Graph Representations, BFS, and Dijkstra’s Algorithm
CSE 373 Winter 2020

Instructor: Hannah C. Tang

Teaching Assistants:
Aaron Johnston      Ethan Knutson      Nathan Lipiarski
Amanda Park        Farrell Fileas      Sam Long
Anish Velagapudi   Howard Xiao        Yifan Bai
Brian Chan         Jade Watkins       Yuma Tou
Elena Spasova      Lea Quan
Announcements

- Midterm is *this Friday*
  - If your student number ends in an odd number, go to KNE 210
  - If your student ends in an even number, go to KNE 220
  - Workshops will be focused on your midterm questions
  - Review session Thursday night: 4:30-6:30 @ ARC 147
Lecture Outline

❖ Graph Representations

❖ Graph Traversals: BFS

❖ Shortest Paths: Dijkstra’s Algorithm
Review: Depth-First Search

```java
connected(Node s, Node t) {
    if (s == t) {
        return true;
    } else {
        s.visited = true;
        for (Node n : s.neighbors) {
            if (n.visited) {
                continue;
            }
            if (connected(n, t)) {
                return true;
            }
        }
        return false;
    }
}
```

What data structure should we store the graph in?
Graph ADT

Graph ADT. A collection of vertices and the edges connecting them.

- We can query for vertices connected to, or edges leaving from, a vertex \( v \)
- Edges are specified as pairs of vertices
- We can add/remove edges from the graph

Key operations include:
- `getAllVertices()`
- `getAllEdges()`
- `addEdge(v, w)`
- `getAllEdgesFrom(v)`
- `hasEdge(v, w)`
Graph Data Structures

- Just as we saw multiple representations for a tree, there are multiple data structures that implement the Graph ADT
  - Node class with left/right pointers vs Binary Heap’s array representation

- Option 1: Adjacency Matrix
- Option 2: Edge Sets
- Option 2: Adjacency List
Graph Data Structure #1: Adjacency Matrix

- Directed Graphs:
  - Each edge is represented twice in the matrix: simplicity vs space

- Undirected Graphs:
  - Each edge is represented twice in the matrix: simplicity vs space
What is the runtime for `getAllEdges()` if we use an adjacency matrix representation, where V is the number of vertices and E is the number of edges?

A. $\Theta(V)$  
B. $\Theta(V + E)$  
C. $\Theta(V^2)$  
D. $\Theta(V*E)$  
E. I’m not sure ...
Graph Data Structure #2: Edge Set

❖ A simple collection of edges
  ▪ Eg: HashSet<Edge>. Recall that each edge is a (possibly directed) pair of vertices

❖ Directed Graph:

❖ Undirected Graph:

\{0, 1\}, \{1, 0\}, \{1, 2\}, \{0, 2\}

(vertex ordering matters for directed graphs)

\{0, 1\}, \{1, 2\}, \{0, 2\)
Graph Data Structure #3: Adjacency List

- Array of lists of vertices, indexed by vertex
  - Most popular approach

- Directed Graph:

- Undirected Graph:
  - As with adjacency matrix, each edge is represented twice
What is the runtime for `getAllEdges()` if we use an **adjacency list** representation, where \( V \) is the number of vertices and \( E \) is the number of edges?

A. \( \Theta(V) \)
B. \( \Theta(V + E) \)
C. \( \Theta(V^2) \)
D. \( \Theta(V\times E) \)
E. I’m not sure ...
Graph Representations

- In practice, adjacency lists are most common
  - Many graph algorithms rely heavily on `getAllEdgesFrom(v)`
  - Most graphs are sparse (i.e., not many edges)

<table>
<thead>
<tr>
<th></th>
<th><code>getAllEdgesFrom(v)</code></th>
<th><code>hasEdge(v, w)</code></th>
<th><code>getAllEdges()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjacency Matrix</td>
<td>Θ(V)</td>
<td>Θ(1)</td>
<td>Θ(V²)</td>
</tr>
<tr>
<td>Edge Set</td>
<td>Θ(E)</td>
<td>Θ(E)</td>
<td>Θ(E)</td>
</tr>
<tr>
<td>Adjacency List</td>
<td>O(V)</td>
<td>Θ(degree(v))</td>
<td>Θ(E + V)</td>
</tr>
</tbody>
</table>

*best and worst case don’t match, so no Θ-bound exists*
Lecture Outline

❖ Graph Representations

❖ Graph Traversals: BFS

❖ Shortest Paths: Dijkstra’s Algorithm
Review: Depth-First Search

- DFS goes “deep” instead of “broad”
  - 0, 1, 2, 3
  - 0, 1, 5, 6, 7, 3

```java
class Node {
    int value;
    Node[] neighbors;
    boolean visited;
}

public boolean connected(Node s, Node t) {
    if (s == t) {
        return true;
    } else {
        s.visited = true;
        for (Node n : s.neighbors) {
            if (n.visited) {
                continue;
            } else {
                if (connected(n, t)) {
                    return true;
                }
            }
        }
        return false;
    }
}
```
Breadth-First Search (1 of 2)

- Breadth-First Search (BFS) is the graph analogue of a tree’s level-order traversal
  - Goes “broad” instead of “deep”
  - Added benefit: finds the shortest path from as source to all other vertices, not just a single target t!

- Challenge: how would you implement BFS from s?
  - Hint 1: How will you visit vertices in BFS order?
  - Hint 2: You’ll need to use some kind of data structure
  - Hint 3: Don’t use recursion
Breadth-First Search (2 of 2)

- Demo: https://docs.google.com/presentation/d/1JoYCeIH4YE6IkSMq_LfTJMzJ00WxDj7rEa49gYmAtc4/present?ueb=true&slide=id.g76e0dad85_2_380

- Note: we call the need-to-explore vertices “the fringe”

```java
shortestPaths(Node s, Graph g) {
    Queue fringe;
    fringe.enqueue(s);
    Map<Node, Integer> distances;
    distances.setAll(g.allNodes(), ∞);
    Map<Node, Node> previousNode;
    while (!fringe.isEmpty()) {
        Node n = fringe.dequeue();
        for (Node i : n.neighbors) {
            if (distances[i] < distances[n] + 1) {
                continue;
            } else {
                distances[i] = distances[n] + 1;
                fringe.enqueue(i);
                previousNode[i] = n;
            }
        }
    }
}
```
Breadth-First Search Application

- Graph with vertices as actors and edges as movies

- Perform BFS from Kevin Bacon (or your actor of choice)
  - [https://en.wikipedia.org/wiki/Erd%C5%91s%E2%80%93Bacon_number](https://en.wikipedia.org/wiki/Erd%C5%91s%E2%80%93Bacon_number)
Lecture Outline

❖ Graph Representations

❖ Graph Traversals: BFS

❖ Shortest Paths: Dijkstra’s Algorithm
Shortest Paths in Weighted Graphs

- Breadth-First Search (BFS) finds the shortest path from a source to all other vertices in an unweighted graph

- Let’s try it in a map
Breadth First Search for Mapping Applications

- BFS yields the wrong route from s to t
  - Length of ~190 instead of ~110
  - We need an algorithm that takes into account edge weights!
Adapting BFS

Can we adapt BFS to accommodate edge weights?

```java
def shortestPaths(Node s, Graph g) {
    Queue fringe;
    fringe.enqueue(s);
    Map<Node, Integer> distances;
    distances.setAll(g.allNodes(), ∞);
    Map<Node, Node> previousNode;

    while (!fringe.isEmpty()) {
        Node n = fringe.dequeue();
        for (Node i : n.neighbors) {
            if (distances[i] < distances[n] + 1) {
                continue;
            } else {
                distances[i] = distances[n] + 1;
                fringe.enqueue(i);
                previousNode[i] = n;
            }
        }
    }
}
```
Dijkstra’s Algorithm

\[
\text{shortestPaths}(\text{Node } s, \text{ Graph } g) \{
\quad \text{Queue fringe;}
\quad \text{fringe} \text{.enqueue}(s);
\quad \text{Map}<\text{Node, Integer}> \text{ distances;}
\quad \text{Map}<\text{Node, Node}> \text{ previousNode;}
\endgroup
\]

\[
\text{while} \ (\text{! fringe.isEmpty()}) \ {\text{}}\{
\quad \text{Node } n = \text{fringe.dequeue();}
\quad \text{for } (\text{Node i : n.neighbors}) \ {\text{}}\{
\quad\quad \text{if } (\text{distances[i]} < \text{distances[n]} + \text{g.edgeWeight(n, i)}) \ {\text{}}\{
\quad\quad\quad \text{continue;}
\quad\quad\}\text{else } \{\text{}}\{
\quad\quad\quad \text{distances[i]} = \text{distances[n]} + \text{g.edgeWeight(n, i)};
\quad\quad\quad \text{fringe.enqueue(i);}
\quad\quad\quad \text{previousNode[i]} = n;
\quad\quad\}\}\}\}\}
\]
Dijkstra’s Algorithm

- Replace the queue with a priority queue whose priorities are distance from s

- The key operation: “relaxing” an edge on the fringe

- Dijkstra’s returns optimal results only in graphs with non-negative edge weights!

```java
dijkstras(Node s, Graph g) {
    PriorityQueue unvisited = new PriorityQueue();
    unvisited.addAll(g.allNodes(), ∞)
    unvisited.changePriority(s, 0);
    Map<Node, Integer> distances = new HashMap<>();
    Map<Node, Node> previousNode = new HashMap<>();

    while (!unvisited.isEmpty()) {
        Node n = unvisited.removeMin();
        for (Node i : n.neighbors) {
            if (distances[i] < distances[n] + g.edgeWeight(n, i)) {
                continue;
            } else {
                distances[i] = distances[n] + g.edgeWeight(n, i);
                unvisited.changePriority(i, distances[i]);
                previousNode[i] = n;
            }
        }
    }
}
```
Dijkstra’s Algorithm: Why It Works

❖ Invariants (“something that always holds true”)
  ▪ **distances** contains the best known total distance from s to any v
    - If v has a finite distance, then it must be the optimal distance
  ▪ **unvisit** contains all unvisited vertices, ordered by distance from s
    - Thus, we visit vertices in order of total distance from s

❖ Proof sketch
  ▪ Base case: distances[s] = 0, which is optimal
  ▪ Inductive step: after relaxing all edges from s, let v be the minimum vertex in unvisit. Claim: distances[v] is optimal
    - Proof by contradiction: there is an unvisited path e₁, e₂, ... eₙ whose sum is less than w. However if that were true, we would have visited each of e₁, e₂, ... eₙ already or one of those edges has a negative weight. Therefore our claim is true
Dijkstra’s Algorithm: Demo

- Demo:
  https://docs.google.com/presentation/d/1_bw2z1ggUkquPdhl7gwdVBoTaoJmaZdpkV6MoAgxlJc/pub?start=false&loop=false&delayms=3000
Negative Weights vs Negative Cycles

- Negative weights: Dijkstra’s won’t guarantee correct results
  - But other algorithms might

- Negative cycles: no algorithm can find a finite optimal path
  - Because you can always decrease the path cost by going through the negative cycle a few more times
Dijkstra’s Algorithm: Runtime

- Assuming a binary heap implementation

<table>
<thead>
<tr>
<th># Operations</th>
<th>Cost per operation</th>
<th>Total Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>PQ add</td>
<td>V</td>
<td>O(log V)</td>
</tr>
<tr>
<td>PQ removeMin</td>
<td>V</td>
<td>O(log V)</td>
</tr>
<tr>
<td>PQ changePriority</td>
<td>E</td>
<td>O(log V)</td>
</tr>
</tbody>
</table>

- Runtime: O(V logV + V logV + E logV)
  - Assuming E > V, Dijkstra’s is O(E log V)
tl;dr

❖ Graph implementations have an impact on algorithm runtime

❖ BFS and Dijkstra’s search “shallow” nodes before searching “deep” nodes, whereas DFS searches “deep” nodes first

❖ BFS finds optimal paths in an unweighted graph; Dijkstra’s in a weighted (non-negative) graph