Graph Applications: Minimum-cost Spanning Trees

Lecture 28
by Marina Barsky
Recall: Connectivity in undirected graphs

- Two vertices are connected, if there is a path between them.

$v_1$ and $v_6$ are connected.
Recall: Connected graph

- A graph is **connected**, if any two of its nodes are connected. In other words, there is a path between any pair of nodes.

This graph is connected.

This graph is not connected.
How to find if the graph is connected

1. How to find out whether an undirected Graph is connected?

- Hint: traversals
- What is the running time of these algorithms?
Minimum spanning trees: Motivation

- Connect all the computers in a new office building using the least amount of cable

- Road repair: repair only min-cost roads such that all the cities are still connected

- Airline: downsize operations but preserve connectivity
Definitions

- A **Spanning Tree** of a graph $G$, is a subgraph of $G$ which is a tree and contains all vertices of $G$.

- A **Minimum Spanning Tree (MST)** of a weighted graph $G$ is a spanning tree with the smallest total weight.
Problem: compute MST of Graph G

Input: undirected graph $G=(V, E)$ and the weight $w_e$ for each edge

Output: minimum-cost tree $T \in E$ that spans all the vertices $V$

Simplifying assumptions:
• $G$ is undirected and simple (that is, it has no self-loops and no parallel edges)
• Input graph $G$ is connected

Tree means a subgraph that:

❑ has no cycles
❑ has exactly n-1 edges
❑ is connected
Which of the following subgraphs (in red) are Spanning trees

- A
- B
- C
- More than one of the above
- None of the above
Problem: compute MST of Graph G

**Input:** undirected graph $G=(V, E)$ and the weight $w_e$ for each edge

**Output:** minimum-cost tree $T \in E$ that spans all the vertices $V$

Simplifying assumptions:
- $G$ is undirected and simple (that is, it has no self-loops and no parallel edges)
- Input graph $G$ is connected

Tree means a subgraph that:
- has no cycles
- has exactly $n-1$ edges
- is connected
MST Algorithm by **Prim**

Grows a tree **starting from a single (arbitrarily selected) vertex.**

- Start from an arbitrary vertex
- Span another vertex by choosing the edge with the min cost (greedy move)
- Now have a tree of 2 vertices
- Check all edges out of this tree and choose the one with min-cost ...
Algorithm Prim_MST (graph G(V,E))

initialize tree T: = ∅ # set of tree edges
X: = {vertex s} # s ∈ V, chosen arbitrarily
# X contains vertices spanned by the tree-so-far

while |X|!=|V|:
    let e=(u,v) be the cheapest edge of G with u ∈ X and v ∈ X
    add e to T
    add v to X
# that increases the number of spanned vertices
Which of the following sets of tree edges can be produced by the Prim algorithm?

A. (AB), (BG), (GC), (BC), (GF), (CD), (FE)
B. (GB), (GC), (AB), (GF), (CD), (DE)
C. (ED), (DC), (CG), (GB), (BA), (GF)
D. More than one of the above
E. None of the above
Prim: illustration
Prim: illustration
Prim: illustration
Prim: illustration
Prim: illustration
Prim: illustration
Prim: illustration

MST cost: $3 + 1 + 2 + 3 + 4 + 4 = 17$
Prim’s Algorithm

- Prim always finds a minimum-cost spanning tree for any connected graph (even if the weights are negative)!
- How can we argue that Prim’s algorithm is optimal?
- Why is it always a good idea to take the cheapest edge from the existing tree-so-far?
Cuts

• A *cut* is a partition \((A, B)\) of \(G\) into 2 non-empty subsets (proper subsets)

• How many different cuts can be in a \(G\) with \(n\) vertices? \((n, n^2, 2^n)\)? \(2^n - 2\)
Crossing Edges Lemma

If there are (at least) two crossing edges for a cut \((A,B)\) in an undirected connected graph, then these edges must be a part of some cycle.

Proof

If there is a path from \(u\) to \(v\) from two different partitions that includes the first crossing edge \(e\), then the second crossing edge \(f\) offers an alternative path from \(u\) to \(v\), thus closing the cycle on vertex \(v\).
Cut Crossing Theorem

• Let G be a weighted connected graph, and let (A, B) be some cut of G.

• If $e$ is the cheapest edge crossing cut (A, B), then $e$ must be a part of some MST.
What we are proving

If we have an edge in a graph and you can find just a single cut for which this edge has the min cost among all edges crossing this cut, then this edge **must** belong to the MST (or one of MSTs in case when the weights are not unique)

Note that edge 4 is never min of all crossing edges, no matter how we cut – so edge 4 is not in MST
Exchange argument!

- Any nontree edge must have weight that is ≥ every edge in the cycle created by that edge and a minimum spanning tree.

- Suppose edge e has weight 32 and edge f in the same cycle has weight 33. Edge f is a part of MST (shown with bold edges), and edge e is not.

- But then we could replace f by e and get a spanning tree with lower total weight, which would contradict the fact that we started with a minimum spanning tree.
Prim: cut

Set X of vertices already in MST

Set V-X of remaining vertices
Algorithm Prim_MST (graph G(V,E))

initialize tree T: = ∅ # set of tree edges
X: = {vertex s} # s ∈ V, chosen arbitrarily
# X contains vertices spanned by the tree-so-far
Min-PQ: = ∅ # set of all edges out of X prioritized by cost
current:= vertex s

while |X|!=|V|:
    for each e in neighbors(current):
        Min-PQ.enqueue(e)
    Select e =(u,v) as Min-PQ.dequeue()
    if u ∈ X and v ∉ X: What data structure to use to check this quickly?
        add e to T
        add v to X
    current:=v
Algorithm Prim_MST (graph G(V,E))

initialize tree T: = ∅
X: = {vertex s}
# X contains vertices spanned by the tree-so-far
Min-PQ: = ∅
current:= vertex s

while |X|!=|V|:

    for each e in neighbors(current):

        Min-PQ.enqueue(e)

    Select e =(u,v) as Min-PQ.dequeue()

    if u ∈ X and v ∉ X:

        add e to T
        add v to X
        current:=v

O(n)

No more than O(m) edges in total, O(log m) for dequeue

O(1)

Total running time is \(O(m \log m)\)
Algorithm by Kruskal

Sort all edges by weight (from smaller to larger – ascending)

**Add the next smallest edge** to the spanning tree, but only if adding it does not create a cycle

Sorted edges:
- (a,b) ✓
- (a,c) ✓
- (a,d) ✓
- (b,d) ✗
- (c,d) ✗
Algorithm Kruskal_MST (graph G(V,E))

\[ E_{\text{sorted}} := \text{edges of } G \text{ sorted by weights} \]
\[ T := \emptyset \quad \# \text{ set of spanning tree } edges \]

for i from 1 to m:
    if \( T \cup \{E_{\text{sorted}}[i]\} \) has no cycles
        add \( E_{\text{sorted}}[i] \) to T

return T
Which sequence represents the order in which edges are added to MST by the Kruskal algorithm?

A. (BG), (GC), (BA), (FG), (ED), (FE)
B. (BG), (BC), (GA), (CG), (ED), (FE)
C. (BG), (GC), (GF), (FE), (ED), (CE)
D. More than one of the above
E. None of the above
Kruskal illustration
Kruskal illustration
Kruskal illustration
Kruskal illustration
Kruskal illustration

Note that at this point T is not even a spanning tree (not connected)
Kruskal illustration

MST cost: $1 + 2 + 3 + 3 + 4 + 4 = 17$
Repeatedly add a minimum-cost edge that does not create a cycle.
Kruskal algorithm

Algorithm Kruskal_MST (graph G(V,E))

\[ E' := \text{edges of } G \text{ sorted by weights} \]
\[ T := \emptyset \quad \# \text{collects edges of the future MST} \]

for i from 1 to m:
    if \( T \cup \{E'[i]\} \) has no cycles
        add \( E'[i] \) to \( T \)
    if \( |T| = |V| - 1 \):
        \# we can stop once we have a tree
        break

return \( T \)

Stop when
n-1 edges have been selected
Running time

Kruskal\_MST (graph G(V,E))

1. $E' :=$ edges of $G$ sorted by weights
2. $T := \emptyset$
3. for $i$ from 1 to $m$:
   4. if $T \cup \{E'[i]\}$ has no cycles
      5. add $E'[i]$ to $T$
6. if $|T| = |V| - 1$
   7. break
8. return $T$

Line 1: sorting $m$ edges by weight. $O(m \log m)$. This is the same as $O(m \log n)$ Why?

Line 3: outer for loop. $O(m)$. We check all $m$ edges in the worst case.

Line 4: need to find if edge $E'[i] = (u,v)$ creates a cycle.
Find out if there is already a path from $u$ to $v$ in $T$ by any graph traversal (DFS or BFS). DFS of $T$ with $n$ vertices and $n-1$ edges is $O(n + n^2) = O(n)$.

Thus, total time of the for loop is $O(m) \times O(n) = O(mn)$

Kruskal MST runs in time $O(m \log n) + O(mn) = O(mn)$
### Running time

**Kruskal_MST (graph G(V,E))**

1. \( E' := \) edges of \( G \) sorted by weights
2. \( T := \emptyset \)
3. **for** \( i \) **from** 1 **to** \( m \):
   - **if** \( T \cup \{E'[i]\} \) has no cycles
     - add \( E'[i] \) to \( T \)
4. **if** \( |T| = |V| - 1 \):
   - **break**
5. **return** \( T \)

**Can we do better?**

Kruskal MST runs in time \( O(mn) \)
Kruskal as union of sets

We can look at Kruskal from a Set point of view

• First we have n sets: each vertex i is in its own set $S_i$ – we need to be able to MAKE-SET for a single element

• Next we combine two sets of vertices $S_i$ and $S_j$ into one set: we perform UNION ($S_i$ and $S_j$), adding an edge $(u,v)$ such that $u \in S_i$ and $v \in S_j$

• However we do the union only if $S_i \neq S_j$. In other words, we need to know if u and v are already in the same set, in the same connected component, we need to FIND out set names for u and for v and compare them for equality

Note that all the sets are disjoint: each node belongs to a single set during the execution of the algorithm
Kruskal as union of sets

Set B = UNION (B, G)
Kruskal as union of sets

Set B = UNION (B, C)
Kruskal as union of sets

Set B = UNION (B, A)
Kruskal as union of sets

Set $B = \text{UNION} (B, F)$
Kruskal as union of sets

Set E = UNION (D, E)
Kruskal as union of sets

Set $E = \text{UNION} \ (B, E)$
Kruskal as union of sets

Set spanning all vertices of G with selected edges:
MST of G
New ADT: **UNION-FIND**  
(= Disjoint Set ADT)

**UNION-FIND** is an Abstract Data Type that supports the following operations:

- **MAKESET**(x): Creates a new set X containing a single element x.
- **UNION**(X, Y): Creates a new set containing the elements of sets X and Y in their union and deletes the previous sets X and Y.
- **FIND**(x): Returns the name of the set to which element x belongs.

Read about an efficient implementation of UNION-FIND in more detailed slides or in this textbook chapter.
Kruskal running time with UNION-FIND

Kruskal_MST (graph G(V,E))

1. $E' :=$ edges of G sorted by weights
2. $T := \emptyset$
3. for $i$ from 1 to $n$:
   MAKE-SET (node $i$)
4. for each edge $(u,v)$ in $E'$:
   if $\text{FIND}(u) \neq \text{FIND}(v)$:
     $T := T \cup (u,v)$
   UNION($u$, $v$)
5. if $|T| = |V| - 1$:
     break
6. return $T$

Line 1: sorting $m$ edges by weight. $O(m \log n)$.

Line 3: Making an array of size $n$: $O(n)$.

Line 5: $O(m)$ edges in the worst case.
For each edge: perform FIND: $O(\log n)$ and
sometimes UNION: in time $O(1)$

Thus, total time of the for loop is $O(m \log n)$

Kruskal MST with UNION-FIND runs in time $O(m \log n) + O(n) + O(m \log n) = O(m \log n)$
Both MST algorithms are greedy

All the algorithms follow some greedy strategy.

**Algorithm MST (graph G(V,E))**

<table>
<thead>
<tr>
<th>T: = ∅</th>
<th># collects edges of the future MST</th>
</tr>
</thead>
<tbody>
<tr>
<td>while</td>
<td></td>
</tr>
<tr>
<td></td>
<td>while</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

return T