• Homework 6
  • Due Friday at 8pm
• Google CodeJam
  • Registration open until end of online qualification round
  • Online qualification round begins on April 8 and runs for 27 hours
  • Bonus points!
    • 2 pts if you pass online qualification round
    • 3 pts if you pass online round 1
For directed, weighted graphs:
  * Without negative weights

Dijkstra’s algorithm:
  * Recall BFS: instead of enqueuing just neighbors, enqueue (total weight to source, neighbor), with the priority queue sorting by weight
  * Minimum distance from source to other vertices stored in array
    * Array updated as smaller-weight paths are found
    * Initialized with inf
  * \(O((|V| + |E|) \log |V|)\)
Question: Shortest paths from 2 to all other nodes?

Priority Queue (distance, node index)
\{(0, 2)\}

Start from node 2

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
</table>
| d[i] | INF | INF | 0 | INF | INF | Distance table
Question: Shortest paths from 2 to all other nodes?

Priority Queue (distance, node index)
\{(0, 2)\}
\{(2, 1), (6, 0), (7, 3)\}

Distance table

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>d[i]</td>
<td>6</td>
<td>2</td>
<td>0</td>
<td>7</td>
<td>INF</td>
</tr>
</tbody>
</table>

Add all unvisited nodes from node 2 to the priority queue.
The PQ sorts the distances so the “next closest” node floats to the top.
Right now the closest node is 1, followed by 0, then 3.
Question: Shortest paths from 2 to all other nodes?

Priority Queue (distance, node index)
{(0, 2)}
{(2, 1), (6, 0), (7, 3)}
{(5, 3), (6, 0), (7, 3), (8, 4)}

Distance table

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>d[i]</td>
<td>6</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

Poll from the PQ to get node 1.
Add all neighboring nodes to node 1 that haven't been polled yet.
BUT be sure to add all nodes that may already be in the queue with longer distances
– there may be a shorter way to reach them
Question: Shortest paths from 2 to all other nodes?

Priority Queue (distance, node index)

\[
\begin{align*}
&\{(0, 2)\} \\
&\{(2, 1), (6, 0), (7, 3)\} \\
&\{(5, 3), (6, 0), (7, 3), (8, 4)\} \\
&\{(6, 0), (7, 3), (8, 4)\}
\end{align*}
\]

Distance table

\[
\begin{array}{cccccc}
i  & 0 & 1 & 2 & 3 & 4 \\
d[i] & 6 & 2 & 0 & 5 & 8 \\
\end{array}
\]

Poll from the PQ to get node 3.
Since we know there is a faster way to get node 4, don't bother adding node 4 to PQ
Dijkstra's Algorithm: Example

Question: Shortest paths from 2 to all other nodes?

Priority Queue (distance, node index)

\[
\{ (0, 2) \} \\
\{ (2, 1), (6, 0), (7, 3) \} \\
\{ (5, 3), (6, 0), (7, 3), (8, 4) \} \\
\{ (6, 0), (7, 3), (8, 4) \} \\
\{ (7, 3), (7, 4), (8, 4) \}
\]

Distance table

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>d[i]</td>
<td>6</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

Poll from the PQ to get node 0.
Since we know there is a faster way to get node 4, don't bother adding node 4 to PQ
Question: Shortest paths from 2 to all other nodes?

Priority Queue (distance, node index)
{(0, 2)}
{(2, 1), (6, 0), (7, 3)}
{(5, 3), (6, 0), (7, 3), (8, 4)}
{(6, 0), (7, 3), (8, 4)}
{(7, 3), (7, 4), (8, 4)}
{(7, 4), (8, 4)}

Now the (7, 3) state is ignored because it's been determined that 7 is a longer path than another existing path to node 3.
Question: Shortest paths from 2 to all other nodes?

Priority Queue (distance, node index)

\[
\begin{align*}
&\{(0, 2)\} \\
&\{(2, 4), (6, 0), (7, 3)\} \\
&\{(5, 3), (6, 0), (7, 3), (8, 4)\} \\
&\{(6, 0), (7, 3), (8, 4)\} \\
&\{(7, 4), (7, 4), (8, 4)\} \\
&\{(7, 4), (8, 4)\} \\
&\{(8, 4)\}
\end{align*}
\]

Distance table:

\[
\begin{array}{cccccc}
i & 0 & 1 & 2 & 3 & 4 \\
d[i] & 6 & 2 & 0 & 5 & 7
\end{array}
\]

Nowhere to go, so nothing is added to the PQ.
Dijkstra’s Algorithm: Example

Question: Shortest paths from 2 to all other nodes?

Priority Queue (distance, node index)

\{(0, 2)\}
\{(2, 1), (6, 0), (7, 3)\}
\{(6, 3), (6, 0), (7, 3), (8, 4)\}
\{(6, 0), (7, 3), (8, 4)\}
\{(7, 3), (7, 4), (8, 4)\}
\{(7, 4), (8, 4)\}
\{(8, 4)\}
\}

Distance table

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>d[i]</td>
<td>6</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

State (8, 4) is ignored because 8 > 7
**Exercise:** Will Dijkstra’s work for undirected, unweighted graphs?

**Exercise:** How do you find the single-source longest path?
• Dijsktra’s will not work for negative weights
  • Greedy nature (i.e., the priority queue)
  • Infinite loop if there’s a cycle
• What to do in case of negative weights?
  • Bellman-Ford: relax all E edges V-1 times
    • Arbitrary order
    • If you can continue to relax edges after V-1 times, then a negative cycle exists
  • Running time: $O(|E| \times |V|)$
Problem: **UVA 104: Arbitrage**

**Summary**: Take advantage of currency fluctuations to make money.

**Input**: Table of exchange rates

**Output**: Shortest sequence of currencies to buy that will yield a profit of more than 1%
• Dijkstra’s and Bellman-Ford give you single-source shortest path, what if you want to find all-pairs shortest path?
• What happens if you want to find the shortest distance between all pairs of nodes?
  • On a weighted, connected graph, use Floyd Warshall algorithm
  • Implement in ~4 lines of code
  • $O(V^3)$ instead of $V$ Dijkstra's algorithm, which would be $O(V^3 \log V)$
  • Dynamic programming
// inside int main()
// precondition: m[i][j] contains the weight of edge (i, j)
// or INF if there is no such edge
// (m is an adjacency matrix)

for (int k = 0; k < V; k++)
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            m[i][j] = min(m[i][j], m[i][k] + m[k][j]);

// common error: remember that loop order is k->i->j
• Intuition: gradually allow the usage of intermediate vertices [1..k]
• E.g., suppose we’re looking for the shortest path from 3 to 4:

![Diagram of a graph with vertices and edges labeled with numbers.]

The current content of Adjacency Matrix D at k = -1:

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

sp(3, 2, -1) = 3  sp(2, 4, -1) = 1  sp(3, 4, -1) = 5

We will monitor these two values.
• k=0 allows us to find shorter paths from $3 \rightarrow 0 \rightarrow 1$, $3 \rightarrow 0 \rightarrow 2$, $3 \rightarrow 0 \rightarrow 4$
• $k=2$ allows us to find shorter paths from $0 \rightarrow 2 \rightarrow 4$, $3 \rightarrow 2 \rightarrow 4$
• Recall that $3 \rightarrow 2 \rightarrow 4$ shortest path is actually $3 \rightarrow 0 \rightarrow 2 \rightarrow 4$
Exercise: The diameter of a graph is the longest distance between any pair of vertices. Explain how to find the diameter of a graph.

Exercise: Explain how to find the strongly connected components of a graph in $O(V^3)$ time.
We are going to build a DP single source shortest path algorithm as follows (weighted directed graph with negative edges). The dp array \texttt{double[]} \texttt{dists} will store the length of the shortest path from the starting node \texttt{v} to vertex \texttt{i} using any walk of length \texttt{k} or less. The algorithm iterates \texttt{k} = 1, 2, \ldots \text{ updating \texttt{dists} at each step. Answer the following questions:}

(a) How should we initialize the array \texttt{dists}?

(b) Explain how to update \texttt{dists} during each iteration.

(c) When can we stop iterating through \texttt{k}-values?

(d) How can we use the above algorithm to find negative cycles?
We are going to build a DP all pairs shortest path algorithm as follows (weighted directed graph with negative edges). The dp array `double[][] dists` will store the length of the shortest path from vertex `i` to vertex `j` only using vertices numbered less than `k`. The algorithm iterates `k = 1, 2, ...` updating `dists` at each step. Answer the following questions:

(a) How should we initialize the array `dists`?
(b) Explain how to update `dists` during each iteration.
(c) When can we stop iterating through `k`-values?
(d) How can we use the above algorithm to find negative cycles?
• A graph is **Eulerian** if it has an Eulerian circuit
• An **Eulerian circuit** is a walk that uses each edge once and ends where it starts
• A graph is Eulerian iff it is connected and all vertices have even degree
• Proof:
  • In an Eulerian circuit, every time you enter a vertex via an edge you must leave via another untraversed edge
  • Consider max length walk that doesn’t reuse edges starting at $v_s$ and ending at $v_e$:
    • Since walk is max length, $v_s = v_e$
    • If not every edge is used along the walk, then there is a vertex outside the walk connected to a vertex in the walk, so we can add it and thus contradict our premise
Consider a rectangle $R$ in the plane with corners $(0, 0)$ and $(a, b)$ with $a$, $b$ positive real numbers. Assume $R$ can be tiled with rectangles (may be all distinct) such that each tile has at least 1 integer side. Prove $R$ has an integer side.
• Create a graph using tile corners as vertices
  • Denote each tile as an H-tile or a V-tile if its horizontal or vertical edge is an integer (if both, just choose one label)
• For each H-tile, draw an edge along its horizontal sides, and do the same for the vertical edges of the V-tiles
  • Each vertex is either touching 2 or 4 rectangles, unless it is an outer corner touching only one.
• The resulting graph has all even degree, except the outer corners, and hence there is a path from one outer corner to another.
• Trees are undirected acyclic connected graphs

• **Definition**: the least common ancestor (LCA) of two nodes is the deepest (w.r.t the root) ancestor of both nodes

• Algorithm for finding LCA:
  1. Build a list of nodes in the graph using depth first traversal
  2. Add a node to the list every time it is visited
  3. For each node, also store its depth **for each list entry**
  4. To find the LCA, find the range minimum on the list of depths between two nodes
     • Each query is $O(\lg n)$ with segment trees
     • Since the list is static, you can precompute $O(n^2)$ and query $O(1)$
Competitive Programming 4.1-4.6

More solutions for the tiling rectangles problem:
http://www.inference.phy.cam.ac.uk/mackay/rectangles/