Data Structures in Java

Session 16
Instructor: Bert Huang
http://www.cs.columbia.edu/~bert/courses/3134
Announcements

• Homework 4 due next class
• Midterm grades posted. Avg: 79/90
• Remaining grades:
  • hw4, hw5, hw6 – 25%
  • Final exam – 30%
Today’s Plan

• Graphs
• Topological Sort
• Shortest Path Algorithms: Dijkstra’s
Graphs

Linked List

Tree

Graph
Graph Terminology

- A **graph** is a set of **nodes** and **edges**
  - nodes aka vertices
  - edges aka arcs, links
- Edges exist between pairs of nodes
  - if nodes x and y share an edge, they are **adjacent**
Graph Terminology

• Edges may have \textbf{weights} associated with them
• Edges may be \textbf{directed} or \textbf{undirected}
• A \textbf{path} is a series of adjacent vertices
  • the \textbf{length} of a path is the sum of the edge weights along the path (1 if unweighted)
• A \textbf{cycle} is a path that starts and ends on a node
Graph Properties

- An undirected graph with no cycles is a tree
- A directed graph with no cycles is a special class called a **directed acyclic graph (DAG)**
- In a **connected** graph, a path exists between every pair of vertices
- A **complete** graph has an edge between every pair of vertices
Graph Applications: A few examples

- Computer networks
- The World Wide Web
- Social networks
- Public transportation
- Probabilistic Inference
- Flow Charts
Implementation

- Option 1:
  - Store all nodes in an indexed list
  - Represent edges with adjacency matrix

- Option 2:
  - Explicitly store adjacency lists
Adjacency Matrices

- 2d-array \( A \) of boolean variables
- \( A[i][j] \) is true when node \( i \) is adjacent to node \( j \)
- If graph is undirected, \( A \) is symmetric

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Adjacency Lists

- Each node stores references to its neighbors

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Diagram: A graph with nodes 1, 2, 3, 4, 5 connected as follows:
- 1 connects to 2 and 4
- 2 connects to 1 and 4
- 3 connects to 1 and 4
- 4 connects to 2, 3, and 5
- 5 is connected only to 4
Math Notation for Graphs

- **Set Notation:**
  - $v \in V$ (v is in V)
  - $U \cup V$ (union)
  - $U \cap V$ (intersection)
  - $U \subset V$ (U is a subset of V)

- **Graph Notation:**
  - $G = \{V, E\}$
  - G is the graph
  - V is set of vertices
  - E is set of edges
  - $(v_i, v_j) \in E$
  - $|V| = N = \text{size of V}$
Topological Sort

• Problem definition:

• Given a directed acyclic graph $G$, order the nodes such that for each edge $(v_i, v_j) \in E$, $v_i$ is before $v_j$ in the ordering.

• e.g., scheduling errands when some tasks depend on other tasks being completed.
Topological Sort Ex.

- Buy Groceries
- Look up recipe online
- Mail recipe to Grandma
- Cook Dinner
- Mail Postcard
- Mail Tax Form
- Buy Stamps
- Fix Computer
- Go to ATM
- Taxes
Topological Sort
Naïve Algorithm

- **Degree** means # of edges, **indegree** means # of incoming edges

1. Compute the **indegree** of all nodes
2. Print any node with indegree 0
3. Remove the node we just printed. Go to 1.

- Which nodes’ indegrees change?
Topological Sort
Better Algorithm

1. Compute all indegrees
2. Put all indegree 0 nodes into a Collection
3. Print and remove a node from Collection
4. Decrement indegrees of the node’s neighbors.
5. If any neighbor has indegree 0, place in Collection. Go to 3.
Queue
mail taxes
postcard

<table>
<thead>
<tr>
<th>ATM</th>
<th>comp</th>
<th>groceries</th>
<th>recipe</th>
<th>stamps</th>
<th>taxes</th>
<th>cook</th>
<th>grandma</th>
<th>postcard</th>
<th>mail taxes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2 + 0</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>2 + 1</td>
<td>2 + 0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ATM</td>
<td>comp</td>
<td>groceries</td>
<td>recipe</td>
<td>stamps</td>
<td>taxes</td>
<td>cook</td>
<td>grandma</td>
<td>postcard</td>
<td>mail taxes</td>
</tr>
<tr>
<td>-----</td>
<td>------</td>
<td>-----------</td>
<td>--------</td>
<td>--------</td>
<td>-------</td>
<td>------</td>
<td>---------</td>
<td>---------</td>
<td>-----------</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
Topological Sort
Running time

• Initial indegree computation: $O(|E|)$
• Unless we update indegree as we build graph
• $|V|$ nodes must be enqueued/dequeued
• Dequeue requires operation for outgoing edges
• Each edge is used, but never repeated
• Total running time $O(|V| + |E|)$
Shortest Path

• Given $G = (V, E)$, and a node $s \in V$, find the shortest (weighted) path from $s$ to every other vertex in $G$.

• Motivating example: subway travel
  • Nodes are junctions, transfer locations
  • Edge weights are estimated time of travel
Approximate MTA Express Stop Subgraph

- A few inaccuracies (don’t use this to plan any trips)
Breadth First Search

• Like a level-order traversal
• Find all adjacent nodes (level 1)
• Find new nodes adjacent to level 1 nodes (level 2)
• ... and so on
• We can implement this with a queue
Unweighted Shortest Path Algorithm

- Set node \( s \)'s distance to 0 and enqueue \( s \).
- Then repeat the following:
  - Dequeue node \( v \). For unset neighbor \( u \):
    - set neighbor \( u \)'s distance to \( v \)'s distance + 1
    - mark that we reached \( v \) from \( u \)
    - enqueue \( u \)
<table>
<thead>
<tr>
<th></th>
<th>168th Broad.</th>
<th>145th and 8th</th>
<th>125th and 8th</th>
<th>59th Broad.</th>
<th>Port Auth.</th>
<th>96th Broad.</th>
<th>72nd Broad.</th>
<th>Times Square</th>
<th>Penn Station</th>
<th>86th Lex.</th>
<th>59th Lex.</th>
<th>Grand Central</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dist</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>prev</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The table above represents a graph with nodes labeled with street names and their distances and previous nodes. The table shows distances (dist) and previous nodes (prev). The graph is structured with edges connecting the nodes, and the Times Square node is highlighted in red. The table indicates that the distance from the source to Times Square is 0.
125th and 8th

<table>
<thead>
<tr>
<th></th>
<th>168th Broad.</th>
<th>145th Broad.</th>
<th>125th Broad.</th>
<th>59th Broad.</th>
<th>Port Auth.</th>
<th>59th Lex.</th>
<th>86th Lex.</th>
<th>59th Lex.</th>
<th>Grand Central</th>
</tr>
</thead>
<tbody>
<tr>
<td>dist</td>
<td></td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>168th Broad.</td>
<td>145th Broad.</td>
<td>125th Broad.</td>
<td>59th Broad.</td>
<td>Port Auth.</td>
<td>96th Broad.</td>
<td>72nd Broad.</td>
<td>Times Square</td>
<td>Penn Station</td>
</tr>
<tr>
<td>-------</td>
<td>--------------</td>
<td>--------------</td>
<td>--------------</td>
<td>-------------</td>
<td>------------</td>
<td>-------------</td>
<td>-------------</td>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>dist</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Weighted Shortest Path

- The problem becomes more difficult when edges have different weights
- Weights represent different costs on using that edge
- Standard algorithm is Dijkstra’s Algorithm
Dijkstra’s Algorithm

- Keep distance overestimates $D(v)$ for each node $v$ (all non-source nodes are initially infinite)
- 1. Choose node $v$ with smallest *unknown* distance
- 2. Declare that $v$’s shortest distance is *known*
- 3. Update distance estimates for neighbors
Updating Distances

• For each of v’s neighbors, w,
• if min(D(v)+ weight(v,w), D(w))
  • i.e., update D(w) if the path going through v is cheaper than the best path so far to w
<table>
<thead>
<tr>
<th></th>
<th>59th Broad</th>
<th>Port Auth.</th>
<th>72nd Broad</th>
<th>Times Sq.</th>
<th>Penn St.</th>
</tr>
</thead>
<tbody>
<tr>
<td>inf</td>
<td>6</td>
<td>inf</td>
<td>?</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>?</td>
<td>Penn St.?</td>
<td>?</td>
<td>Penn St.</td>
<td>home</td>
<td></td>
</tr>
</tbody>
</table>
2 + 12 = 14
6
2 + 4 = 6
2
0
Times Sq?
Penn St?
Times Sq?
Penn St.
home
<table>
<thead>
<tr>
<th></th>
<th>59th Broad.</th>
<th>Port Auth.</th>
<th>72nd Broad</th>
<th>Times Sq.</th>
<th>Penn St.</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 + 6 = 11</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Port Auth?</td>
<td>Penn St.</td>
<td></td>
<td>Times Sq?</td>
<td>Penn St.</td>
<td>home</td>
</tr>
</tbody>
</table>
Dijkstra’s Algorithm Analysis

• First, convince ourselves that the algorithm works.
• At each stage, we have a set of nodes whose shortest paths we know.
• In the base case, the set is the source node.
• Inductive step: if we have a correct set, is greedily adding the shortest neighbor correct?
Proof by Contradiction (Sketch)

• Contradiction: Dijkstra’s finds a shortest path to node $w$ through $v$, but there exists an even shorter path.

• This shorter path must pass from inside our known set to outside.

• Call the 1st node in cheaper path outside our set $u$.

• The path to $u$ must be shorter than the path to $w$.

• But then we would have chosen $u$ instead.
Computational Cost

- Keep a priority queue of all unknown nodes
- Each stage requires a `deleteMin`, and then some `decreaseKeys` (the # of neighbors of node)
- We call `decreaseKey` once per edge, we call `deleteMin` once per vertex
- Both operations are $O(\log |V|)$
- Total cost: $O(|E| \log |V| + |V| \log |V|) = O(|E| \log |V|)$
Reading

- Weiss Section 9.1-9.3 (today’s material)