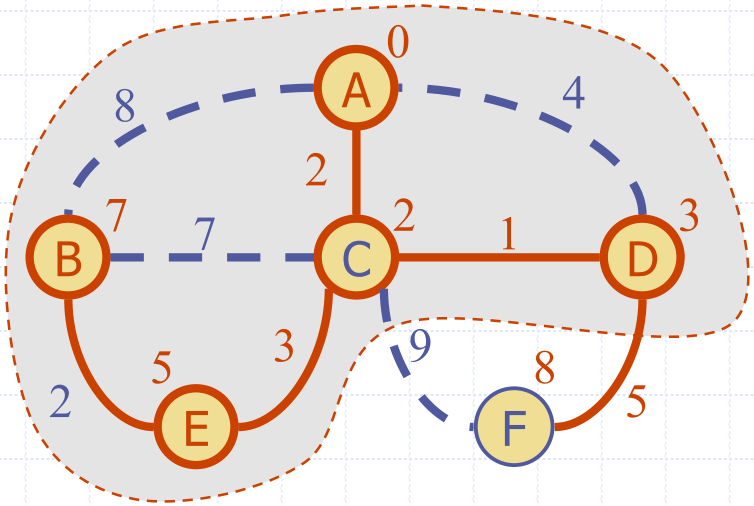
Dijkstra's Algorithm

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

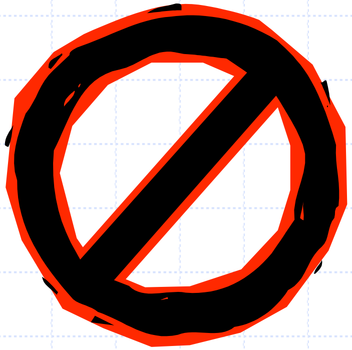
```
Algorithm DijkstraDistances(G, s)
  Q ← new heap-based priority queue
  for all v ∈ G.vertices()
    if v = s
      setDistance(v, 0)
    else
      setDistance(v, ∞)
    l ← Q.insert(getDistance(v), v)
    setLocator(v,l)
  while ¬Q.isEmpty()
    u ← Q.removeMin()
    for all e ∈ G.incidentEdges(u)
      { relax edge e }
      z ← G.opposite(u,e)
      r ← getDistance(u) + weight(e)
      if r < getDistance(z)
        setDistance(z,r)
        Q.replaceKey(getLocator(z),r)
```

Why Dijkstra's Algorithm Works

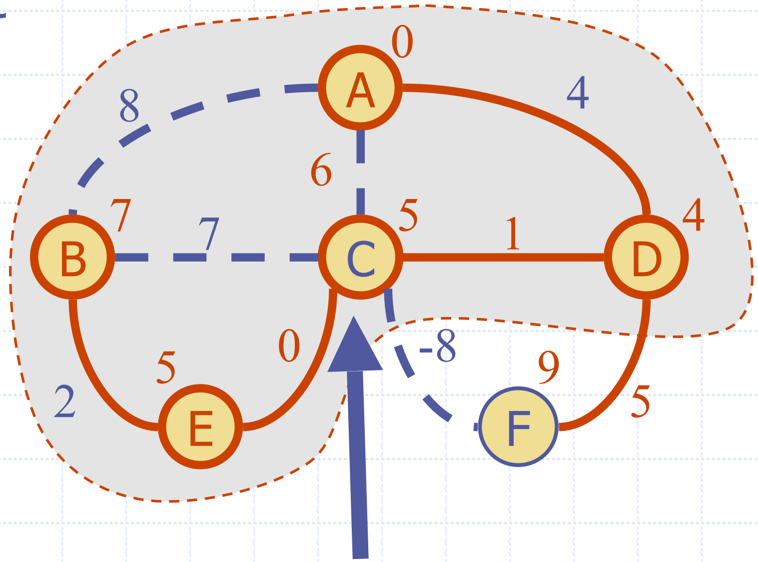
- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.
  - Suppose it didn't find all shortest distances. Let F be the first wrong vertex the algorithm processed.
  - When the previous node, D, on the true shortest path was considered, its distance was correct.
  - But the edge (D,F) was **relaxed** at that time!
  - Thus, so long as d(F)≥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.
  - There is an alternative, called the Bellman-Ford algorithm, which is less efficient but works even with negative-weight edges (as long as there is no negative-weight cycle).



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

Shortest Paths Tree

- 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)

    …

  for all  v ∈ G.vertices()
    …
    setParent(v, ∅)
    …

    for all  e ∈ G.incidentEdges(u)
      { relax edge e }
      z ← G.opposite(u,e)
      r ← getDistance(u) + weight(e)
      if  r < getDistance(z)
        setDistance(z,r)
        setParent(z,e)
        Q.replaceKey(getLocator(z),r)
```

Analysis of Dijkstra's Algorithm (and Prim-Jarník Algorithm)

- Insert/remove from Priority Queue once per vertex (total n)
- Update distances once per edge (total m)
- Adjacency List structure for Graph [efficient incidentEdges()]
- Overall efficiency depends on Priority Queue Implementation:

	Heap	Unordered Array/Vector
n x remove	$O(n \log n)$	$O(n^2)$
m x update	$O(m \log n)$	$O(m)$
total	$O((m+n) \log n)$	$O(n^2+m) = O(n^2)$

Better if $m < n^2/ \log n$

Better if $m > n^2/ \log n$