CS/COE 1501

cs.pitt.edu/~bill/1501/

Weighted Graphs
We define graphs as sets of vertices and edges. However, we’ll often want to be able to reason about the bandwidth, distance, capacity, etc. of the real world things modeled by the edges in our graphs:

- Whether a link is 1 gigabit or 10 megabit will drastically affect our analysis of traffic flowing through a network.
- Having a road between two cities that is a 1 lane country road is very different from having a 4 lane highway.
- If two airports are 2000 miles apart, the number of flights going in and out between them will be drastically different from airports 200 miles apart.

Last time, we said spatial layouts of graphs were irrelevant.
We can represent such information with edge weights

- How do we store edge weights?
  - Adjacency matrix?
  - Adjacency list?
  - Do we need a whole new graph representation?

- How do weights affect finding spanning trees/shortest paths?
  - The weighted variants of these problems are called finding the *minimum spanning tree* and the *weighted shortest path*
Minimum spanning trees (MST)

- Graphs can potentially have multiple spanning trees.
- MST is the spanning tree that has the minimum sum of the weights of its edges.
Prim’s algorithm

- Initialize T to contain the starting vertex
  - T will eventually become the MST
- While there are vertices not in T:
  - Find the minimum-weight edge that connects a vertex in T to a vertex not yet in T
  - Add the edge with its vertex to T
Prim’s algorithm in the abstract
At each step, check all possible edges

For a complete graph:
  ○ First iteration:
    ■ v - 1 possible edges
  ○ Next iteration:
    ■ 2(v - 2) possibilities
      ● Each vertex in T shared v-1 edges with other vertices, but
        the edges they shared with each other already in T
  ○ Next:
    ■ 3(v - 3) possibilities
  ○ ...

Runtime:
  ○ $\sum_{i=1}^{v} (i \times (v - i))$
    ■ Evaluates to $\Theta(v^3)$
Do we need to look through all remaining edges?

- No! We only need to consider the *best* edge for possible for each vertex!
Prim’s algorithm: Parent/best edge

Parent:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>--</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

Best Edge:

|   | 0 | 4 | 8 | 2 | 5 | 1 | 2 |
OK, so what’s our runtime?

- For every vertex we add to T, we’ll need to check all of its neighbors to check for edges to add to T next
  - Let’s assume we use an adjacency matrix:
    - Takes $\Theta(v)$ to check the neighbors of a given vertex
    - Time to update parent/best edge arrays?
    - Time to pick next vertex?
  - What about with an adjacency list?
What about a faster way to pick the best edge?

- Sounds like a job for a priority queue!
  - Priority queues can remove the min value stored in them in $\Theta(lg \ n)$
    - Also $\Theta(lg \ n)$ to add to the priority queue
- What does our algorithm look like now?
  - Visit a vertex
  - Add edges coming out of it to a PQ
  - While there are unvisited vertices, pop from the PQ for the next vertex to visit and repeat
Prim’s with a priority queue

PQ:
1: (0, 2)
2: (5, 3)
3: (1, 4)
4: (2, 5)
5: (2, 3)
5: (0, 3)
5: (2, 1)
6: (0, 1)
6: (2, 4)
6: (5, 4)
Have to insert all \( e \) edges into the priority queue

- In the worst case, we’ll also have to remove all \( e \) edges

So we have:

- \( e \cdot \Theta(\lg e) + e \cdot \Theta(\lg e) \)

- \( = \Theta(2 \cdot e \cdot \lg e) \)

- \( = \Theta(e \cdot \lg e) \)

This algorithm is known as \textit{lazy Prim’s}
I suppose we could be less lazy...

Just like with the “parent/best-edge” implementation, we only need the best edge for each vertex
  ○ PQ will need to be indexable

This is the idea of *eager Prim’s*
  ○ Runtime is $\Theta(e \lg v)$
Comparison of Prim’s implementations

● Parent/Best-edge array Prim’s
  ○ Runtime: $\Theta(v^2)$
  ○ Space: $\Theta(v)$

● Lazy Prim’s
  ○ Runtime: $\Theta(e \lg e)$
  ○ Space: $\Theta(e)$
    ○ Requires a PQ

● Eager Prim’s
  ○ Runtime: $\Theta(e \lg v)$
  ○ Space: $\Theta(v)$
    ○ Requires an indexable PQ

How do these compare?
Weighted shortest path

- Dijkstra’s algorithm:
  - Set a distance value of MAX_INT for all vertices except start
  - Set cur = start
  - While destination is not visited:
    - For each unvisited neighbor of cur:
      - Compute tentative distance from start to the unvisited neighbor through cur
      - Update any vertices for which a lesser distance is computed
    - Mark cur as visited
    - Let cur be the unvisited vertex with the smallest tentative distance from start
Dijkstra’s example

Distance

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>21</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Via

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How to implement?

- Best path/parent array?
  - Runtime?

- PQ?
  - Turns out to be very similar to Eager Prim’s
    - Storing paths instead of edges
  - Runtime?
Back to MSTs: One more MST algorithm

- Kruskal’s MST:
  - Insert all edges into a PQ
  - Grab the min edge from the PQ that does not create a cycle in the MST
  - Remove it from the PQ and add it to the MST
Kruskal’s example

PQ:
1: (0, 2)
2: (3, 5)
3: (1, 4)
4: (2, 5)
5: (2, 3)
5: (0, 3)
5: (1, 2)
6: (0, 1)
6: (2, 4)
6: (4, 5)
Kruskal’s runtime

- Instead of building up the MST starting from a single vertex, we build it up using edges all over the graph.
- How do we efficiently implement cycle detection and/or connectedness checking?