CSE 373: Data Structures and Algorithms

Lecture 18: Minimum Spanning Trees (Graphs)

Instructor: Lilian de Greef
Quarter: Summer 2017
Today

• Spanning Trees
  • Approach #1: DFS
  • Approach #2: Add acyclic edges

• Minimum Spanning Trees
  • Prim’s Algorithm
  • Kruskal’s Algorithm
Announcements

• Midterms
  • I brought midterms with me, can get them after class
  • Next week, will only have them at CSE220 office hours

• Reminder: hw4 due on Friday!
Spanning Trees & Minimum Spanning Trees

For undirected graphs
Introductory Example

All the roads in Seattle are covered in snow. You were asked to shovel or plow snow from roads so that Seattle drivers can travel. Because you don’t want to shovel/plow that many roads, what is the smallest set of roads to clear in order to reconnect Seattle?

Spanning Tree!
Spanning Trees

- Goal: Given a connected undirected graph $G=(V,E)$, find a minimal subset of edges such that $G$ is still connected
  - A graph $G_2 = (V,E_2)$ such that $G_2$ is connected and removing any edge from $E_2$ makes $G_2$ disconnected
Observations

1. Any solution to this problem is a tree
   • Recall a tree does not need a root; just means acyclic
   • For any cycle, could remove an edge and still be connected

2. Solution not unique unless original graph was already a tree

3. Problem ill-defined if original graph not connected
   • So $|E| \geq |V| - 1$

4. A tree with $|V|$ nodes has $|V|$ edges
   • So every solution to the spanning tree problem has $|V|$ edges
Two Approaches

Different algorithmic approaches to the spanning-tree problem:

1. Do a graph traversal (e.g., depth-first search, but any traversal will do), keeping track of edges that form a tree

2. Iterate through edges; add to output any edge that does not create a cycle
Approach #1: Using DFS (Example)

Do a graph traversal, keeping track of edges that form a tree

Stack:

Output:
Approach #1: Spanning Tree via DFS

Spanning Tree via DFS

```java
spanning_tree(Graph G) {
    for each node i: i.marked = false
    for some node i: f(i)
}
f(Node i) {
    i.marked = true
    for each j adjacent to i:
        if(!j.marked) {
            add(i,j) to output
            f(j) // DFS
        }
}
```

Correctness:
DFS reaches each node.
We add one edge to connect it to the already visited nodes.
Order affects result, not correctness.

Time:
Approach #2: Add Acyclic Edges (Example)

Iterate through edges; add to output any edge that does not create a cycle

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

Output:
Approach #2: Add Acyclic Edges

Iterate through edges; output any edge that does not create a cycle.

Correctness (hand-wavy):
- Goal is to build an acyclic connected graph
- When we add an edge, it adds a vertex to the tree
  - Else it would have created a cycle
- The graph is connected, so we reach all vertices

Efficiency:
- Depends on how quickly you can detect cycles
- (Not covered: there is a way to detect these cycles at almost average O(1))
Summary So Far

The **spanning-tree** problem – two approaches:
- Add nodes to partial tree approach
- Add acyclic edges approach

More compelling: we have a *weighted* undirected graph and we want a spanning tree with minimum total weight

a.k.a. the **spanning-tree** problem
Introductory Example: version 2

All the roads in Seattle are covered in snow.
You were asked to shovel or plow snow from roads so that Seattle drivers can travel.
Because you don’t want to shovel/plow that many roads, what is the smallest set of roads to clear in order to reconnect Seattle?
Because you want to do the minimum amount of effort, what is the shortest total distance to clear in order to reconnect Seattle?

Minimum Spanning Tree!
Minimum Spanning Tree: Example Uses

How to most efficiently lay out...

• Telephone lines
• Electrical power lines
• Hydraulic pipes
• TV cables
• Computer networks (like the Internet!)
Minimum Spanning Tree Algorithms

The **minimum-spanning-tree** problem
- Given a weighted undirected graph, give a spanning tree of minimum weight
- Same two approaches, with minor modifications, will work

<table>
<thead>
<tr>
<th>Algorithm for Unweighted Graph</th>
<th>Similar Algorithm for Weighted Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS for shortest path</td>
<td>Algorithm (shortest path)</td>
</tr>
<tr>
<td>DFS for spanning tree</td>
<td>Algorithm (minimum spanning tree)</td>
</tr>
<tr>
<td>Adding acyclic edges approach for spanning tree</td>
<td>Algorithm (minimum spanning tree)</td>
</tr>
</tbody>
</table>
Prim’s Algorithm: Idea

Idea: Grow a tree by adding an edge from the “known” vertices to the “unknown” vertices. *Pick the edge with the smallest weight that connects “known” to “unknown.”*

Recall Dijkstra “picked edge with closest known distance to source”
- That is not what we want here
- Otherwise identical (!)
Prim’s Algorithm: Pseudocode

1. For each node $v$, set $v\text{.cost} = \infty$ and $v\text{.known} = \text{false}$

2. Choose any node $v$
   a) Mark $v$ as known
   b) For each edge $(v, u)$ with weight $w$, set $u\text{.cost}=w$ and $u\text{.prev}=v$

3. While there are unknown nodes in the graph
   a) Select the unknown node $v$ with lowest cost
   b) Mark $v$ as known and add $(v, v\text{.prev})$ to output
   c) For each edge $(v, u)$ with weight $w$,
      
      \[
      \text{if}(w < u\text{.cost}) \{ \\
      \quad \quad \quad u\text{.cost} = w; \\
      \quad \quad \quad u\text{.prev} = v; \\
      \}
      \]
Practice Time!
Using Prim’s Algorithm starting at vertex A, what’s the minimum spanning tree?

A) (A,B), (A,C), (A,D), (D,E), (C,F), (E,G)
B) (B,E), (C,D), (D,A), (E,D), (F,C), (G,E)
C) (B,A), (C,A), (D,A), (E,D), (F,C), (G,E)
D) (B,A), (C,D), (D,A), (E,D), (F,C), (G,D)
(extra space for scratch-work)
Prim’s Algorithm: Example

Let’s start here.

<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(extra space for scratch-work)
Analysis

• Correctness ??
  • A bit tricky
  • Intuitively similar to Dijkstra

• Run-time
  • Same as Dijkstra
  • $O(|E| \log |V|)$ using a priority queue
    • Costs/priorities are just edge-costs, not path-costs
Kruskal’s Algorithm: Idea

Idea:

- Grow a forest out of edges that do not grow a cycle, just like for the spanning tree problem.

But now consider the edges in order by
Kruskal’s Algorithm: Pseudocode

1. Sort edges by weight (better: put in min-heap)
2. Each node in its own set
3. While output size < |V|-1
   • Consider next smallest edge (u, v)
   • If adding edge (u, v) doesn’t introduce cycles, output (u, v)
Example

Edges in sorted order:
1: (A,D), (C,D), (B,E), (D,E)
2: (A,B), (C,F), (A,C)
3: (E,G)
5: (D,G), (B,D)
6: (D,F)
10: (F,G)

Output:
(extra space for scratch-work)
Kruskal’s Algorithm: Correctness

It clearly generates a spanning tree. Call it $T_K$.

Suppose $T_K$ is not minimum:

- Pick another spanning tree $T_{\text{min}}$ with lower cost than $T_K$
- Pick the smallest edge $e_1=(u,v)$ in $T_K$ that is not in $T_{\text{min}}$
- $T_{\text{min}}$ already has a path $p$ in $T_{\text{min}}$ from $u$ to $v$
  - Adding $e_1$ to $T_{\text{min}}$ will create a cycle in $T_{\text{min}}$
- Pick an edge $e_2$ in $p$ that Kruskal’s algorithm considered after adding $e_1$ (must exist: $u$ and $v$ unconnected when $e_1$ considered)
  - $\Rightarrow$ cost($e_2$) $\geq$ cost($e_1$)
  - $\Rightarrow$ can replace $e_2$ with $e_1$ in $T_{\text{min}}$ without increasing cost!
- Keep doing this until $T_{\text{min}}$ is identical to $T_K$
  - $\Rightarrow$ $T_K$ must also be minimal – contradiction!