ECE250:
Algorithms and Data Structures

Minimum Spanning Tree (MST) and Greedy Algorithms

Ladan Tahvildari, PEng, SMIEEE

Associate Professor

Software Technologies Applied Research (STAR) Group

Dept. of Elect. & Comp. Eng.

University of Waterloo

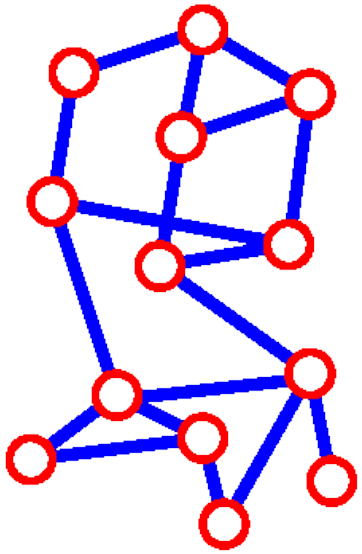
Materials from CLRS: Chapter 23, 16.1, 16.2

Spanning Tree

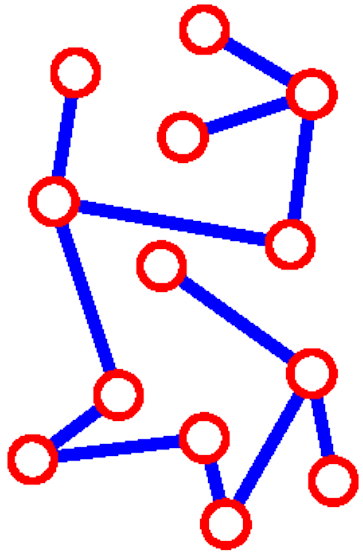
- A **spanning tree** of **G** is a subgraph which
  - is a tree
  - contains all vertices of **G**

- How many edges are there in a spanning tree, if there are *V* vertices?

- An Application
  - Electronic Circuit Design



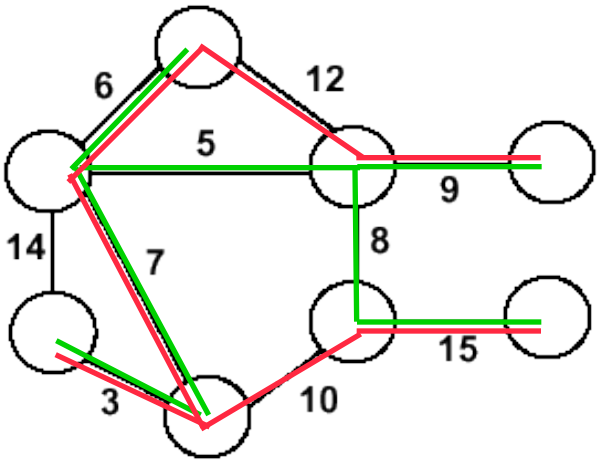
G



spanning tree of G

Minimum Spanning Trees

- Undirected, connected graph $G = (V,E)$
- **Weight** function W: $E \rightarrow R$ (assigning cost or length or other values to edges)



- Spanning tree: tree that connects all the vertices
- Optimization problem – **Minimum spanning tree** (MST): tree T that connects all the vertices and minimizes

$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

Prim-Jarnik Algorithm

- Vertex based algorithm
- Grows one tree T, **one vertex at a time**
- A set *A* covering the portion of T already computed
- Label the vertices *v* outside of the set *A* with *v*.**key**() – the minimum weigth of an edge connecting *v* to a vertex in *A*, *v*.**key**() = ∞, if no such edge exists

Prim-Jarnik Algorithm (cont')

```
MST-Prim(G,r)
01 for each vertex u ∈ G.V()
02     u.setkey(∞)
03     u.setparent(NIL)
04 r.setkey(0)
05 Q.init(G.V())    // Q is a priority queue ADT
06 while not Q.isEmpty()
07     u ← Q.extractMin()    // making u part of T
08     for each v ∈ u.adjacent() do
09         if v ∈ Q and G.w(u,v) < v.key() then
10             v.setkey(G.w(u,v))
11             Q.modifyKey(v)
12             v.setparent(u)
```

updating keys

Prim-Jarnik's Running Time

- Time = $|V|T(\text{extractMin}) + O(E)T(\text{modifyKey})$
- Binary heap implementation:
  - Time = $O(V \lg V + E \lg V) = O(E \lg V)$

Q	T(extractMin)	T(modifyKey)	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$

Kruskal's Algorithm

- Edge based algorithm

- Add the edges one at a time, in increasing weight order

- The algorithm maintains *A* – a **forest of trees**. An edge is accepted it if connects vertices of distinct trees (the cut respects A)

Disjoint-Set ADT

- We need a *Disjoint-Set* ADT that maintains a partition, i.e., a collection S of disjoint sets. Operations:
  - **makeSet**(x:*Vertex*): *SetID*
    - $S \leftarrow S \cup \{\{x\}\}$
  - **union**(x:*Vertex*, y:*Vertex*) –
    - $X \leftarrow S$.**findSet**(x)
    - $Y \leftarrow S$.**findSet**(y)
    - $S \leftarrow S - \{X, Y\} \cup \{X \cup Y\}$
  - **findSet**(x:*Vertex*): *SetID*
    - returns unique $X \in S$, where $x \in X$

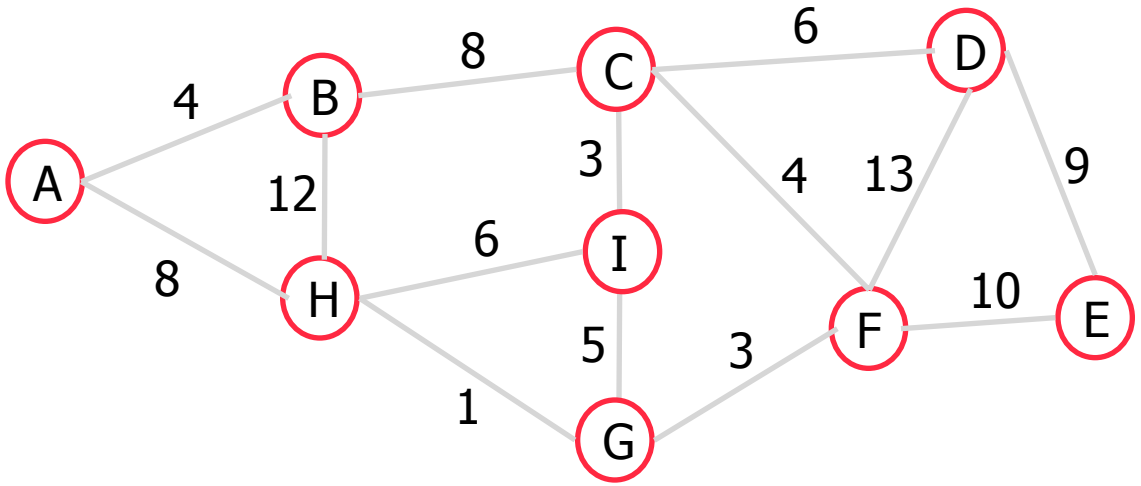
Kruskal's Algorithm

- The algorithm keeps adding the cheapest edge that connects two trees of the forest

```
MST-Kruskal(G)
01 A ← ∅
02 S.init()     // Init disjoint-set ADT
03 for each vertex v ∈ G.V()
04    S.makeSet(v)
05 sort the edges of G.E() by non-decreasing G.w(u,v)
06 for each edge (u,v) ∈ E, in sorted order do
07   if S.findSet(u) ≠ S.findSet(v) then
08      A ← A ∪ {(u,v)}
09      S.union(u,v)
10 return A
```

Kruskal's Example

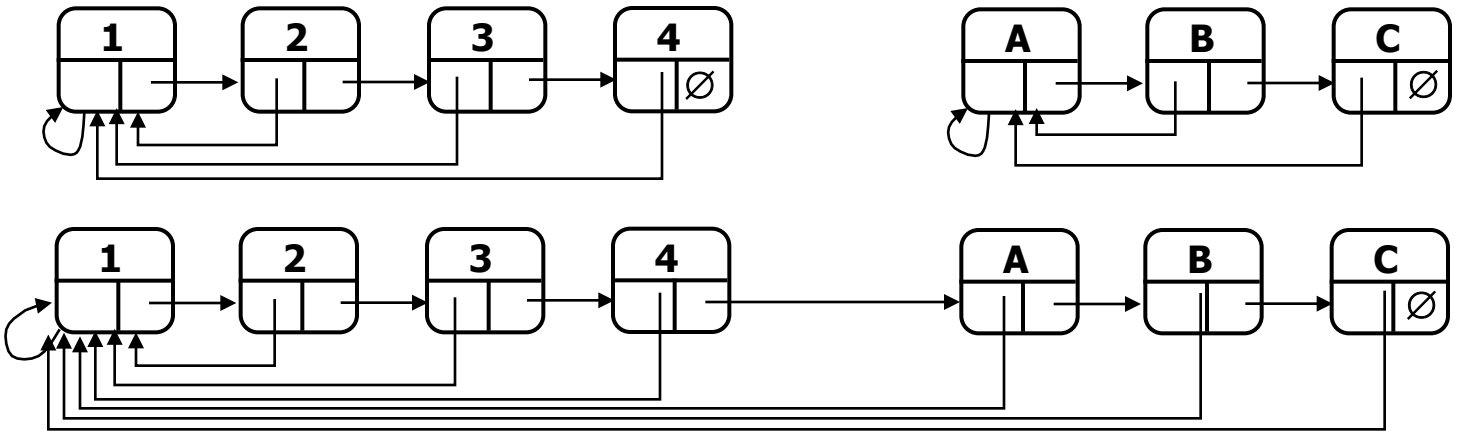
- Let's do an example:



- Let's keep track:
  - What is set *A*, what is a collection *S* , what cuts are we making

Disjoint Sets as Lists

- Each set – a list of elements identified by the first element, all elements in the list point to the first element
- Union – add a smaller list to a larger one
- FindSet: $O(1)$, Union(u,v): $O(\min\{|C(u)|, |C(v)|\})$



Kruskal Running Time

- Initialization $O(V)$ time
- Sorting the edges $\Theta(E \lg E) = \Theta(E \lg V)$
- $O(E)$ calls to FindSet
- Union costs
  - Let $t(v)$ be the number of times v is moved to a new cluster
  - Each time a vertex is moved to a new cluster the size of the cluster containing the vertex at least doubles: $t(v) \leq \log V$
  - Total time spent doing Union

$$\sum_{v \in V} t(v) \leq |V| \log |V|$$

- Total time: $O(E \lg V)$