Data Structures in Java

Session 17
Instructor: Bert Huang
http://www.cs.columbia.edu/~bert/courses/3134
Announcements

- Homework 4 due
- Homework 5 posted
- All-pairs shortest paths
Review

- Graphs
- Topological Sort
  - Print out a node with indegree 0,
  - update indegrees
Today’s Plan

- Shortest Path algorithms
- Breadth first search
- Dijkstra’s Algorithm
- All-Pairs Shortest Path
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’ 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
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 \( \min(D(v) + \text{weight}(v, w), D(w)) \)
• i.e., update \( D(w) \) if the path going through \( v \) is cheaper than the best path so far to \( w \)
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 1\(^{st}\) 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

• If the graph is dense, we scan the vertices to find the minimum edge $O(V)$
  
• This happens $|V|$ times

• We also update the distances once per edge, $O(|E|)$

• Thus, total running time is $O(|E| + |V|^2)$
Computational Cost (sparse)

• 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: $O(|E| \log |V| + |V| \log |V|) = O(|E| \log |V|)$
All Pairs Shortest Path

- Dijkstra’s Algorithm finds shortest paths from one node to all other nodes
- What about computing shortest paths for all pairs of nodes?
- We can run Dijkstra’s |V| times. Total cost: $O(|V|^3)$
- Floyd-Warshall algorithm is often faster in practice (though same asymptotic time)
Recursive Motivation

- Consider the set of numbered nodes 1 through k
- The shortest path between any node i and j using only nodes in the set \{1, ..., k\} is the minimum of
  - shortest path from i to j using nodes \{1, ..., k-1\}
  - shortest path from i to j using node k
- dist(i, j, k) = min(dist(i, j, k-1), dist(i, k, k-1) + dist(k, j, k-1))
Dynamic Programming

• Instead of repeatedly computing recursive calls, store lookup table

• To compute dist(i,j,k) for any i,j, we only need to look up dist(-,-, k-1)
  • but never k-2, k-3, etc.

• We can incrementally compute the path matrix for k=0, then use it to compute for k=1, then k=2...
Floyd-Warshall Code

- Initialize \( d = \) weight matrix
- for (k=0; k<N; k++)
  - for (i=0; i<N; i++)
    - for (j=0; j<N; j++)
      - if \( d[i][j] > d[i][k] + d[k][j] \)
        - \( d[i][j] = d[i][k] + d[k][j] \);
- Additionally, we can store the actual path by keeping a “midpoint” matrix
Midpoint Matrix

- We can store the $N^2$ paths efficiently with a midpoint matrix:

  $$\text{path}(i,j) = \text{path}(i, \text{midpoint}[i][j]) + \text{path}(\text{midpoint}[i][j], j)$$

- We only need a N\times N matrix to store all the paths
All Pairs Shortest Path Example

\[
\begin{array}{cccc}
  & 1 & 2 & 3 & 4 \\
1 & - & 4 & - & - \\
2 & - & - & 3 & 1 \\
3 & 2 & - & - & 4 \\
4 & - & - & 2 & - \\
\end{array}
\]

\[k=0\]
Transitive Closure

• For any nodes i, j, is there a path from i to j?
• Instead of computing shortest paths, just compute Boolean if a path exists
• path(i,j,k) = path(i,j,k-1) OR
  path(i,k,k-1) AND path(k,j,k-1)
• Transitive closure can tell you whether a graph is connected
Reading

- Weiss Section 9.1-9.3,
- Weiss Section 10.3.4
  (All-Pairs Shortest Path)