

Analysis of Algorithms, I

CSOR W4231.002

Eleni Drinea
Computer Science Department

Columbia University

Minimum spanning trees: Prim's and Kruskal's algorithms

- 1 Minimum Spanning Trees (MSTs)
 - Prim's algorithm
 - Kruskal's algorithm
 - More MST algorithms

- 1 Minimum Spanning Trees (MSTs)
 - Prim's algorithm
 - Kruskal's algorithm
 - More MST algorithms

The problem

Motivation: build the cheapest communication network over a set of locations.

Input: a weighted, undirected graph $G = (V, E, w)$

Output: a subset of edges $E_T \subseteq E$ such that

1. the graph $T = (V, E_T)$ is connected;
2. $\sum_{e \in E_T} w(e)$ is minimal.

Minimum weight Spanning Trees (MST)

Remark 1.

The graph $T = (V, E_T)$ is a tree: if there is a cycle, remove any edge from the cycle and obtain a connected graph with less cost.

Definition 1 (Spanning tree of a graph $G = (V, E)$).

A tree that spans all the nodes in V .

Output (restated): a **minimum weight** spanning tree of G .

Remarks

- ▶ Brute-force won't work: even simple graphs have many spanning trees—*how many in a simple cycle?*
- ▶ #spanning trees in the **complete** graph on n vertices: n^{n-2}

The cut property

Definition 2 (Cut).

A cut $(S, V - S)$ is a bipartition of the vertices.

Claim 1 (Cut property).

Assume all edge weights are distinct. Let $S \subset V$ ($S \neq \emptyset$). Let e be the minimum-weight edge with one endpoint in S and the other in $V - S$. Then every MST contains e .

Remark 2.

The assumption of distinct edge weights is just for the purposes of the analysis; we will show how to remove it later.

Proof of the cut property

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

We will derive a contradiction by using an **exchange** argument.

- ▶ Let T' be a minimum-weight spanning tree that does not contain $e = (u, v)$.
- ▶ Then there must be some other path P in T' from u to v .
- ▶ Starting at u , follow the vertices of P : since (u, v) crosses from S to $V - S$, there must be some first vertex $v' \in V - S$ on P . Let u' be the last vertex before it in S .
- ▶ Then $e' = (u', v') \in E_{T'}$ **and** e' crosses between $S, V - S$.

Proof of the cut property (cont'd)

Exchange e with e' to obtain the set of edges

$$E_T = E_{T'} + \{e\} - \{e'\}.$$

T is a spanning tree:

- ▶ T is connected: any path in T' that used $e' = (u', v')$ is rerouted to follow P from u' to u , (u, v) and P from v to v' .
- ▶ T is acyclic (*why?*).

Since both e' and e cross between S and $V - S$ but e is the lightest edge with this property, $w(e) < w(e')$. Thus

$$w(T) < w(T').$$

Using the cut property to design MST algorithms

The cut property says: construct MST **greedily** by taking the **lightest** edge across two regions not yet connected.

Generic-MST($G = (V, E, w)$)

$E_T = \emptyset$ // the set of edges that will form our MST

while $|E_T| \leq n - 1$ **do**

Pick $S \subseteq V$ s.t. no edge in E_T crosses between $S, V - S$

 Let $e \in E$ be a lightest edge that crosses between $S, V - S$

$E_T = E_T \cup \{e\}$

end while

Prim's algorithm

In Prim's algorithm, the edges in E_T always form a subtree and S is chosen to be the set of this subtree's vertices.

In other words:

1. Start with a root node s .
2. **Greedily** grow a tree outward from s by adding the node that can be attached as cheaply as possible at every step.

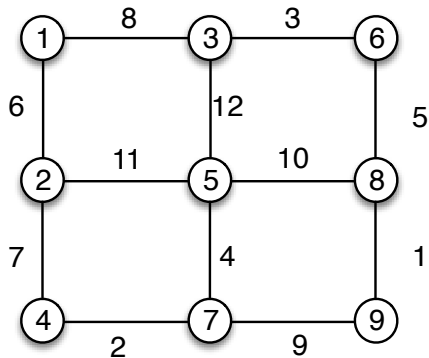
Detailed description of Prim's algorithm

1. $E_t = \emptyset$
2. Maintain a set $S \subseteq V$ on which a spanning tree has been constructed so far. Initially, $S = \{s\}$.
3. In each iteration, update
 - 3.1 $S = S \cup \{v\}$, where v is the vertex in $V - S$ that minimizes the attachment cost:

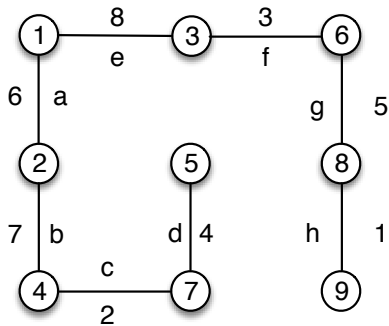
$$\min_{e=(u,v):u \in S} w(e).$$

- 3.2 $E_T = E_T \cup \{e\}$

Example graph



Prim's MST for example graph (letters indicate the order in which edges were added)



Follows directly from the Cut property: at every iteration an edge (u, v) is added such that

- ▶ $u \in S, v \in V - S$ (recall that Prim's algorithm sets S to be the set of vertices on which a partial MST has been constructed);
- ▶ (u, v) is the lightest edge that crosses between S and $V - S$.

Implementing Prim's algorithm

Similarly to Dijkstra's algorithm,

- ▶ store every node $v \in V - S$ in a priority queue Q , e.g., implemented as a binary min-heap (key= weight of the lightest edge between some node in S and v). Initially, $S = \{s\}$.
- ▶ maintain two arrays
 - ▶ $dist[v]$: stores the weight of the lightest edge between v and any vertex in S (in Dijkstra, it stored a conservative overestimate of the distance of v from the source s)
 - ▶ $prev[v]$: stores the node responsible for adding v to S

Pseudocode: how does this compute $T = (V, E_T)$?

```
Prim( $G = (V, E, w), s$ )  
  for  $u \in V$  do  
     $dist[v] = \infty; prev[v] = NIL$   
  end for  
   $dist[s] = 0$   
   $Q = \{V; dist\}$   
   $S = \emptyset$   
  while  $Q \neq \emptyset$  do  
     $u = \text{ExtractMin}(Q)$   
     $S = S \cup \{u\}$   
    for  $(u, v) \in E$  and  $v \in V - S$  do  
      if  $dist[v] > w(u, v)$  then  
         $dist[v] = w(u, v)$   
         $prev[v] = u$   
        DecreaseKey( $Q, v$ )  
      end if  
    end for  
  end while
```


Further implementations of Prim's algorithm

Notation: $|V| = n$, $|E| = m$

Implementation	ExtractMin	Insert/ DecreaseKey	Time
Array	$O(n)$	$O(1)$	$O(n^2)$
Binary heap	$O(\log n)$	$O(\log n)$	$O((n + m) \log n)$
d -ary heap	$O(\log n)$	$O(\log n)$	$O((nd + m) \frac{\log n}{\log d})$
Fibonacci heap	$O(\log n)$	$O(1)$ amortized	$O(n \log n + m)$

- ▶ Optimal choice for $d \approx m/n$ (the *average* degree of the graph)
- ▶ d -ary heap works well for both sparse and dense graphs
 - ▶ If $m = n^{1+x}$, what is the running time of Prim's algorithm using a d -ary heap?
- ▶ **Amortized** analysis: coming up in the next lecture

Kruskal's algorithm

Short description: at every step, add to E_T the *lightest* edge that does not create a cycle with the edges already in E_T .

Thus, at all times, E_T is a subset of an MST.

Alternative view: merging partial trees

Initially, every vertex forms its own trivial tree (no edges).
Maintain a *forest* of trees at all times.

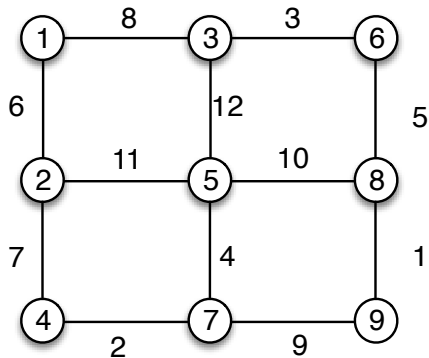
Let $T(v)$ be the tree where vertex v belongs.

1. Initialize $E_T = \emptyset$
2. **Sort** the edges by increasing weight.
3. For every edge $e = (u, v)$ in **increasing order** of weight:
 - ▶ If u and v belong to the same tree, discard e .
 - ▶ Else
 - ▶ $E_T = E_T \cup \{e\}$;
 - ▶ **merge** $T(u)$, $T(v)$ into a single tree.

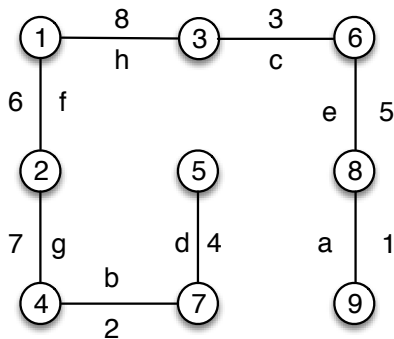
\triangle *Need a data structure that allows*

1. *to check if u, v belong to the same tree;*
2. *for updates to reflect the merging of two trees into one.*

Example graph



Kruskal's MST for example graph (letters indicate the order in which edges were added)



- ▶ Let (u, v) be the edge added at the current iteration.
 - ▶ Let S be the set of nodes that have a path to u by edges in A just before (u, v) is added; then $u \in S$ but $v \notin S$.
 - ▶ Also, (u, v) must be the **first** edge between S and $V - S$ encountered so far: otherwise, if such an edge was encountered before, it would have been added to A since its inclusion would not cause a cycle.
- ⇒ (u, v) is the **lightest** edge that crosses between S and $V - S$
- ▶ By the Cut Property, (u, v) belongs to the MST.

Implementing Kruskal's algorithm

Kruskal's algorithm maintains a forest of trees at all times, starting from n trivial trees (no edges).

Want a data structure that maintains a **collection of disjoint sets** and supports operations:

1. **MakeSet**(u): Given an element u , create a new tree containing only u . **Target worst-case time: $O(1)$**
2. **Find**(u): Given an element u , find which tree u belongs to. **Target worst-case time: $O(\log n)$**
3. **Union**(u, v): Merge the tree containing u and the tree containing v into a single tree. **Target worst-case time: $O(\log n)$**

Pseudocode

```
Kruskal( $G = (V, E, w)$ )  
   $E_T = \emptyset$   
  Sort( $E$ ) by  $w$   
  for  $u \in V$  do MakeSet( $u$ )  
  end for  
  for  $(u, v) \in E$  by increasing  $w$  do  
    if Find( $u$ )  $\neq$  Find( $v$ ) then  
       $E_T = E_T \cup \{(u, v)\}$   
      Union( $u, v$ )  
    end if  
  end for
```


Running time analysis

- ▶ Sorting: $O(m \log m) = O(m \log n)$
- ▶ n `Makeset()` operations: $O(n)$
- ▶ $2m$ `Find()` operations: $2m \cdot O(\log n)$
- ▶ $\leq n - 1$ `Union()` operations: $n \cdot O(\log n)$

Running time: $O(m \log n)$

When is it safe to not include an edge to the MST?

Fact 3 (The Cycle Property).

Assume that all edge costs are distinct. Let C be any cycle in T , and let edge (u, v) be the heaviest edge in C . Then e does not belong to any MST of G .

Proof of the cycle property

- ▶ Let T be a spanning tree that contains e . We want to show that T is not optimal.
 - ▶ To this end, we will exchange e for some e' to get a spanning tree T' with less weight.
 - ▶ First, delete e from T ; T is now partitioned into two components: the set S containing u and the set $V - S$ containing v .
- ⇒ We want an edge e' with one endpoint in S and another in $V - S$ so as to reconnect them.

Proof of the cycle property (cont'd)

- ▶ We can find such an edge by following the cycle C .
- ▶ Consider the edges of C except for e : they form a path from u to v .
- ▶ So if we start at u , following this path, at some point there is an edge e' that crosses from S to $V - S$. Construct

$$E_{T'} = E_T - \{e\} + \{e'\}.$$

- ▶ Now T' is connected and has $n - 1$ edges. Moreover, since e is the heaviest edge in the cycle

$$w(T') < w(T).$$

More MST algorithms

Fact 3 yields yet another algorithm for finding an MST.

Reverse-Delete($G = (V, E, w)$)

- ▶ Start with the full graph.
- ▶ Sort the edges in decreasing weight.
- ▶ Repeatedly delete edges in order of decreasing weight, so long as the graph does not become disconnected.

More MST algorithms: combine the Cut property (to add edges) and the Cycle property (to eliminate edges).

△ Such algorithms may be subtle to implement.

Removing the assumption of unequal edge weights

- ▶ Suppose some edges have equal weights.
 - ▶ Slightly perturb all edge weights by different, **tiny** amounts.
- ⇒ All edge weights are now distinct.
- ▶ Apply the algorithms discussed in the previous sections.

Remark 3.

*Perturbations serve as **tie-breakers**: edges whose weights differed before still have the same relative order.*