Shortest Paths: Dijkstra’s Algorithm
SSSP: Context

- Finding the shortest path from a source to goal, or source to every node
- Google/Apple maps
Recall: Breadth-First Search

- Finds a path of fewest edges from a source to every reachable node in the graph
Recall: Breadth-First Search

- Finds a path of fewest edges from a source to every reachable node in the graph

<table>
<thead>
<tr>
<th>Destination</th>
<th>Path</th>
<th>Path length</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>[S]</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>[S, A]</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>[S, A, B]</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>[S, D, C]</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>[S, D]</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>none</td>
<td>∞</td>
</tr>
</tbody>
</table>
Introduce: Weighted Graphs

- Each edge has a **weight**, or **cost**, associated with it

<table>
<thead>
<tr>
<th>Destination</th>
<th>Path</th>
<th>Path cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>[S]</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>[S, A]</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>[S, A, B]</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>[S, D, C]</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>[S, D]</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>none</td>
<td>∞</td>
</tr>
</tbody>
</table>
Introduce: Weighted Graphs

- Edge weight could represent a time cost, financial cost, or a relation between two nodes

---

**Destination** | **Path** | **Path cost**
--- | --- | ---
SEA | [S] | 0
ATL | [S, A] | 4.18
BDL | [S, A, B] | 6.00
CLE | [S, A, B C] | 7.13
DUB | [S, D] | 9.56
I | none | ∞
Problem: Breadth-First Search on Weighted Graphs

- BFS is not designed to consider edge weights!

```
1  procedure BFS(G, start_v):
2      let S be a queue
3      S.enqueue(start_v)
4      while S is not empty
5          v = S.dequeue()
6          if v is the goal:
7              return v
8          for all edges from v to w in G.adjacentEdges(v) do
9              if w is not labeled as discovered:
10                 label w as discovered
11                 w.parent = v
12                 S.enqueue(w)
```
Solution: Dijkstra’s Algorithm

- Explore nodes in a **greedy, shortest-path-first** manner
- Key feature: explore (dequeue/pop) the closest node not yet explored.
Pseudocode: With Inefficiencies

- Overview of pseudocode

```plaintext
Dijkstra(Graph, source):
  create vertex list Q
  for each vertex v in Graph:
    v.dist = INFINITY    //dist represents distance from source
    v.parent = null
    add v to Q
  source.dist = 0
  while Q is not empty:
    u = vertex in Q with smallest "dist"
    remove u from Q
    for each neighbor v of u:    // only v that are still in Q
      alt = u.dist + weight(u, v)
      if alt < v.dist:
        v.dist = alt
        v.parent = u
  // return depends on the application
```
Pseudocode: Addressing Inefficiencies

- Use your Data Structures...

```plaintext
function Dijkstra(Graph, source):
    create vertex set Q
    source.dist = 0
    for each vertex v in Graph:
        v.dist = INFINITY
        v.prev = null
        Q.addWithPriority(v, v.dist)

    while Q is not empty:
        u = Q.extractMin() // O(log(n))

        for each neighbor v of u:  // only v that are still in Q
            alt = u.dist + weight(u, v)
            if alt < v.dist:
                v.dist = alt
                v.prev = u
                Q.decreasePriority(v, v.dist) //O(log(n)) if we can find it quickly

    // return depends on the application
```
Recall: PriorityQueue (Min-Heap Implementation)

- add(Object key, int priority) - O(log(n))
- extractMin() - O(log(n))
- decreaseKey(Object key, int newPriority) - O(log(n))
GYHD: Example

Assuming S is the source

<table>
<thead>
<tr>
<th>Destination</th>
<th>Cost</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conditions for Dijkstra’s

- No negative cycles; a negative cycle will always be a problem
- In fact, no negative edges; a negative edge will sometimes be a problem
- GYHD: come up with a graph that has no negative cycles, has a negative edge, where Dijkstra’s algorithm would not find the shortest path from a source to a destination
Counterexample

![Graph Diagram]

- **Nodes:** S, B, D, A
- **Edges and Weights:**
  - S to B: 50
  - B to D: 1
  - S to A: 101
  - A to B: -100
If Time: Java implementation

- `compareTo(Node other)` vs. `compare(Node a, Node b)`
- Cats
- Time complexity and ease of implementation
Finally

- Thank you!
- PLEASE take survey on teaching feedback - will send to you shortly