

Minimum Spanning Trees

Given a connected, weighted, undirected graph $G = (V, E)$, in which each edge $(u, v) \in E$ has a weight $w(u, v)$ associated with it.

The **Minimum Spanning Tree (MST) problem**:

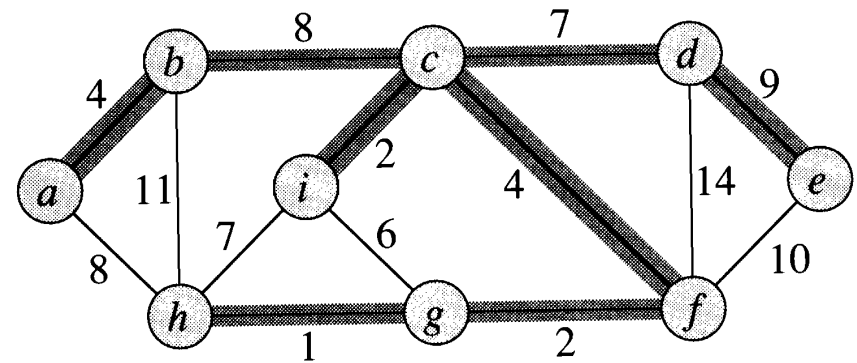
Find a **spanning tree** $T = (V, E')$ such that the sum of the weights of the edges in E' is minimised.

i.e., *minimise*

$$w(T) = \sum_{(u,v) \in E'} w(u,v).$$

Clearly $|E'| = |V| - 1$.

Minimum Spanning Tree Example



Approaches for Finding MSTs

- A “generic” algorithm
- Kruskal’s algorithm
- Prim’s algorithm

“Growing” a Minimum Spanning Tree

A “generic” algorithm grows the minimum spanning tree, one edge at a time, using the **greedy strategy**.

The algorithm maintains a subset $A \subseteq E$ which is a subset of the edges of *some* minimum spanning tree.

At each step, an edge $(u, v) \in E - A$ is added to A .

Generic MST Algorithm

Generic_MST(G, w)

```
1   $A \leftarrow \emptyset$ ;  
2  while the edges in  $A$  do not form a spanning tree do  
3      find an edge  $(u, v) \in E - A$  that is “safe” for  $A$   
4       $A \leftarrow A \cup \{(u, v)\}$ ;  
5  return  $A$ 
```

Generic MST Algorithm

Some important notations.

- cut $(S, V - S)$
- respect
- edge crossing
- light edge

An edge e is “safe” to A (a subset of the edges of *some* MST) if its addition to A maintains the invariant (that A is a subset of the edges of *some* MST).

See Fig. 23.2, (page 564).

Generic MST Algorithm

Theorem

- Let $G(V, E)$ be a connected, weighted, undirected graph with a non-negative, real-valued weight function w defined on E .
- Let A be a subset of E that is included in some MST for G .
- Let $(S, V - S)$ be any cut of G that respects A .
- Let (u, v) be a light edge crossing $(S, V - S)$.

Then, (u, v) is safe for A .

Fig. 23.3 (page 565).

Kruskal's Algorithm

The basic idea of Kruskal's algorithm is as follows.

- The set A is a forest.
- The safe edge added to A is always a minimum weight edge (light edge) in the graph that connects two distinct components.

In other words, let (u, v) be an edge of least weight that connects two trees in the forest and let C_1 and C_2 denote two subtrees that are connected by (u, v) , then add (u, v) to A .

The above procedure continues until all the vertices in V are included in a single tree.

Kruskal's Algorithm

Kruskal_MST(G, w)

```
1   $A \leftarrow \emptyset$ 
2  for each vertex  $v \in V$  do
3      Make_Set( $v$ )
4  sort the edges of  $E$  into nondecreasing order by weight  $w$ 
5  for each edge  $(u, v) \in E$  taken in nondecreasing order by weight do
6      if Find_Set( $u$ )  $\neq$  Find_Set( $v$ )
7          then  $A \leftarrow A \cup \{(u, v)\}$ 
8              Union( $u, v$ )
9  return  $A$ 
```

See Fig 23.4,(pages 568–569).

Kruskal's Algorithm

The running time of Kruskal's algorithm:

- Initialisation takes $O(1)$ time
- Sorting takes $O(m \log m)$ time ($m = |E|$)
- Set operations take $O(m \log m)$ time

The running time of Kruskal's algorithm is $O(m \log m) = O(m \log n)$.

Prim's Algorithm

Prim's algorithm has the property that the edges in A always form a single tree.

The tree starts at an arbitrary root vertex r and grows until the tree spans all the vertices in V .

At edge step, a light edge is added to A that connects A to an isolated vertex of $G_A = (V, A)$.

Prim's Algorithm

Prim_MST(G, w, r)

```
1  for each  $u \in V(G)$ 
2      do  $key[u] \leftarrow \infty$ 
3           $\pi[u] \leftarrow \text{NIL}$ 
4   $key[r] \leftarrow 0$ ;
5   $Q \leftarrow V(G)$  /* Priority Queue (a min-heap) */
6  while  $Q \neq \emptyset$ 
7      do  $u \leftarrow \text{EXTRACT\_MIN}(Q)$ 
8          for each  $v \in \text{Adj}[u]$ 
9              do if  $v \in Q$  and  $w(u, v) < key[v]$ 
10                 then  $\pi[v] \leftarrow u$ 
11                      $key[v] \leftarrow w(u, v)$ 
```

Prim's Algorithm

The performance of Prim's algorithm depends on how we implement the priority queue Q .

- ▶ If we use a min-heap as the priority queue,
 - ▶ Steps 1-5 take $O(n)$ time;
 - ▶ The total time for all calls to `EXTRACT_MIN` is $O(n \log n)$;
 - ▶ The **for** loop is executed $O(E)$ times, each requiring $O(\log n)$ time;
- ▶ Thus, the total running time is $O(n \log n + m \log n) = O(m \log n)$.

See Fig. 23.5 (page 571).