CS16 Section 6

Monday July 12 - Wednesday July 14
Agenda

1. Icebreakers
2. Dijkstra’s Algorithm
3. MST: Prim’s & Kruskal
4. Decision Trees
Shortest Distance Graphs (Dijkstra's) and MSTs
Shortest Distance Graph vs. MST

1. Shortest Distance Graph:
   a. Shortest path from a specified start vertex to every other vertex
   b. Minimizes this path weight
   c. Have to specify a starting vertex

2. Minimum Spanning Tree:
   a. A spanning tree with minimum total edge weight
   b. Every vertex is reached in the tree
   c. Minimizes the sum of all edges in the graph
   d. Do not specify a starting vertex
Dijkstra: Review

- **General Idea**: Given a starting node, finds the shortest distance from this node to each other node (Shortest Distance Graph)
- **Runtime depends on the implementation**
  - Binary Heap: $O((|V| + |E|) \log|V|)$
  - Array and LinkedList: $O(|V|^2)$
- **NOTE**: Dijkstra will not work with negative edge weights
function dijkstra(G, s):
    // Input: A graph G with vertices V, and a start vertex s
    // Output: Nothing
    // Purpose: Decorate nodes with shortest distance from s
    for v in V: // O(|V|)
        v.dist = infinity
        v.prev = null
    s.dist = 0

    PQ = PriorityQueue(V) // Depends on PQ implementation!
    while PQ not empty: // O(|V|)
        u = PQ.removeMin() // Depends on PQ implementation!
        for all edges (u, v): // O(|E|)
            if v.dist > u.dist + cost(u, v):
                v.dist = u.dist + cost(u, v)
                v.prev = u
                PQ.replaceKey(v, v.dist) // Depends on PQ implementation!
Prim-Jarnik

- Similar to Dijkstra’s
- **General Idea:** Finds the Minimum Spanning Tree given a priority queue
- Runtime using a Heap implementation: $O( (|V| + |E|) \log |V| )$
- Picking the lowest cost edge at each iteration . . . what kind of algorithm does it sound like?
- Cool animation here: [https://visualgo.net/en/mst](https://visualgo.net/en/mst) (Show the Prim’s animation)
MST: Prim-Jarnik and Kruskal
Subsection 1.1 -- Prim Jarnik’s Algorithm

- Similar to Dijkstra’s
- **General Idea**: Finds the Minimum Spanning Tree given a priority queue
- Runtime using a Heap implementation: $O( (|V| + |E|) \log|V| )$
- Picking the lowest cost edge at each iteration . . . what kind of algorithm does it sound like?
- Cool animation here: [https://visualgo.net/en/mst](https://visualgo.net/en/mst) (Show the Prim’s animation)
function prim(G):
    // Input: weighted, undirected graph G with vertices V
    // Output: list of edges in MST
    for all v in V:
        v.cost = ∞
        v.prev = null
    source = a random v in V
    source.cost = 0
    MST = []
    PQ = PriorityQueue(V) // priorities will be v.cost values
    while PQ is not empty:
        v = PQ.removeMin()
        if v.prev != null:
            MST.append((v, v.prev))
        for all incident edges (v,u) of v:
            if u.cost > (v,u).weight:
                u.cost = (v,u).weight
                u.prev = v
                PQ.replaceKey(u, u.cost)
    return MST
Subsection 1.1 -- Prim Jarnik’s Algorithm

- Split up into partners and take out a sheet of paper
- Hand simulate Prim Jarnik’s on the graph shown in the next slide
- We will give you 5 to 10 minutes to hand simulate this graph, and then we will go over the answer together.
Random node set to cost 0

\[ PQ = [(0, A), (\infty, B), (\infty, C), (\infty, D), (\infty, E), (\infty, F)] \]
Dequeue from PQ and update neighbors

\[
PQ = [(4, B), (5, D), (\infty, C), (\infty, E), (\infty, F)]
\]
Dequeue from PQ and update neighbors

\[
PQ = \left[ (4, C), (4, D), (6, E), (8, F) \right]
\]
Dequeue from PQ and update neighbors

\[
PQ = [(2,E), (4,D), (8,F)]
\]
$PQ = \{ (4, D), (4, F) \}$

Dequeue from PQ and update neighbors
Dequeue from PQ and update neighbors

\[ PQ = [(3, F)] \]
PQ = [ ]

Dequeue from PQ and update neighbors
Subsection 1.2 -- Kruskal’s Algorithm

- Union Find
- Intuition -- ‘clouds’, used to make sure that the edges that are added do not create cycles
- Path compression -- how does it work?
- Cool animation here: https://visualgo.net/en/mst (Show the Kruskal’s animation)
Path Compression

- Instead of traversing up tree every time D's cloud is asked for
  - We only search for D's root once
  - As we follow chain of parents to A we set parents of D & C to A

\[O(\log |\mathcal{V}|)\]

Amortized \[O(1)\]
Subsection 1.2 -- Kruskal’s Algorithm

- Split up into partners and take out a sheet of paper
- Hand simulate Kruskal’s on the graph shown in the next slide
- We will give you 5 to 10 minutes to hand simulate this graph, and then we will go over the answer together.
edges = [(C, E), (D, F), (B, C), (E, F), (B, D), (A, B), (A, D), (B, E), (B, F)]
edges = [(C, E), (D, F), (B, C), (E, F), (B, D), (A, B), (A, D), (B, E), (B, F)]
edges = [(D,F), (B,C), (E,F), (B,D), (A,B), (A,D), (B,E), (B,F)]
edges = [(B, C), (E, F), (B, D), (A, B), (A, D), (B, E), (B, F)]
edges = [(E,F), (B,D), (A,B), (A,D), (B,E), (B,F)]
edges = [ (B,D), (A,B), (A,D), (B,E), (B,F) ]
BD cannot be added because it would lead to a cycle

edges = [(A, B), (A, D), (B, E), (B, F)]
edges = [(A, D), (B, E), (B, F)]
AD cannot be added because it would lead to a cycle

edges = [(B,E), (B,F)]
BE cannot be added because it would lead to a cycle

edges = [(B, F)]
BF cannot be added because it would lead to a cycle

edges = [ ]
Kruskal Pseudocode

function kruskal(G):
    // Input: undirected, weighted graph G
    // Output: list of edges in MST
    for vertices v in G:
        makeCloud(v) // put every vertex into its own set
    MST = []
    Sort all edges
    for all edges (u,v) in G sorted by weight:
        if u and v are not in same cloud:
            add (u,v) to MST
            merge clouds containing u and v
    return MST
Decision Trees
Decision Tree Structure
## Decision Trees: Training Data

<table>
<thead>
<tr>
<th>Ex.</th>
<th>Alt</th>
<th>Bar</th>
<th>Fri</th>
<th>Hun</th>
<th>Pat</th>
<th>Price</th>
<th>Rain</th>
<th>Res</th>
<th>Type</th>
<th>Est</th>
<th>Classif.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Some</td>
<td>$$$$</td>
<td>No</td>
<td>Yes</td>
<td>French</td>
<td>0-10</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Full</td>
<td>$</td>
<td>No</td>
<td>No</td>
<td>Thai</td>
<td>30-60</td>
<td>No</td>
</tr>
<tr>
<td>3</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Some</td>
<td>$</td>
<td>No</td>
<td>No</td>
<td>Burger</td>
<td>0-10</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Full</td>
<td>$</td>
<td>Yes</td>
<td>No</td>
<td>Thai</td>
<td>10-30</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Full</td>
<td>$$$$</td>
<td>No</td>
<td>Yes</td>
<td>French</td>
<td>&gt;60</td>
<td>No</td>
</tr>
<tr>
<td>6</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Some</td>
<td>$</td>
<td>Yes</td>
<td>Yes</td>
<td>Italian</td>
<td>0-10</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>None</td>
<td>$</td>
<td>Yes</td>
<td>No</td>
<td>Burger</td>
<td>0-10</td>
<td>No</td>
</tr>
<tr>
<td>8</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Some</td>
<td>$$</td>
<td>Yes</td>
<td>Yes</td>
<td>Thai</td>
<td>0-10</td>
<td>Yes</td>
</tr>
<tr>
<td>9</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Full</td>
<td>$</td>
<td>Yes</td>
<td>No</td>
<td>Burger</td>
<td>&gt;60</td>
<td>No</td>
</tr>
<tr>
<td>10</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Full</td>
<td>$$$$</td>
<td>No</td>
<td>Yes</td>
<td>Italian</td>
<td>10-30</td>
<td>No</td>
</tr>
<tr>
<td>11</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>None</td>
<td>$</td>
<td>No</td>
<td>No</td>
<td>Thai</td>
<td>0-10</td>
<td>No</td>
</tr>
<tr>
<td>12</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Full</td>
<td>$</td>
<td>No</td>
<td>No</td>
<td>Burger</td>
<td>30-60</td>
<td>Yes</td>
</tr>
</tbody>
</table>
ID3 Review

- Creating a tree where leaves are output classifications and other nodes are attributes
- Edges connecting nodes represent the possible values the parent attribute can have
- Try to make the tree as small as possible--how?
ID3 Review: What to Split on

- We want to choose attribute nodes that will lead to high confidence output classifications
  - Ex. Consider 10 seahorses as a dataset. 5 data points are classified as cute, 5 are not. If the 5 seahorses labeled cute all correspond to being small, and the 5 who aren't cute correspond to being large, we perfectly learn to classify the data from this. Compare this to the data point for number of eyes which is the same for every seahorse. This is not a good attribute to learn about cuteness of seahorses since it's always the same.
ID3 Review: What to Split On

- What does this look like in code?
  - Compute the entropy of an attribute's children
  - Compute the information gain of an attribute
  - Split on the attribute with highest information gain
Calculating Information Gain

- Information Gain is how much *information we gain* by splitting on an attribute
  - Entropy -- $H(\text{data})$ -- is a measure of homogeneity (How much information we have)
  - Info Gain = $H(\text{data})$ - Remainder(attr)
- Remainder(attr) = Weighted Sum of remaining entropy after splitting

\[
R(\text{Att}) = \sum_{i=1}^{d} \frac{|S_i|}{|E|} \cdot H(S_i),
\]

- Calculating Entropy

\[
q = \frac{p}{p + n}, \quad H(S) = - \left( q \cdot \log_2 q + (1 - q) \cdot \log_2 (1 - q) \right)
\]
```python
id3_algorithm(data, attributes, parent_data):
    if data is empty:
        return new node w/ most frequent classification in parent_data
    else if all examples in data have same classification:
        return new node with that classification
    else if attributes is empty:
        return new node w/ most frequent classification in data
    else:
        A = attribute with largest information gain
        tree = new decision tree with root A
        for each value a of attribute A
            new_data = all examples in data such that example.A = a
            subtree = id3_algorithm(new_data, attributes - A, data)
            attach subtree to tree with branch labeled "a"
        return tree
```
<table>
<thead>
<tr>
<th>Input Pat</th>
<th>Classif.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Some</td>
<td>Yes</td>
</tr>
<tr>
<td>Full</td>
<td>No</td>
</tr>
<tr>
<td>Some</td>
<td>Yes</td>
</tr>
<tr>
<td>Full</td>
<td>Yes</td>
</tr>
<tr>
<td>Full</td>
<td>No</td>
</tr>
<tr>
<td>Some</td>
<td>Yes</td>
</tr>
<tr>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>Some</td>
<td>Yes</td>
</tr>
<tr>
<td>Full</td>
<td>No</td>
</tr>
<tr>
<td>Full</td>
<td>No</td>
</tr>
<tr>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>Full</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Mini-Assignment
Mini Assignment Graph
Problem 1 Solution: MST
Problem 2 Solution -- Kruskal’s Algorithm
Problem 2 Solution -- Kruskal’s Algorithm
You could add BD or CD in this step
Problem 2 Solution -- Kruskal’s Algorithm
1. A spanning tree and a minimum spanning tree are both a subset of edges in a graph that span every vertex (forming a tree). A minimum spanning tree is the spanning tree with the least possible total edge weight for a given graph.

2. A spanning forest is the collection of STs for unconnected graphs, while a MSF is the collection of MSTs.