Lecture 16:
• Minimum Spanning Trees

Nov 2, 2018
Minimum Spanning Trees
Network Design

• **Build a cheap, well connected network** (= graph)
• We are given
  • a set of nodes \( V = \{v_1, \ldots, v_n\} \)
  • a set of possible edges \( E \subseteq V \times V \)
• Want to build a network to connect these locations
  • Every \( v_i, v_j \) must be connected
  • Must be as **cheap** as possible

• Many variants of network design
  • Recall the bus routes problem from HW2
Minimum Spanning Trees (MST)

• **Input**: a weighted graph  \( G = (V, E, \{w_e\}) \)
  - Undirected, connected, weights may be negative
  - All edge weights are distinct (makes life simpler)

• **Output**: a spanning tree \( T \) of minimum cost
  - A spanning tree of \( G \) is a subset of \( T \subseteq E \) of the edges such that \( (V, T) \) forms a tree (connected, acyclic)
  - Cost of a spanning tree \( T \) is the sum of the edge weights
    \[
    \text{cost}(T) = \sum_{e \in T} w(e)
    \]

\[\text{MST}: \quad T^* \in \arg\min_{\text{trees } T} \text{cost}(T)\]
Minimum Spanning Trees (MST)
Minimum Spanning Trees (MST)
MST Algorithms

• There are at least four reasonable MST algorithms
  
  • **Borůvka’s Algorithm**: start with $T = \emptyset$, in each round add cheapest edge out of each connected component
  
  • **Prim’s Algorithm**: start with some $s$, at each step add cheapest edge that grows the connected component
  
  • **Kruskal’s Algorithm**: start with $T = \emptyset$, consider edges in ascending order, adding edges unless they create a cycle
  
  • **Reverse-Kruskal**: start with $T = E$, consider edges in descending order, deleting edges unless it disconnects
Cycles and Cuts

- **Cycle**: a set of edges $(v_1, v_2), (v_2, v_3), ..., (v_k, v_1)$

  ![Cycle Example]
  
  Cycle $C = (1,2),(2,3),(3,4),(4,5),(5,6),(6,1)$

- **Cut**: a subset of nodes $S$

  ![Cut Example]
  
  Cutset $S = \{4, 5, 8\}$
  "Edges cut by $S$"
Cycles and Cuts

• **Fact:** a cycle and a cutset intersect in an even number of edges

\[
| \text{Cut} \cap C | = 0
\]

\[
| \text{Cut} \cap C | = 2
\]

\[
| \text{Cut} \cap C | = 4
\]

"Every time I leave S, I must come back."
Properties of MSTs

• **Cut Property:** Let $S$ be a cut. Let $e$ be the minimum weight edge cut by $S$. Then the MST $T^*$ contains $e$
  • We call such an $e$ a **safe edge**

• **Cycle Property:** Let $C$ be a cycle. Let $f$ be the maximum weight edge in $C$. Then the MST $T^*$ does not contain $f$.
  • We call such an $f$ a **useless edge**
Proof of Cut Property

- **Cut Property**: Let $S$ be a cut. Let $e$ be the minimum weight edge cut by $S$. Then the MST $T^*$ contains $e$

- **Proof by Contradiction**:
  - Let $T^*$ be an MST, $e \notin T^*$
  - There is some $f \in T^*$ that is also in $\text{Cutset}(S)$
  - $w(f) > w(e)$ because $e$ is a safe edge for cut $S$

\[
\Rightarrow \text{cost}(T^* - \{f\} + \{e\}) < \text{cost}(T^*)
\]
Proof of Cut Property

• **Cut Property**: Let $S$ be a cut. Let $e$ be the minimum weight edge cut by $S$. Then the MST $T^*$ contains $e$

• $T^* - \{f, 3\} + \{e, 3\}$ is a spanning tree
  - $T^* - \{f, 3\}$ has two connected components, $S$ and $S^c$
  - $e$ bridges $S$ and $S^c$

• Then $T^*$ is not an MST, contradiction.
Proof of Cycle Property

- **Cycle Property**: Let $C$ be a cycle. Let $f$ be the max weight edge in $C$. The MST $T^*$ does not contain $f$.

- **Proof by contradiction**:
  - Assume $T^*$ is an MST, $f \notin T^*$
  - $T^* - \{f\}$ has two connected components $S, S^c$
  - $C$ intersects $\text{Cutset}(S)$ in an even number of edges.
    $$\Rightarrow$$ there is an $e \in C$ and $e \in \text{Cutset}$
    $$\Rightarrow$$ $\omega(e) < \omega(f)$
Proof of Cycle Property

- **Cycle Property:** Let $C$ be a cycle. Let $f$ be the max weight edge in $C$. The MST $T^*$ does not contain $e$.

  - $\text{cost}(T^* - \{f\} + \{e\}) < \text{cost}(T^*)$
  - $T^* - \{f\} + \{e\}$ is spanning tree
  - But then $T^*$ is not an MST, contradiction.
Ask the Audience

• Assume $G$ has distinct edge weights

• **True/False?** If $e$ is the edge with the smallest weight, then $e$ is always in the MST $T^*$

• **True/False?** If $e$ is the edge with the largest weight, then $e$ is never in the MST $T^*$

$e$ is the safe edge for $S = \{u\}$
Ask the Audience

• Assume $G$ has distinct edge weights

• **True/False?** If $e$ is the edge with the smallest weight, then $e$ is always in the MST $T^*$

• **True/False?** If $e$ is the edge with the largest weight, then $e$ is never in the MST $T^*$

what if there is only one edge?
The “Only” MST Algorithm

• **GenericMST:**
  • Let $T = \emptyset$
  • Repeat until $T$ is connected:
    • Find one or more safe edges not in $T$
    • Add safe edges to $T$

• **Theorem:** **GenericMST** outputs an MST

  **Proof:**
  1. We only add safe edges
  2. If $T$ not connected, then there exists a safe edge
Suppose $T$ is not connected.

There must be edges between each component or else $E$ is not connected.

$\Rightarrow$ there is some edge in the cut $C_i$.

$\Rightarrow$ there is a safe edge in the cut $C_i$. 

Borůvka’s Algorithm

**Borůvka:**

1. Let $T = \emptyset$
2. Repeat until $T$ is connected:
   1. Let $C_1, ..., C_k$ be the connected components of $(V, T)$
   2. Let $e_1, ..., e_k$ be the safe edge for the cuts $C_1, ..., C_k$
   3. Add $e_1, ..., e_k$ to $T$

**Correctness:** every edge we add is safe
Borůvka’s Algorithm

Initially $T = \emptyset$

Label Connected Components of the graph $(V, T)$
Borůvka’s Algorithm

Add Safe Edges

Diagram:

- Nodes: 1, 2, 3, 4, 5, 6, 7, 8
- Edges:
  - 1 to 2, weight 6
  - 2 to 3, weight 14
  - 2 to 4, weight 3
  - 3 to 4, weight 8
  - 4 to 5, weight 10
  - 5 to 6, weight 15
  - 5 to 7, weight 7
  - 6 to 7, weight 9

The algorithm would start by selecting the minimum weight edge from each node to its nearest neighbor. In this case, it would add the edges 1-2, 2-3, and 2-4 as the initial set of safe edges.
Borůvka’s Algorithm

Label Connected Components
Borůvka’s Algorithm

Add Safe Edges
Borůvka’s Algorithm

Done!
Borůvka’s Algorithm (Running Time)

• **Borůvka**
  • Let $T = \emptyset$
  • Repeat until $T$ is connected:
    • Let $C_1, \ldots, C_k$ be the connected components of $(V, T)$
    • Let $e_1, \ldots, e_k$ be the safe edge for the cuts $C_1, \ldots, C_m$
    • Add $e_1, \ldots, e_k$ to $T$

• **Running time**
  • How long to find safe edges?
  • How many times through the main loop?
Borůvka’s Algorithm (Running Time)

**FindSafeEdges(G,T):**

find connected components \( C_1, ..., C_k \)

let \( L[v] \) be the component of node \( v \)

Let \( S[i] \) be the safe edge of \( C_i \)

for each edge \((u,v)\) in \( E\):

If \( L[u] \neq L[v] \):

If \( w(u,v) < w(S[L[u]]) \):

\( S[L[u]] = (u,v) \)

If \( w(u,v) < w(S[L[v]]) \):

\( S[L[v]] = (u,v) \)

Return \( \{S[1],...,S[k]\} \) (Remove duplicates)

Running Time to find safe edges is \( O(m) \)
Borůvka’s Algorithm (Running Time)

• **Claim:** every iteration of the main loop halves the number of connected components.

• \[ \Rightarrow \text{# of iterations is } O(\log n) \]

• **“Proof”**

  After iteration \( i \), we have components \( C_1, \ldots, C_k \)

  ![Diagram showing the halving of connected components](image)

  Iteration \( i+1 \), each component contains at least two previous components

  \[
  \frac{\text{# comp's after } i+1}{\text{# of comp's after } i} \leq \frac{1}{2}
  \]
Borůvka’s Algorithm (Running Time)

**Borůvka**
- Let $T = \emptyset$
- Repeat until $T$ is connected:
  - Let $C_1, \ldots, C_k$ be the connected components of $(V, T)$
  - Let $e_1, \ldots, e_k$ be the safe edge for the cuts $C_1, \ldots, C_m$
  - Add $e_1, \ldots, e_k$ to $T$

**Running Time:**
- How long to find safe edges? $O(m)$
- How many times through the main loop? $O(\log n)$

Total time: $O(m \log n)$
Prim’s Algorithm

• **Prim Informal**
  • Let $T = \emptyset$
  • Let $s$ be some arbitrary node and $S = \{s\}$
  • Repeat until $S = V$
    • Find the cheapest edge $e = (u, v)$ cut by $S$. Add $e$ to $T$ and add $v$ to $S$

• **Correctness:** every edge we add is safe
Prim’s Algorithm

**Prim(G=(V,E))**

- let Q be a priority queue storing V
- value[v] ← ∞, last[v] ← ⊥
- value[s] ← 0 for some arbitrary s

while (Q ≠ ∅):
  - u ← ExtractMin(Q)
  - for each edge (u,v) in E:
    - if v ∈ Q and w(u,v) < value[v]:
      - DecreaseKey(v,w(u,v))
      - last[v] ← u

T = {(1,last[1]),..., (n,last[n])} (excluding s)

return T
Kruskal’s Algorithm

• **Kruskal’s Informal**
  • Let $T = \emptyset$
  • For each edge $e$ in ascending order of weight:
    • If adding $e$ would decrease the number of connected components add $e$ to $T$

• **Correctness:** every edge we add is safe
Claim: Every edge added by Kruskal is a safe edge.

Proof: Consider some edge $e$, added by Kruskal, when we considered $e$, the $T$ looked like

There are other edges leaving the cut $C_1$, suppose $e$ were not the minimum. If $w(f) < w(e)$ then we already considered $f$. Why didn’t we add $f$? At the time we considered $f$, its endpoints were also in two different components. But then we would have added $f$!

So there is no $f \in \text{Cut}(C_1)$ st. $w(f) < w(e)$.
Kruskal’s Algorithm
Implementing Kruskal’s Algorithm

• **Union-Find**: group items into components so that we can efficiently perform two operations:
  • Find(u): lookup which component contains u
  • Union(u,v): merge connected components of u,v

• Can implement **Union-Find** so that
  • Find takes $O(1)$ time
  • Any $k$ Union operations takes $O(k \log k)$ time

• Naïve Implementation is an array
  • Find takes $O(1)$ time
  • Union can take $O(n)$ time
Kruskal’s Algorithm (Running Time)

• **Kruskal’s Informal**
  - Let $T = \emptyset$
  - For each edge $e$ in ascending order of weight:
    - If adding $e$ would decrease the number of connected components add $e$ to $T$

• Time to sort: $O(m \log m)$
• Time to test edges: $2m$ find operations $\rightarrow O(m)$ time
• Time to add edges: $n-1$ union operations $\rightarrow O(n \log n)$ time

Total time is $O(m \log m)$
Implementing Union Find

1. Maintain an array with the component of each item

   \[ \begin{array}{cccccc}
   1 & 2 & 3 & 4 & 5 & 6 & 7 \\
   1 & 2 & 3 & 4 & 5 & 6 & 7 \\
   \end{array} \]

   \[ \text{\textsc{Union}}(5, 7) \]

   \[ \begin{array}{cccccc}
   1 & 2 & 3 & 4 & 5 & 6 & 7 \\
   1 & 2 & 3 & 4 & 5 & 6 & 5 \\
   \end{array} \]

   \( \text{Find} = O(1) \), \( \text{Union} = O(n) \)

2. For every component, maintain a linked list of the items in that component

   \[ \begin{array}{cccc}
   1 & 1 & \text{T} \\
   2 & 2 & \text{T} \\
   3 & 3 & \text{T} \\
   \end{array} \]

   \[ \text{\textsc{Union}}(1, 3) \]

   \[ \begin{array}{cccc}
   1 & 1 & \text{T} \rightarrow \text{T} & \text{T} \\
   2 & 2 & \text{T} & \text{T} \\
   3 & 3 & \text{T} & \text{T} \\
   \end{array} \]

   \[ \vdots \]

   \( \text{\textsc{Union}}(i, j) \) takes time = to size of component \( j \)
(3) Keep the size of each component, merge the smaller into the bigger.

Claim: $k$ unions takes $O(k \log k)$ time.

Pf:  
1. After $k$ unions only $O(k)$ items have changed component at all.
2. The largest component has size $O(k)$.
3. Every time an item changes component, the size of its component doubles.
   \[\Rightarrow \text{no item changed component more than } O(\log k) \text{ times}\]

\[
\therefore \text{Total changes of component is } O(k \log k)
\]
Comparison

• **Boruvka’s Algorithm:**
  • Only algorithm worth implementing
  • Low overhead, can be easily parallelized
  • Each iteration takes $O(m)$, very few iterations in practice

• **Prim’s/Kruskal’s Algorithms:**
  • Reveal useful structure of MSTs
  • Running time dominated by a single sort