Outline and Reading

- Weighted graphs (§12.1)
  - Shortest path problem
  - Shortest path properties
- Dijkstra’s algorithm (§12.6.1)
  - Algorithm
  - Edge relaxation
- The Bellman-Ford algorithm
- Shortest paths in DAGs
- All-pairs 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 Path Problem

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

- 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:
  \[
  d(z) \leftarrow \min(d(z), d(u) + \text{weight}(e))
  \]

Example (cont.)

Dijkstra’s Algorithm

- A priority queue stores the vertices outside the cloud
  - Key: distance
  - Element: vertex
- Locator-based methods
  - \( \text{insert}(s, r) \) returns a locator
  - \( \text{replaceKey}(L, s) \) changes the key of an item
- We store two labels with each vertex:
  - distance \((d(v))\) label
  - locator in priority queue

Analysis

- Graph operations
  - Method \( \text{incidentEdges} \) is called once for each vertex
- Label operations
  - We set/get the distance and locator labels of vertex \( v \) \( O(\log(n)) \) 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 \( \text{deg}(v) \) 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 structure
  - Recall that \( \Sigma_s \text{deg}(r) = 2m \)
  - The running time can also be expressed as \( O(m \log n) \) since the graph is connected

Extension

- Using the template method pattern, we can extend Dijkstra’s algorithm to return a tree of shortest paths from the start vertex to all other vertices
- We store with each vertex a third label:
  - parent edge in the shortest path tree
  - In the edge relaxation step, we update the parent label
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 F be the first wrong vertex the algorithm processed.
- When the previous node, D, on the true shortest path was considered, its distance was correct.
- But the edge (D,F) was relaxed at that time!
- Thus, so long as \( d(F) > d(D) \), F’s distance cannot be wrong. That is, there is no wrong vertex.

Why It Doesn’t Work for Negative-Weight Edges

- 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.
- Dijkstra’s algorithm is based on the greedy method. It adds vertices by increasing distance.
- C’s true distance is 1, but it is already in the cloud with \( d(C) = 5! \).

Bellman-Ford Algorithm

- Works even with negative-weight edges
- Must assume directed edges (for otherwise we would have negative-weight cycles)
- Iteration i finds all shortest paths that use i edges.
- Running time: \( O(nm) \).
- Can be extended to detect a negative-weight cycle if it exists

Bellman-Ford Example

Nodes are labeled with their \( d(v) \) values

DAG-based Algorithm

- Works even with negative-weight edges
- Uses topological order
- Doesn’t use any fancy data structures
- Is much faster than Dijkstra’s algorithm
- Running time: \( O(n+m) \).

DAG Example

Nodes are labeled with their \( d(v) \) values
All-Pairs Shortest Paths

- Find the distance between every pair of vertices in a weighted directed graph G.
- We can make n calls to Dijkstra's algorithm (if no negative edges), which takes O(nmlog n) time.
- Likewise, n calls to Bellman-Ford would take O(n^2m) time.
- We can achieve O(n^3) time using dynamic programming (similar to the Floyd-Warshall algorithm).

Algorithm `AllPair(G)`: assumes vertices 1,...,n
for all vertex pairs (i,j)
if i = j
`D[0][i,j]` ← 0
else if (i,j) is an edge in G
`D[0][i,j]` ← weight of edge (i,j)
else
`D[0][i,j]` ← +∞
for k ← 1 to n do
for i ← 1 to n do
for j ← 1 to n do
`D[k][i,j]` ← min{`D[k-1][i,j]`, `D[k-1][i,k]` + `D[k-1][k,j]`}
return `D[n]`

Uses only vertices numbered 1,...,k-1
Uses only vertices numbered 1,...,k-1 (compute weight of this edge)