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 \( u \)
Edges 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
Example (cont.)
Dijkstra’s Algorithm

**Algorithm** ShortestPath\((G,s)\):

**Input:** A weighted graph \(G\) with nonnegative edge weights, and a distinguished vertex \(s\) of \(G\).

**Output:** The length of a shortest path from \(s\) to \(v\) for each vertex \(v\) of \(G\).

Initialize \(D[s] = 0\) and \(D[v] = \infty\) for each vertex \(v \neq s\).

Let a priority queue \(Q\) contain all the vertices of \(G\) using the \(D\) labels as keys.

while \(Q\) is not empty do

\{pull a new vertex \(u\) into the cloud\}

\(u = \text{value returned by } Q\text{.remove\_min()}

for each vertex \(v\) adjacent to \(u\) such that \(v\) is in \(Q\) do

\{perform the relaxation procedure on edge \((u,v)\}\}

if \(D[u] + w(u,v) < D[v]\) then

\(D[v] = D[u] + w(u,v)\)

Change to \(D[v]\) the key of vertex \(v\) in \(Q\).

return the label \(D[v]\) of each vertex \(v\)
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 $z$ $O(\text{deg}(z))$ 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 $\text{deg}(w)$ 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 $\sum_v \text{deg}(v) = 2m$

  - The running time can also be expressed as $O(m \log n)$ since the graph is connected
```
def shortest_path_lengths(g, src):
    '''Compute shortest-path distances from src to reachable vertices of g.

    Graph g can be undirected or directed, but must be weighted such that
    e.weight() returns a numeric weight for each edge e.

    Return dictionary mapping each reachable vertex to its distance from src.
    '''
    d = {}  # d[v] is upper bound from s to v
    cloud = {}  # map reachable v to its d[v] value
    pq = AdaptableHeapPriorityQueue()  # vertex v will have key d[v]
    pqlocator = {}  # map from vertex to its pq locator

    # for each vertex v of the graph, add an entry to the priority queue, with
    # the source having distance 0 and all others having infinite distance
    for v in g.vertices:
        if v is src:
            d[v] = 0
        else:
            d[v] = float('inf')  # syntax for positive infinity
            pqlocator[v] = pq.add(d[v], v)  # save locator for future updates

    while not pq.is_empty:
        key, u = pq.remove_min()
        cloud[u] = key  # its correct d[u] value
        del pqlocator[u]  # u is no longer in pq
        for e in g.incident_edges(u):  # outgoing edges (u,v)
            v = e.opposite(u)
            if v not in cloud:
                # perform relaxation step on edge (u,v)
                wgt = e.weight()
                if d[u] + wgt < d[v]:  # better path to v?
                    d[v] = d[u] + wgt  # update the distance
                    pq.update(pqlocator[v], d[v], v)  # update the pq entry

    return cloud  # only includes reachable vertices
```
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 $d(C)=5!$