v) \in T$, we are left with two subtrees: T_1 and T_2



Problem has optimal substructure

Theorem

- T_1 is a MST of graph $G_1 = (V_1, E_1)$, the subgraph of G induced by the vertices in T_1.
  - V_1 = vertices of T_1
  - $E_1 = \{(x, v) \in E : x, y \in V_1\}$
- Same thing for T_2.

Proof

By contradiction

- $w(T) = w(T_1) + w(T_2) + w(u, v)$
- If there was a spanning tree T_1' in G_1 with less cost than T_1, then $T' = \{T_1' \cup T_2 \cup (u, v)\}$ would be a spanning tree of G with less cost than T, which is a contradiction. $\square$

Greedy choice property

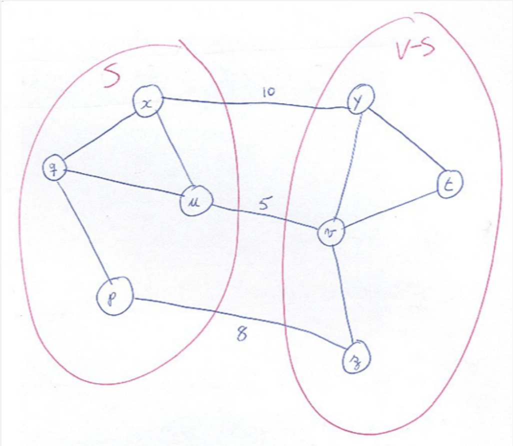
- The problem has the *greedy choice property*: There's a sequence of local optimal choices that can be made that will yield a global optimal solution.

Greedy choice property

We need the following definitions:

- A cut $(S, V - S)$ is a partition of the vertices V into two sets: S and $V - S$.
- An edge $(u, v) \in E$ crosses the cut $(S, V - S)$ if one of its endpoints is in S and the other is in $V - S$.
- A cut $(S, V - S)$ respects a set of edges A if and only if there's no edge in A that crosses the cut.
- An edge is safe if its weight is minimum among all the edges that cross a cut. There can be several safe edges.

Example



- Cut $(S, V-S)$
- There are 3 edges that cross the cut: $(x, y), (u, v), (p, z)$.
- (u, v) is a safe edge for the cut $(S, V-S)$.

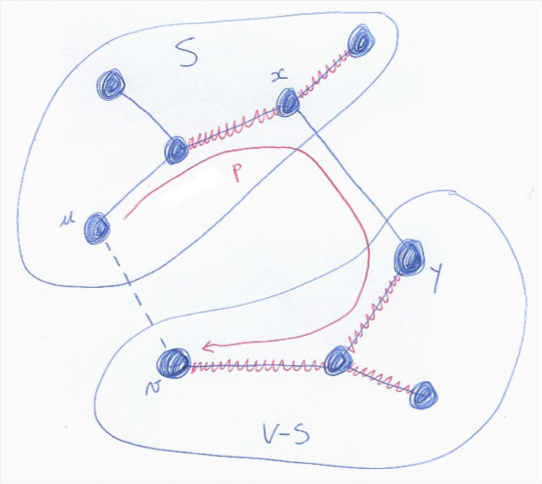
Definition of a safe edge

- Let A be a set of edges $\subseteq$ MST.
- (u, v) is a safe edge for A iff $A \cup \{(u, v)\} \subseteq$ MST.

Theorem:

- Let A be a subset of a MST, and let $(S, V-S)$ be a cut that respects A. If (u, v) is a safe edge for the cut $(S, V-S)$, then (u, v) is a safe edge for A.
  - In other words, (u, v) belongs to a MST.

Proof



$A \rightarrow$ red edges. (u, v) is a safe edge.

Proof (cont.)

- Let T be a MST that contains A and does not include (u, v).
- If it doesn't include (u, v), then it has to include at least some other edge that crosses the cut $(S, V-S)$. Let (x, y) be such an edge.
- Then (x, y) is on the path $u \rightsquigarrow v$ in T because there must be a single path between any two nodes of a tree.
  (See path p in figure.)

Proof (cont.)

- If we remove (x, y), we break T into 2 subtrees. Adding (u, v) joins back the two subtrees and we obtain a new spanning tree $T' = T - \{(x, y)\} \cup \{(u, v)\}$.
- Since (u, v) is a safe edge,

$$\begin{aligned} &\Longrightarrow \quad w(u, v) \leq w(x, y) \\ &\Longrightarrow \quad w(T') \leq w(T) \\ &\Longrightarrow \quad T' \text{ is a MST} \quad \square \end{aligned}$$

Generic algorithm for obtaining a MST

The previous argument gives rise to the generic algorithm that we've seen.

Generic-MST(G)
    $A = \emptyset$
    while A is not a spanning tree
        find an edge (u, v) that is safe for A
        $A = A \cup \{(u, v)\}$
    return A

Generic algorithm for obtaining a MST

- The interesting part is how to find the safe edges.
- We shall see two algorithms that are concrete exemples of the generic algorithm.
  1. Prim's algorithm: Starts with any given vertex s and grows the tree A from s. At each iteration, adds an edge (u, v) where one of the endpoints belongs to A and has minimum cost.
  2. Kruskal's algorithm: Starts with $A = \emptyset$. Sorts the edges of the graph in increasing order of weight. Go through the sorted list of edges and at each iteration add the edge (u, v) to A as long as it doesn't introduce a cycle.

Prim's algorithm

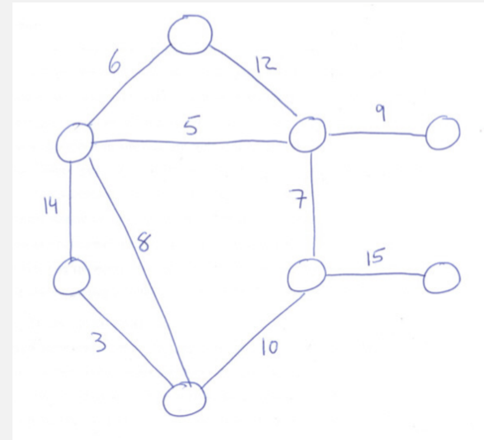
- Initially $A = \emptyset$
- Keep $V - A$ in a priority queue Q.
- The key of each node in the queue indicates the minimum cost of adding that node to any given node in A.
- The algorithm stops when the queue Q becomes empty. The MST A will be:

$$A = \{(v, v.\pi) : v \in V - \{s\}\}$$

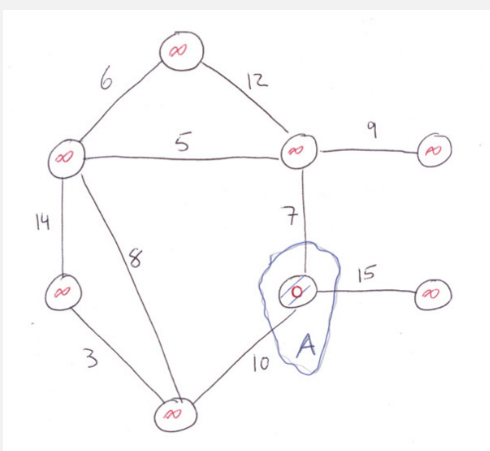
Pseudocode

MST-PRIM(G, w, s)
  for each $u \in G.V$
    $u.key = \infty$
    $u.\pi = \text{NIL}$
  $s.key = 0$
  $Q = G.V$
  while $Q \neq \emptyset$
    $u = \text{EXTRACT-MIN}(Q)$
    for each $v \in G.Adj[u]$
      if $v \in Q$ and $w(u, v) < v.key$
        $v.\pi = u$
        $v.key = w(u, v)$

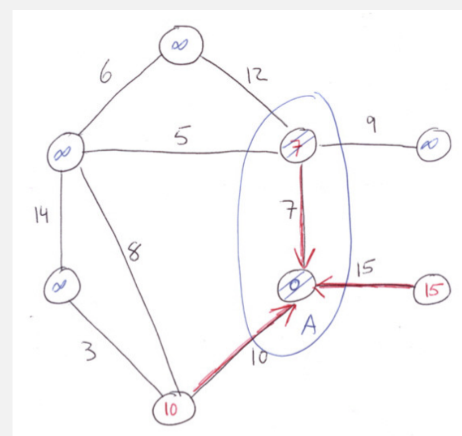
Example



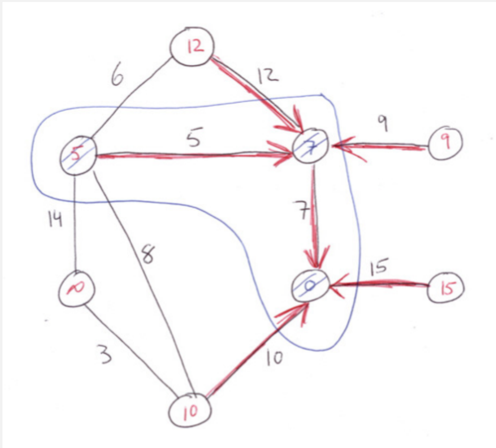
Initialization



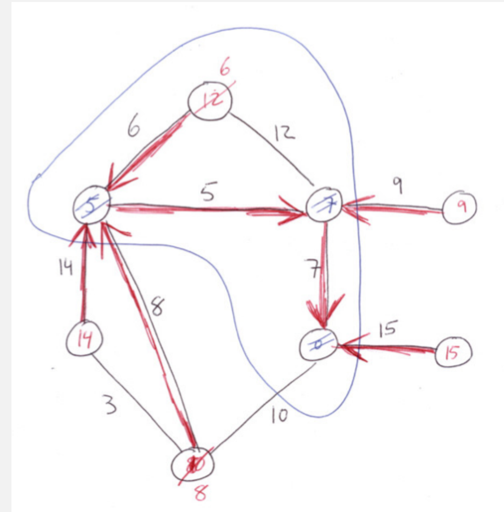
1st iteration of the **while** loop



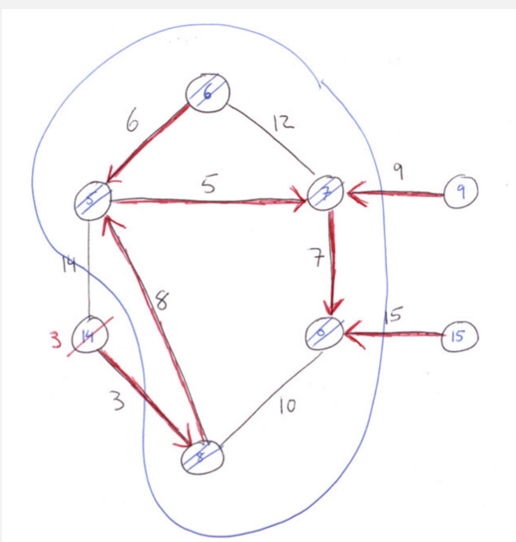
2nd iteration of the **while** loop



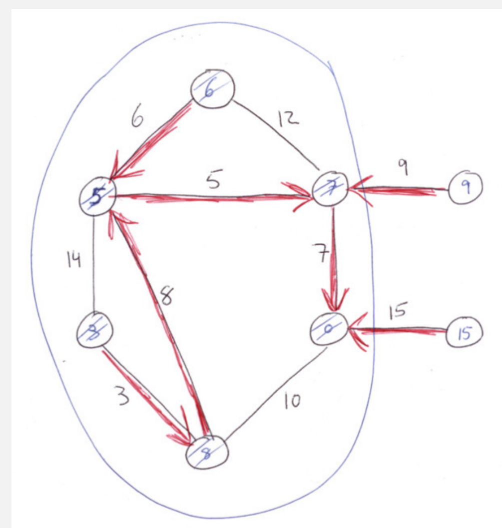
3rd iteration of the **while** loop



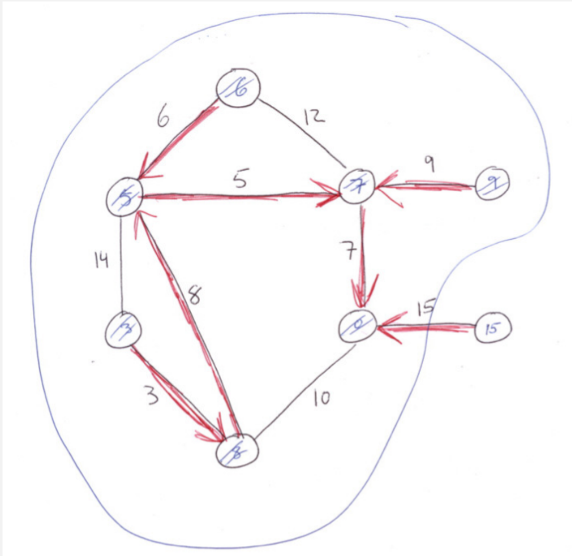
4th iteration of the **while** loop



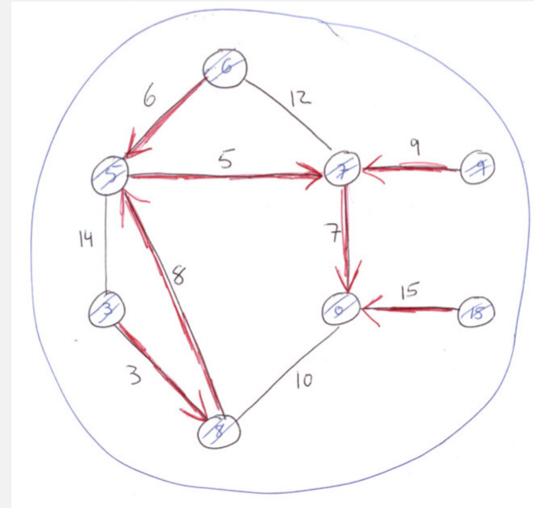
5th iteration of the **while** loop



6th iteration of the **while** loop



7th iteration of the **while** loop



Running time of Prim's algorithm

- Depends on the priority queue implementation.
- If implemented with a binary heap (ch. 6 of textbook):
  - Initialization $\rightarrow$ BUILD-HEAP $\rightarrow$ $O(V)$
  - **while** loop is executed $|V|$ times.
  - $|V|$ EXTRACT-MINS $\rightarrow$ $O(V \lg V)$
  - At most $|E|$ DECREASE-KEYS $\rightarrow$ $O(E \lg V)$
  - Total $= O(V \lg V + E \lg . V) = O(E \lg V)$
- The test **if** $v \in Q$ can be checked in constant time if a bit vector is maintained telling which nodes are in Q.

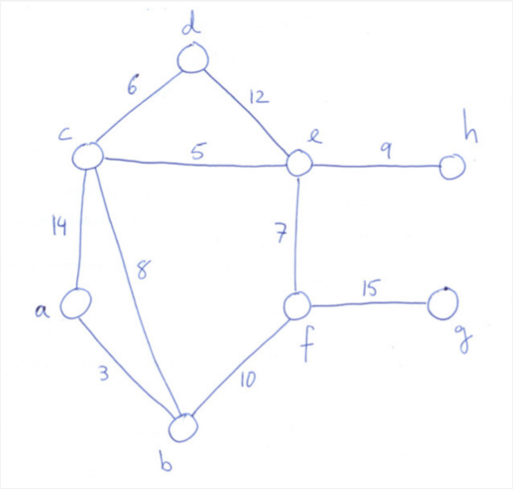
Running time of Prim's algorithm

- It's possible to get a better running time if the priority queue is implemented with a Fibonacci Heap.
  (we didn't go over them. They are described in ch. 19 of your textbook in case you want to know more about it.)
  - Allows $|E|$ DECREASE-KEYS in $O(E)$ time, amortized.
    $\implies T_{Prim} = O(V \lg V + E)$
  - Substantial speedup for sparse graphs.

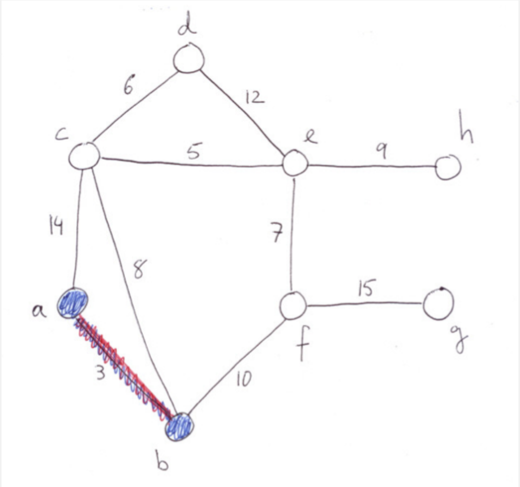
Kruskal's algorithm

- Initially $A = \emptyset$. At the end A will be a MST.
- Sort the edges of the graph by increasing order of weight.
- Go through the sorted list of edges, one by one, and add the edge to A as long as it does not produce a cycle.
- As opposed to Prim's algorithm, Kruskal's algorithm maintain's a forest. Initially the forest has $|V|$ trees, one for each vertex. At the end, the forest consists of a single tree which is a MST.

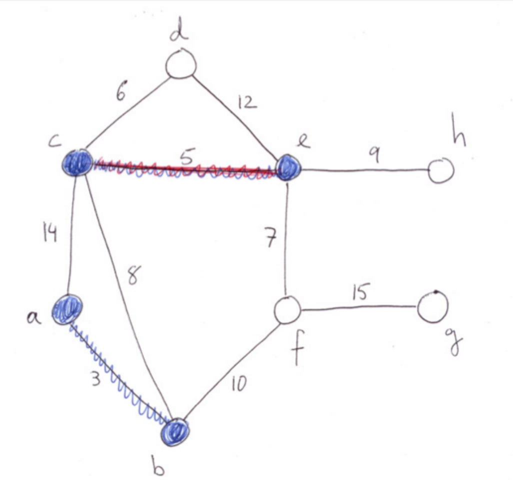
Example: Initialization



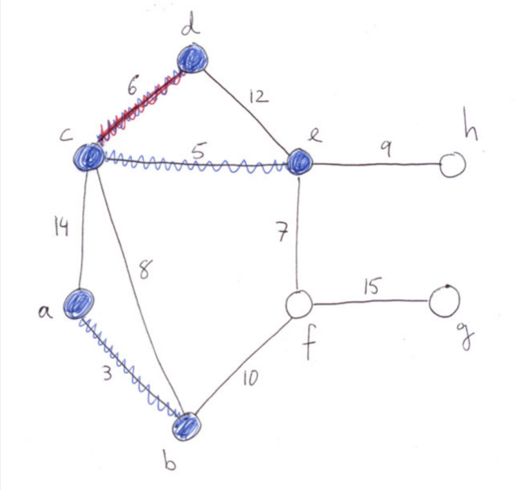
Iteration 1



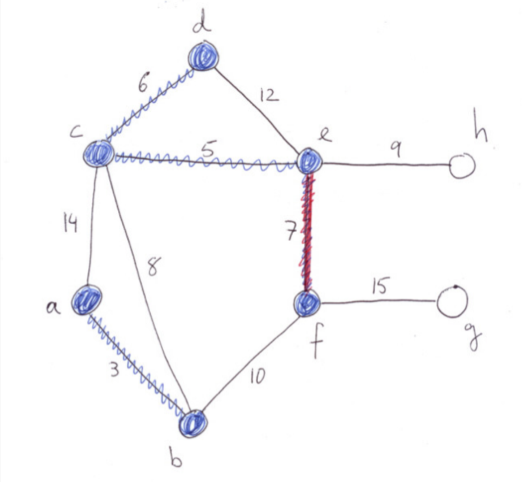
Iteration 2



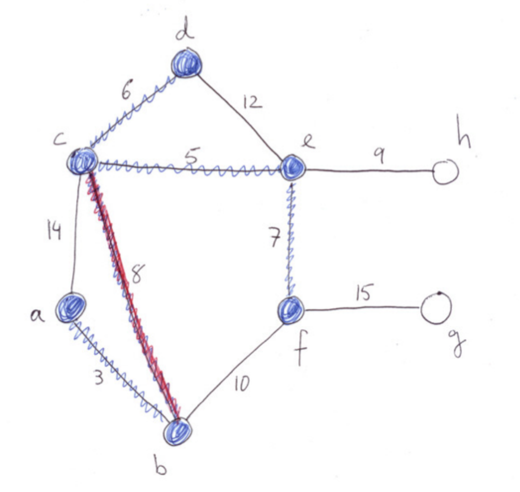
Iteration 3



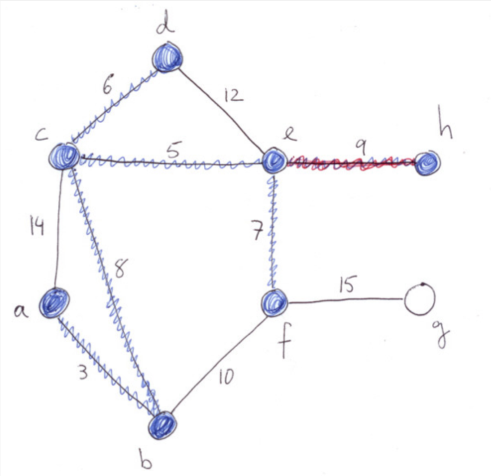
Iteration 4



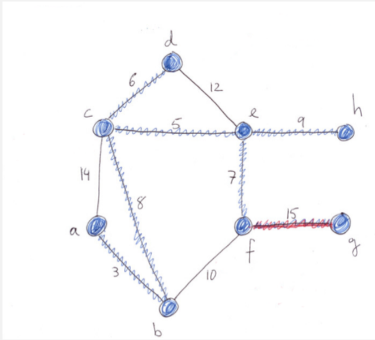
Iteration 5



Iteration 6

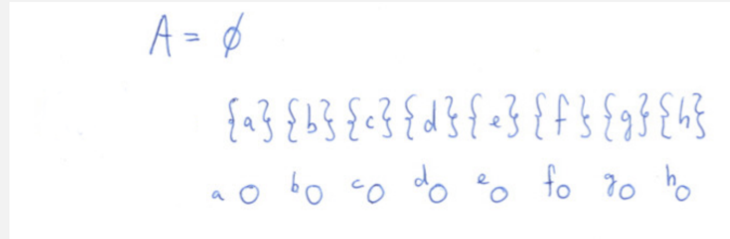


Iteration 7, 8, 9, 10

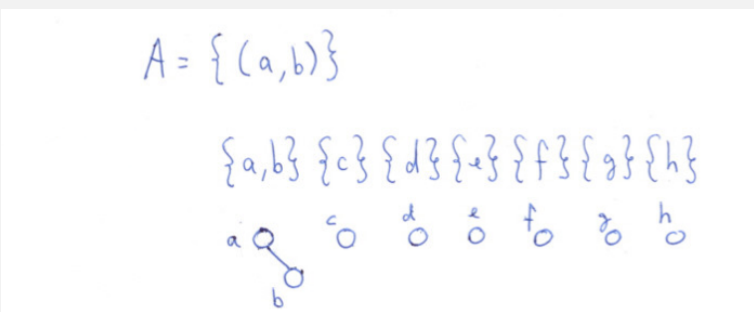


Forest evolution

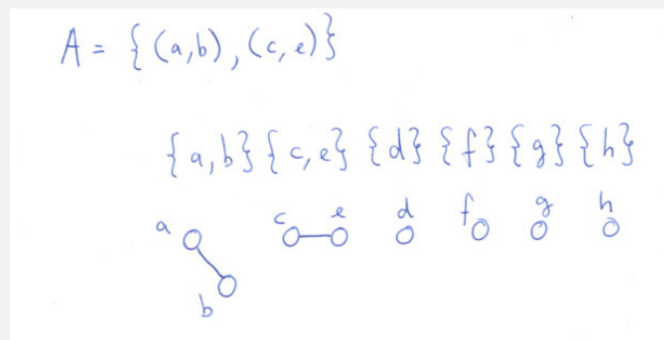
Initialization



Iteration 1



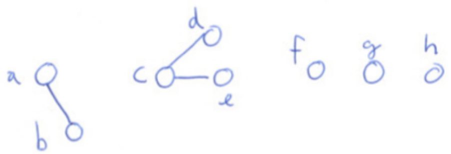
Iteration 2



Iteration 3

$A = \{ (a,b), (c,e), (c,d) \}$

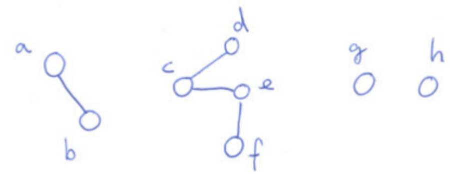
$\{a,b\}\ \{c,d,e\}\ \{f\}\ \{g\}\ \{h\}$



Iteration 4

$A = \{ (a,b), (c,e), (c,d), (e,f) \}$

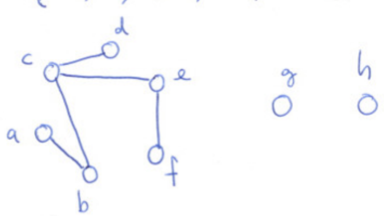
$\{a,b\}\ \{c,d,e,f\}\ \{g\}\ \{h\}$



Iteration 5

$A = \{ (a,b), (c,e), (c,d), (e,f), (c,b) \}$

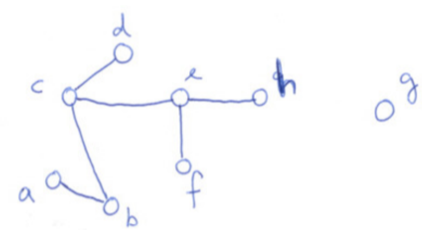
$\{a,b,c,d,e,f\}\ \{g\}\ \{h\}$



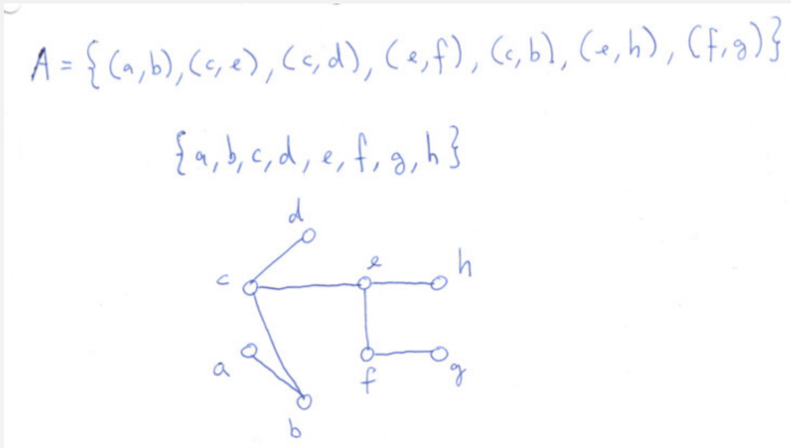
Iteration 6

$A = \{ (a,b), (c,e), (c,d), (e,f), (c,b), (e,h) \}$

$\{a,b,c,d,e,f,h\}\ \{g\}$



Iteration 7, 8, 9, 10



Implementation of Kruskal's algorithm

- Need a data structure that allow us to dynamically keep a set of disjoint trees (the forest).
- Initially we have $|V|$ trees.
- At each iteration we join two trees and we are left with one less tree in the forest.
- In reality, there's no need to keep the trees explicitly.
  - Only need to keep the nodes that each tree has.
  - Note: The trees are disjoint. A node belong to one and only one tree.

Implementation of Kruskal's algorithm

- All we need is a UNION-FIND data structure that we studied a few lectures ago, supporting the operations $\textsc{Make-Set}(x)$, $\textsc{Find-Set}(x)$ and $\textsc{Union}(x, y)$

Pseudocode

$\textsc{MST-Kruskal}(G, w)$

$A = \emptyset$
for each $v \in G.V$
    $\textsc{Make-Set}(v)$
sort $G.E$ in ascending order of weight w
for each $(u, v) \in G.E$, taken in ascending order of weight
    if $\textsc{Find-Set}(u) \neq \textsc{Find-Set}(v)$
        $A = A \cup \{(u, v)\}$
        $\textsc{Union}(u, v)$
return A

Running time of Kruskal's algorithm

- First **for** loop: $O(V)$ MAKE-SETS
- Sort E: $O(E \lg E)$
- Second **for** loop: $O(E)$ FIND-SETS and UNIONS
  In reality we only do $O(V)$ UNIONS. Why?
- Running time depends on the implementation of UNION-FIND.

Running time of Kruskal's algorithm

UNION-FIND implementation with union by rank and path compression:

- First **for** loop: $O(V)$
- Second **for** loop: $O(E \lg^* V)$
  (for all practical purposes, $\lg^* V \leq 5$)
- Total running time is dominated by the time needed to sort the edges: $O(E \lg E) = O(E \lg V)$. Why? Because $|E| \leq |V|^2$

$$\begin{aligned} \Longrightarrow \lg |E| &\leq \lg |V|^2 \\ &= 2 \lg |V| \\ &= O(\lg |V|) \end{aligned}$$