ASSORTED MINUTIAE

• HW5 is out
  • Write up has a Minimum spanning tree question, which we’re covering today
  • Code due next Wednesday, as usual, Write up will be due on Friday.

• H2 finally graded
  • 10 or so students no grade yet, out this weekend. Problems with the script.

• Feedback on HW4 code out this weekend
TODAY’S LECTURE

• Minimum Spanning Trees
  • Prim’s Algorithm (vertex based solution)
  • Kruskal’s Algorithm (edge based solution)
SPANNING TREES

Given a connected undirected graph $G=(V,E)$, find a 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. Problem not defined if original graph not connected. Therefore, we know $|E| \geq |V|-1$

2. 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

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

4. A tree with $|V|$ nodes has $|V|-1$ edges
   - So every solution to the spanning tree problem has $|V|-1$ edges
MOTIVATION

A **spanning tree** connects all the nodes with as few edges as possible.

In most compelling uses, we have a *weighted* undirected graph and we want a tree of least total cost.

Example: Electrical wiring for a house or clock wires on a chip.

Example: A road network if you cared about asphalt cost rather than travel time.

This is the **minimum spanning tree** problem.
LAST CLASS

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
spanning_tree(Graph G) {
    for each node v:
        v.marked = false
        dfs(someRandomStartNode)
}

dfs(Vertex a) { // recursive DFS
    a.marked = true
    for each b adjacent to a:
        if(!b.marked) {
            add(a,b) to output
            dfs(b)
        }
}
MINIMAL SPANNING TREES

• How do we get a minimal spanning tree from a traversal?
  • What parts of a traversal can we change?
  • Select which vertex we visit next by which is closest to an old vertex
PRIM’S ALGORITHM

• A traversal
  • Pick a start node
  • Keep track of all of the vertices you can reach
  • Add the vertex that is closest (has the edge with smallest weight) to the current spanning tree.

• Is this similar to something we’ve seen before?
PRIM’S ALGORITHM

• Modify Dijkstra’s algorithm
  • Instead of measuring the total length from start to the new vertex, now we only care about the edge from our current spanning tree to new nodes
THE ALGORITHM

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 \( w \),
      
      if \( w < u.\text{cost} \) {
          \begin{align*}
          u.\text{cost} &= w; \\
          u.\text{prev} &= v;
          \end{align*}
      

EXAMPLE

\[
\begin{array}{c|c|c|c}
\text{vertex} & \text{known?} & \text{cost} & \text{prev} \\
A & \infty & \infty & \\
B & \infty & \infty & \\
C & \infty & \infty & \\
D & \infty & \infty & \\
E & \infty & \infty & \\
F & \infty & \infty & \\
G & \infty & \infty & \\
\end{array}
\]
A vertex known? cost prev
A Y 0 A
B 2 A
C 1 D
D Y 1 A
E 1 D
F 6 D
G 5 D
vertex | known? | cost | prev
--- | --- | --- | ---
A | Y | 0 | 
B | | 2 | A |
C | Y | 1 | D |
D | Y | 1 | A |
E | | 1 | D |
F | | 2 | C |
G | | 5 | D |
<table>
<thead>
<tr>
<th>vertex</th>
<th>known?</th>
<th>cost</th>
<th>prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Y</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Y</td>
<td>1</td>
<td>E</td>
</tr>
<tr>
<td>C</td>
<td>Y</td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>Y</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>Y</td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>3</td>
<td>E</td>
</tr>
</tbody>
</table>
A known? cost prev
A Y 0
B Y 1 E
C Y 1 D
D Y 1 A
E Y 1 D
F Y 2 C
G 3 E
A
B
C
D
E
F
G

vertex | known? | cost | prev
----- | ------ | ---- | ----
A     | Y     | 0    |      
B     | Y     | 1    | E    
C     | Y     | 1    | D    
D     | Y     | 1    | A    
E     | Y     | 1    | D    
F     | Y     | 2    | C    
G     | Y     | 3    | E    

A vertex known?
PRIM’S ALGORITHM

• Does this give us the correct solution? Why?
  • If we consider the “known” cloud as a single vertex, we will never add edges that form a cycle
  • Each time, we take the edge that has minimal weight going out of the vertex.
  • This is the cheapest way of connecting the two subgraphs.
PRIM’S ALGORITHM

- What is the runtime?
  - Traversals go through all of the edges, in the worst case
    - Need to check if an edge forms a cycle or if it has minimal weight.
    - We can check if it forms a cycle by verifying if the other vertex is in the “known cloud” $O(1)$
    - How long to check if it is minimal? $O(\log |V|)$ if we use a priority queue
PRIM’S ALGORITHM

- $O(|E| \log |V|)$
  - We can use a priority queue to store all of our vertices, and let the edges to them be the priority.
  - Use the decreaseKey() function when the edge to a vertex changes.
  - This also works for Dijkstra’s algorithm, but you aren’t required to do it for HW5
  - Without the priority queue, both Prim’s and Dijkstra’s run in $O(|E||V|)$
KRUSKAL’S ALGORITHM

• Prim’s algorithm works from the vertices, and builds a contiguous spanning tree
  • The spanning tree grows out from a single vertex
• Kruskal’s Algorithm adds edges based on their weight
  • Must check for cycles
  • Use the union-find data structure to speed up this operation
KRUSKAL’S ALGORITHM

• Pseudocode:
  • Sort the edges (or place them into a heap)
  • Create a union-find data structure with all separate vertices
  • For each edge, add it to the minimum spanning tree if the two vertices don’t have the same representative in the union find
  • Union the two vertices in the union find
  • Stop after you’ve added |V|-1 edges
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:

Note: At each step, the union/find sets are the trees in the forest
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: \((A,D)\)

Note: At each step, the union/find sets are the trees in the forest
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: (A,D), (C,D)

Note: At each step, the union/find sets are the trees in the forest
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: (A,D), (C,D), (B,E)

Note: At each step, the union/find sets are the trees in the forest
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: (A,D), (C,D), (B,E), (D,E)

Note: At each step, the union/find sets are the trees in the forest
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: (A,D), (C,D), (B,E), (D,E)

Note: At each step, the union/find sets are the trees in the forest
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: (A,D), (C,D), (B,E), (D,E), (C,F)

Note: At each step, the union/find sets are the trees in the forest
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: (A,D), (C,D), (B,E), (D,E), (C,F)

Note: At each step, the union/find sets are the trees in the forest.
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: (A,D), (C,D), (B,E), (D,E), (C,F), (E,G)

Note: At each step, the union/find sets are the trees in the forest
KRUSKAL’S ALGORITHM

• **Runtime**
  - Put edges into a heap $O(|E|)$ Floyd’s method!
  - Until the MST is complete:
    - Pull the minimum edge out of the heap $O(\log |E|)$
    - Check if it forms a cycle $O(\log |V|)$
  - How many times does the loop run? $O(E)$
  - $O(|E| \log |E|)$
COMPARISONS

• Prim’s
  • \(O(|E| \log |V|)\)

• Kruskal’s
  • \(O(|E| \log |E|)\)

• Since \(|E|\) must be at least \(|V|-1\) for the graph to be connected, which do we prefer?
COMPARISONS

• Prim’s
  • \( O(|E| \log |V|) \)

• Kruskal’s
  • \( O(|E| \log |E|) \)

• Since \(|E|\) must be at least \(|V|-1\) for the graph to be connected, which do we prefer?
  • Since \(|E|\) is at most \(|V|^2\), \( \log|E| \) is at most \( \log(|V|^2) \) which is \( 2\log|V| \).
  • So \( \log|E| \) is \( O(\log|V|) \)
CONCLUSIONS

• Prim’s and Kruskal’s both run in $O(|E| \log |V|)$

• An undirected graph has a unique minimum spanning tree if all of its edge weights are unique.

• If graphs have multiple edges of the same weight, it is possible (but not necessary) that there are many spanning trees of the same weight
NEXT WEEK

• Graph algorithm runtimes
• Conclude Graphs
• New Algorithm Analysis technique
  • Recurrences
• Start sorting