Shortest Paths

Weighted Graphs

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge.
- Edge weights may represent distances, costs, etc.
- Example: In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports.
Shortest Paths

- Given a weighted graph and two vertices \( u \) and \( v \), we want to find a path of minimum total weight between \( u \) and \( v \).
  - Length of a path is the sum of the weights of its edges.
- Example:
  - Shortest path between Providence and Honolulu
- Applications
  - Internet packet routing
  - Flight reservations
  - Driving directions

Shortest Path Properties

Property 1:
- A subpath of a shortest path is itself a shortest path

Property 2:
- There is a tree of shortest paths from a start vertex to all the other vertices

Example:
- Tree of shortest paths from Providence
Dijkstra’s Algorithm

- The distance of a vertex \( v \) from a vertex \( s \) is the length of a shortest path between \( s \) and \( v \).
- Dijkstra’s algorithm computes the distances of all the vertices from a given start vertex \( s \).
- Assumptions:
  - the graph is connected
  - the edges are undirected
  - the edge weights are nonnegative
- We grow a “cloud” of vertices, beginning with \( s \) and eventually covering all the vertices.
- We store with each vertex \( v \) a label \( D[v] \) representing the distance of \( v \) from \( s \) in the subgraph consisting of the cloud and its adjacent vertices.
- At each step
  - We add to the cloud the vertex \( u \) outside the cloud with the smallest distance label, \( D[u] \).
  - We update the labels of the vertices adjacent to \( u \).

Edge Relaxation

- \( D[v] \) = the shortest distance of \( v \) from \( s \) found so far.
- Consider an edge \( e = (u, z) \) such that
  - \( u \) is the vertex most recently added to the cloud
  - \( z \) is not in the cloud
- The relaxation of edge \( e \) updates distance \( d(z) \) as follows:
  \[ D[z] \leftarrow \min\{D[z], D[u] + \text{weight}(e)\} \]
Dijkstra’s Algorithm: Details

- **Input**: A weighted directed graph \( G=(V, E) \), where \( V=\{1, 2, \ldots, n\} \);
- **Output**: The distance from vertex 1 to every other vertex in \( G \);

1. \( X=\{1\}; \ Y\leftarrow V-\{1\}; \ D[1]\leftarrow 0; \)
2. for \( y\leftarrow 2 \) to \( n \)
3. if \( (y \) is adjacent to \( 1) \) \{ \( D[y]\leftarrow \text{length}[1,y]; \ p[y] \leftarrow 1 \) \}
4. else \( D[y]\leftarrow \infty \);
5. for \( j\leftarrow 2 \) to \( n \)
6. Let \( y\in Y \) be such that \( D[y] \) is minimum;
7. \( X\leftarrow X\cup \{y\}; \) // add vertex \( y \) to \( X \)
8. \( Y\leftarrow Y - \{y\}; \) //delete vertex \( y \) from \( Y \)
9. for each edge \( (y, w) \) // edge relaxation
10. if \( (w \in Y \) and \( D[y]+\text{length}[y, w]<D[w]) \)
11. \{ \( D[w]\leftarrow D[y]+\text{length}[y, w]; \ p[w] \leftarrow y \) \}

Example
Example (cont.)

Analysis of Dijkstra’s Algorithm

- Graph operations
  - We find all the incident edges once for each vertex
- Label operations
  - We set/get the distance and locator labels of vertex \( z \) \( O(\deg(z)) \) times
  - Setting/getting a label takes \( O(1) \) time
- Priority queue operations
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes \( O(\log n) \) time
  - The key of a vertex in the priority queue is modified at most \( \deg(w) \) times, where each key change takes \( O(\log n) \) time
- Dijkstra’s algorithm runs in \( O((n + m) \log n) \) time provided the graph is represented by the adjacency list/map structure
  - Recall that \( \sum \deg(v) = 2m \)
- The running time can also be expressed as \( O(m \log n) \) since the graph is connected
Possible Quiz Question

Find the shortest paths from A to all other vertices and draw the tree found by Dijkstra’s Algorithm.

Why Dijkstra’s Algorithm Works

- Dijkstra’s algorithm is based on the greedy method. It adds vertices by increasing distance.
  - Suppose it didn’t find all shortest distances. Let $w$ be the first wrong vertex the algorithm processed.
  - When the previous node, $u$, on the true shortest path was considered, its distance was correct.
  - But the edge $(u,w)$ was relaxed at that time!
  - Thus, so long as $D[w]>D[u]$, $w$’s distance cannot be wrong. That is, there is no wrong vertex $(u,w) = (D,F)$ in this example.
Why It Doesn’t Work for Negative-Weight Edges

- Dijkstra’s algorithm is based on the greedy method. It adds vertices by increasing distance.
  - If a node with a negative incident edge were to be added late to the cloud, it could mess up distances for vertices already in the cloud.
  - Example: The shortest path from A to C is through B: the distance is $3 + (-2) = 1$.

The All-Pairs Shortest Path Problem

- Let $G = (V, E)$ be a directed graph in which each edge $(i, j)$ has a non-negative length $w[i, j]$. If there is no edge from vertex $i$ to vertex $j$, then $w[i, j] = \infty$.
- The problem is to find the minimal distance from each vertex to all other vertices, where the distance from vertex $x$ to vertex $y$ is the sum of the edge lengths in a path from $x$ to $y$. 
The All-Pairs Shortest Path Problem

Example:

Weight:

<table>
<thead>
<tr>
<th></th>
<th>w</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>2</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>8</td>
<td>0</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Distance:

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>7</td>
<td>0</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Design a Dynamic Programming Solution

- How are the subproblems formulated?
- Where are the solutions stored?
- How are the base values computed?
- How do we compute each entry from other entries in the table?
- What is the order in which we fill in the table?
Two DP algorithms for All-pairs shortest paths

- Both are correct. Both produce correct values for all-pairs shortest paths.
- The difference is the subproblem formulation, and hence in the running time.
- Be prepared to provide one or both of these algorithms, and to be able to apply it to an input (on some exam, for example).

Dynamic Programming

First attempt: let \{1,2,...,n\} denote the set of vertices.

Subproblem formulation:

\[
M[i,j,k] = \text{min length of any path from i to j that uses at most } k \text{ edges.}
\]

All paths have at most n-1 edges, so 1 \leq k \leq n-1.

When k=1, \(M[i,j,1] = w[i,j]\), the edge weight from i to j.

Minimum paths from i to j are found in \(M[i,j,n-1]\)

- Question: How to set \(M[i,j,k]\) from other entries?
How to set $M[i,j,k]$ from other entries, for $k>1$?

Consider a *minimum weight* path from $i$ to $j$ that has at most $k$ edges.

- **Case 1**: The minimum weight path has at most $k-1$ edges.
  - $M[i,j,k] = M[i,j,k-1]$  

- **Case 2**: The minimum weight path has exactly $k$ edges.
  - $M[i,j,k] = \min\{ M[i,x,k-1] + w(x,j) : x \in V \}$

Combining the two cases:

$$M[i,j,k] = \min\{\min\{M[i,x,k-1] + w(x,j) : x \in V\}, M[i,j,k-1]\}$$

Finishing the design

- How are the subproblems defined?
  - Subproblem formulation:
    - $M[i,j,k] = \min$ length of any path from $i$ to $j$ that uses at most $k$ edges.

- Where is the answer stored?
  - Minimum paths from $i$ to $j$ are found in $M[i,j,n-1]$.

- How are the base values computed?
  - When $k=1$, $M[i,j,1] = w[i,j]$, the edge weight from $i$ to $j$.

- How do we compute each entry from other entries?
  - $M[i,j,k] = \min\{\min\{M[i,x,k-1] + w(x,j) : x \in V\}, M[i,j,k-1]\}$

- What is the order in which we fill in the matrix?
  - For $k$ from $1$ to $n-1$, compute $M[i,j,k]$.

- Running time?
Pseudo-Code and Complexity Analysis

for j = 1 to n  for i = 1 to n
    M[i,j,1] = w[i,j];
for k = 2 to n-1
    for j = 1 to n
        for i = 1 to n {
            minx = M[i,j,k-1];
            for x = 1 to n
                if (minx > M[i,x,k-1] + w(x,j)) minx = M[i,x,k-1] + w(x,j);
            M[i,j,k] = minx;
        }

- How many entries do we need to compute?  $O(n^3)$
  1 ≤ i ≤ n; 1 ≤ j ≤ n; 1 ≤ k ≤ n-1
- How much time does it take to compute each entry?  $O(n)$
- Total time: $O(n^4)$  Total space: $O(n^3)$ (or $O(n^2)$)

Next DP approach: Marshall’s Algorithm

- Try a new subproblem formulation!

- $Q[i,j,k] = \text{minimum weight of any path from } i \text{ to } j \text{ that uses internal vertices drawn from } \{1,2,...,k\}$. 
Designing a DP solution

- How are the subproblems formulated?
  - Q[i,j,k] = minimum weight of any path from i to j that uses internal vertices (other than i and j) drawn from \{1,2,...,k\}.

- Where is the answer stored?
  - Q[i,j,n] stores the min length from i to j.

- How are the base values computed?
  - Base cases: Q[i,j,0] = w[i,j] for all i,j

- How do we compute each entry from other entries?

- What is the order in which we fill in the matrix?

Solving subproblems

- Q[i,j,k] = minimum weight of any path from i to j that uses internal vertices drawn from \{1,2,...,k\}.

- Such minimum cost path either includes vertex k or does not include vertex k.

- If the minimum cost path P includes vertex k, then you can divide P into the path P_1 from i to k, and P_2 from k to j.

- What is the weight of P_1? Q[i,k,k-1] (why?).

- What is the weight of P_2? Q[k,j,k-1] (why?).

- Thus the weight of P is Q[i,k,k-1] + Q[k,j,k-1].
Marshall’s Algorithm

\[
\begin{align*}
\text{for } j &= 1 \text{ to } n \\
\text{for } i &= 1 \text{ to } n \\
Q[i,j,0] &= w[i,j] \\
\text{for } k &= 1 \text{ to } n \\
\text{for } j &= 1 \text{ to } n \\
\text{for } i &= 1 \text{ to } n \\
Q[i,j,k] &= \min\{Q[i,j,k-1], \ Q[i,k,k-1] + Q[k,j,k-1]\}
\end{align*}
\]

- Each entry only takes \(O(1)\) time to compute
- There are \(O(n^3)\) entries
- Hence, \(O(n^3)\) time.
- Total space: \(O(n^3)\) (or \(O(n^2)\))

Reusing the space

// Use R[i,j] for \(Q[i,j,0], \ Q[i,j,1], \ldots, \ Q[i,j,n]\).

\[
\begin{align*}
\text{for } j &= 1 \text{ to } n \\
\text{for } i &= 1 \text{ to } n \\
R[i,j] &= w[i,j]; \\
\text{for } k &= 1 \text{ to } n \\
\text{for } j &= 1 \text{ to } n \\
\text{for } i &= 1 \text{ to } n \\
R[i,j] &= \min\{R[i,j], R[i,k] + R[k,j]\}
\end{align*}
\]

Claim: For any \(k\), min path of \(i\) to \(j\) \(\leq R[i,j] \leq Q[i,j,k]\).
How to check negative cycles

```c
// Use R[i,j] for Q[i,j,0], Q[i,j,1], ..., Q[i,j,n].
for j = 1 to n
    for i = 1 to n
        R[i,j] = w[i,j];
    for k= 1 to n
        for j = 1 to n
            for i = 1 to n
                R[i,j] = min{R[i,j], R[i,k] + R[k,j]};
    for i = 1 to n
        if (R[i,i] < 0) print("There is a negative cycle");
```

How to compute transitive closure

- The relation $R^* = R_1 \cup R_2 \cup R_3 \cup \ldots \cup R_{n-1}$, where $n$ is the number of nodes, is called the transitive closure of $R$.
- To decide if $(a, b)$ in $R^*$, we need to decide if there is a path from $a$ to $b$ in $G = (A, R)$.

```c
// Post: T is the transitive closure of R
for j = 1 to n
    for i = 1 to n
        T[i,j] = R[i,j]; // R[..] is 0/1 incidence matrix for relation R,
    for k= 1 to n
        for j = 1 to n
            for i = 1 to n
                T[i,j] = T[i,j] || (T[i,k] && T[k,j]);
```