Shortest Paths
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 (§ 12.6)

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 (§ 12.6.1)

- 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

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

© 2004 Goodrich, Tamassia
Example (cont.)
Dijkstra’s Algorithm

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

Algorithm \texttt{DijkstraDistances}(G, s)

\begin{algorithmic}
  \STATE $Q \leftarrow \text{new priority queue}$
  \FORALL { $v \in G\text{.vertices}()$ }
    \IF { $v = s$ }
      \STATE \texttt{setDistance}(v, 0)
    \ELSE
      \STATE \texttt{setDistance}(v, $\infty$)
    \ENDIF
  \ENDFOR
  \STATE $l \leftarrow Q.insert(\text{getDistance}(v), v)$
  \WHILE { $\neg Q.isEmpty()$ }
    \STATE $u \leftarrow Q.removeMin()$
    \FORALL { $e \in G\text{.incidentEdges}(u)$ }
      \STATE { relax edge } $e$
    \FORALL { $z \leftarrow G\text{.opposite}(u, e)$ }
      \STATE $r \leftarrow \text{getDistance}(u) + \text{weight}(e)$
      \IF { $r < \text{getDistance}(z)$ }
        \STATE \texttt{setDistance}(z, r)
        \STATE \texttt{Q.replaceKey}(z, r)
      \ENDIF
    \ENDFOR
  \ENDWHILE
\end{algorithmic}
Analysis of Dijkstra’s Algorithm

- **Graph operations**
  - Method incidentEdges is called 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 structure
  - Recall that \( \sum_v \deg(v) = 2m \)

- The running time can also be expressed as \( O(m \log n) \) since the graph is connected
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.

**Algorithm** $DijkstraShortestPathsTree(G, s)$

\[\text{...}\]

\[\text{for all } v \in G.\text{vertices}()\]
\[\text{...}\]
\[\text{setParent}(v, \emptyset)\]
\[\text{...}\]

\[\text{for all } e \in G.\text{incidentEdges}(u)\]
\[\{ \text{relax edge } e \}\]
\[z \leftarrow G.\text{opposite}(u,e)\]
\[r \leftarrow \text{getDistance}(u) + \text{weight}(e)\]
\[\text{if } r < \text{getDistance}(z)\]
\[\text{setDistance}(z,r)\]
\[\text{setParent}(z,e)\]
\[Q.\text{replaceKey}(\text{getLocator}(z), r)\]
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 \(d(F) \geq d(D)\), F’s distance cannot be wrong. That is, there is no wrong vertex.
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.

C’s true distance is 1, but it is already in the cloud with $d(C)=5$!