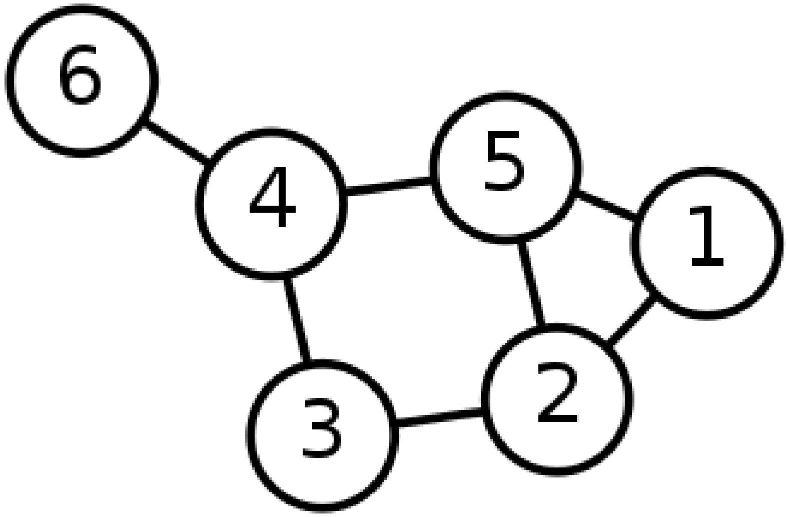
Djikstra’s Algorithm

Slide Courtesy: Uwash, UT

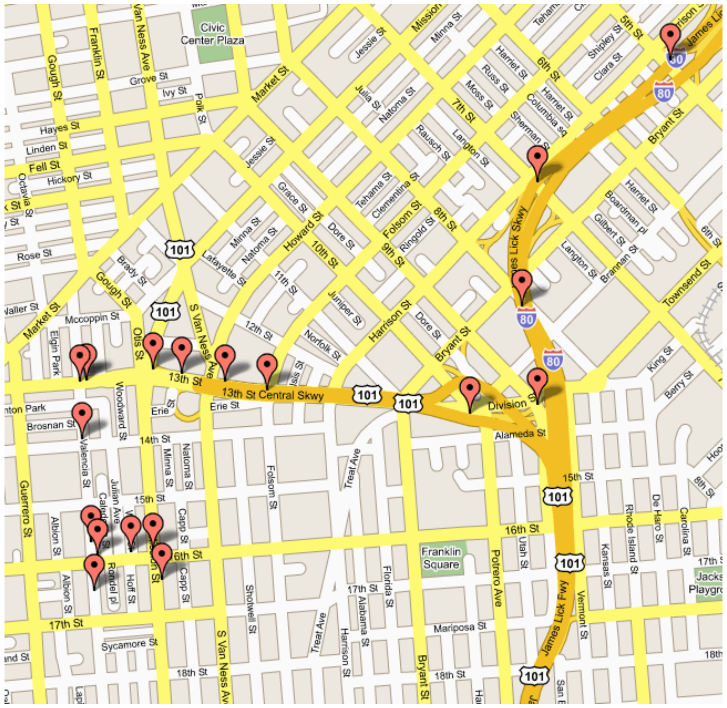
Single-Source Shortest Path Problem

Single-Source Shortest Path Problem - The problem of finding shortest paths from a source vertex *s* to all other vertices in the graph.

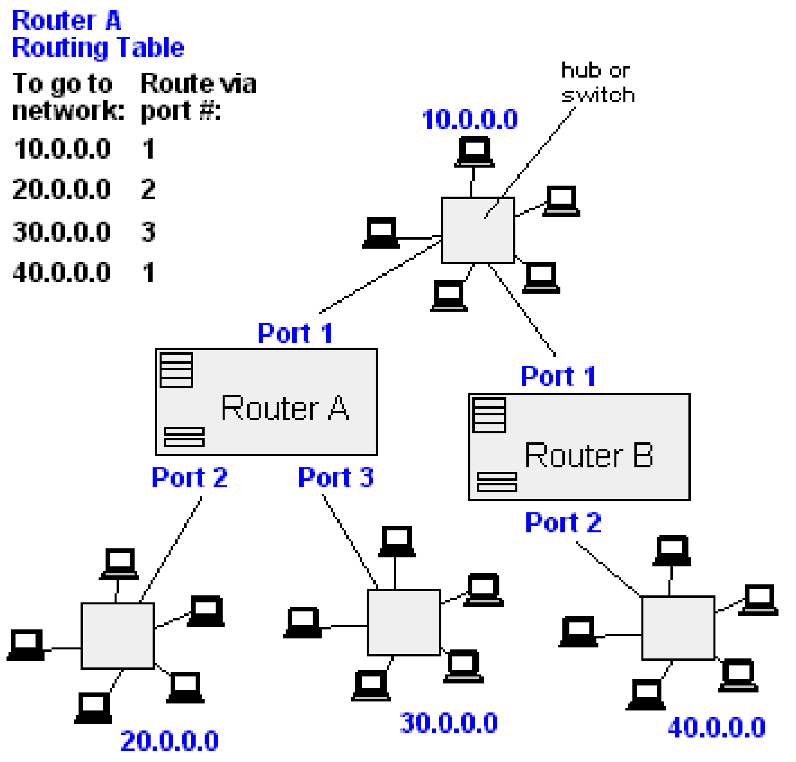


Applications

- Maps (Map Quest, Google Maps)
- Routing Systems



From Computer Desktop Encyclopedia
© 1998 The Computer Language Co. Inc.



Router A
Routing Table

To go to network:	Route via port #:
10.0.0.0	1
20.0.0.0	2
30.0.0.0	3
40.0.0.0	1

Dijkstra's algorithm

Dijkstra's algorithm - is a solution to the single-source shortest path problem in graph theory.

Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

Input: Weighted graph G={E,V} and source vertex *s*∈V, such that all edge weights are nonnegative

Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex *s*∈V to all other vertices

Approach

- The algorithm computes for each vertex u the distance to u from the start vertex s, that is, the weight of a shortest path between s and u.
- the algorithm keeps track of the set of vertices for which the distance has been computed, called the cloud C
- Every vertex has a label D associated with it. For any vertex u, D[u] stores an approximation of the distance between s and u. The algorithm will update a D[u] value when it finds a shorter path from s to u.
- When a vertex u is added to the cloud, its label D[u] is equal to the actual (final) distance between the starting vertex s and vertex u.

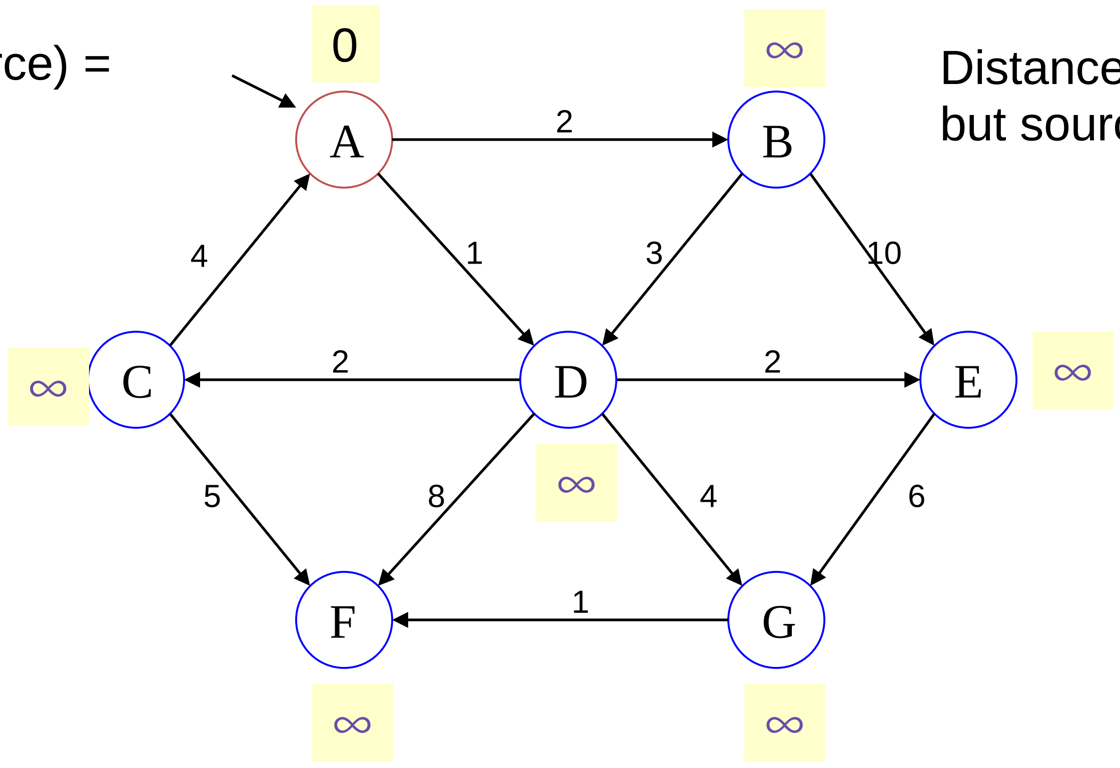
Dijkstra pseudocode

```
Dijkstra(s, t):
   for each vertex v:                    // Initialization
       v's distance := infinity.
       v's previous := none.
   s's distance := 0.
   List := {all vertices}.

   while List is not empty:
       v := remove List vertex with minimum distance.
       mark v as known.
       for each unknown neighbor n of v:
           dist := v's distance + edge (v, n)'s weight.

           if dist is smaller than n's distance:
               n's distance := dist.
               n's previous := v.

   reconstruct path from t back to s,
   following previous pointers.
```

Example: Initialization

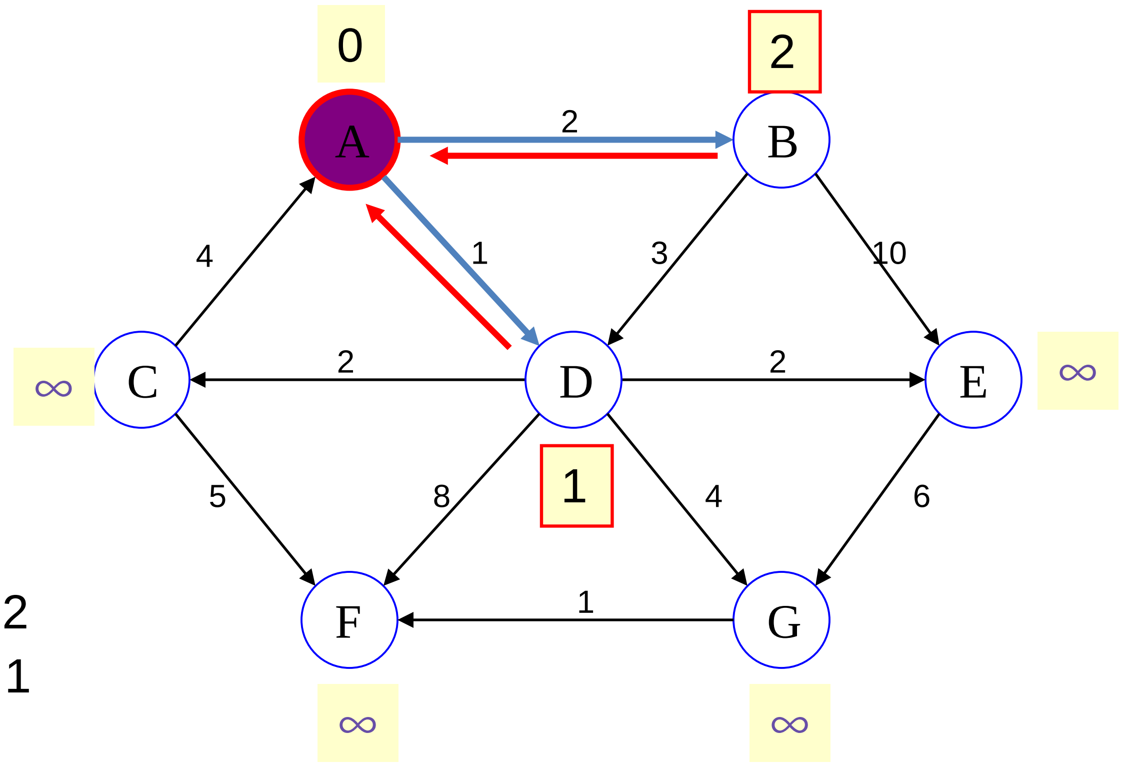
Distance(source) = 0

Distance (all vertices but source) = ∞



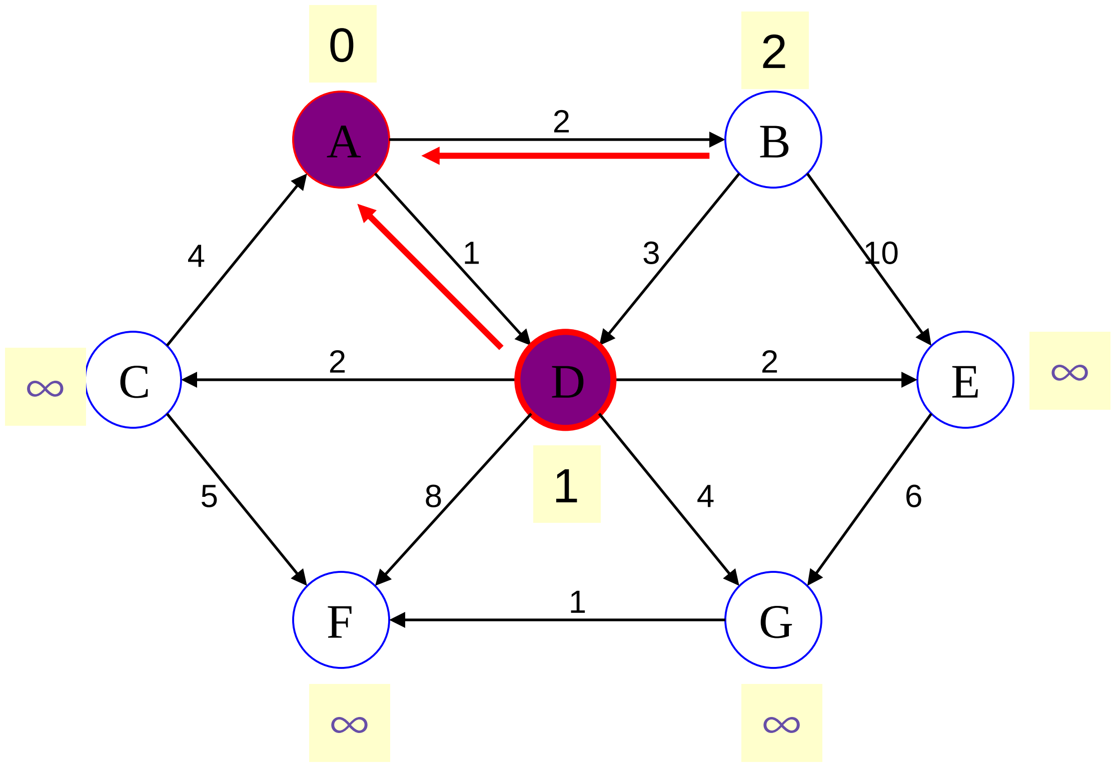
Pick vertex in List with minimum distance.

Example: Update neighbors' distance



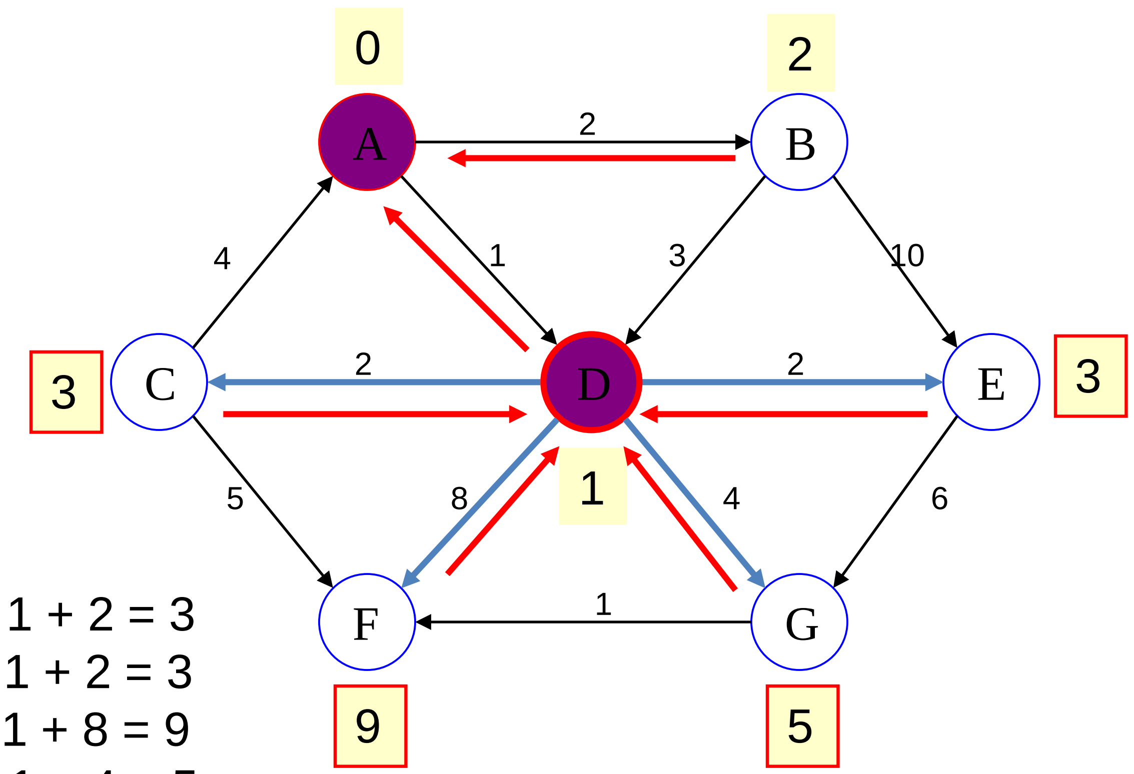
Distance(B) = 2
Distance(D) = 1

Example: Remove vertex with minimum distance



Pick vertex in List with minimum distance, i.e., D

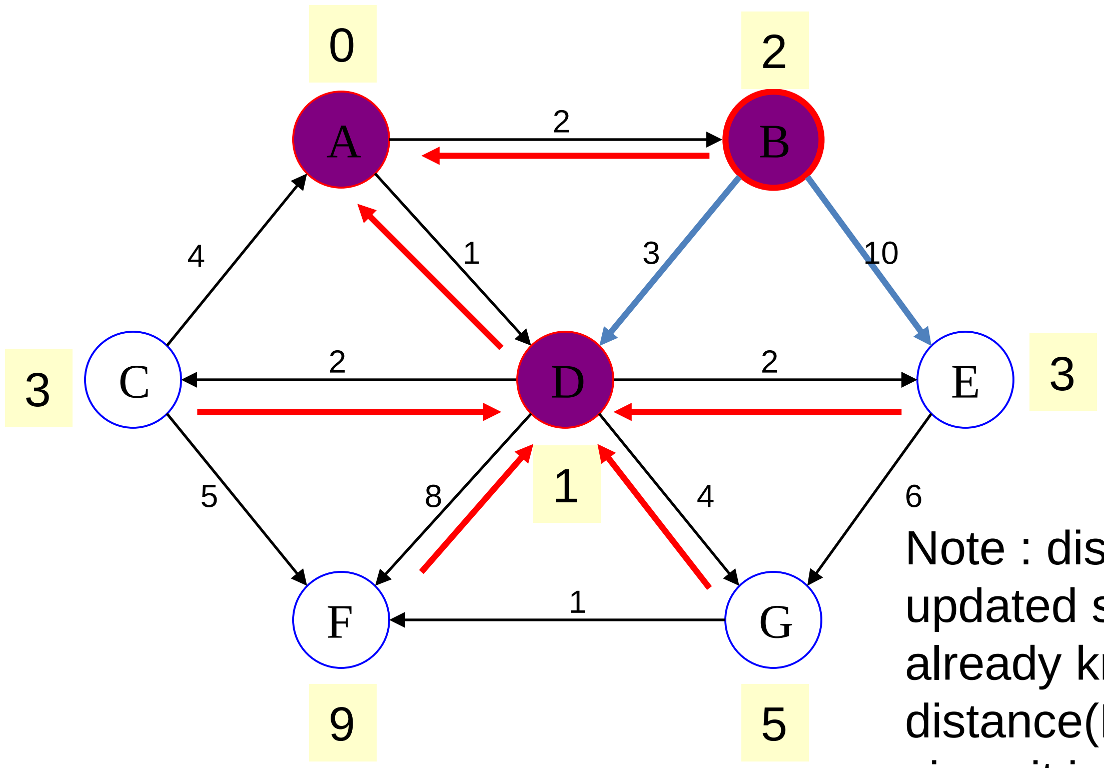
Example: Update neighbors



Distance(C) = 1 + 2 = 3
Distance(E) = 1 + 2 = 3
Distance(F) = 1 + 8 = 9
Distance(G) = 1 + 4 = 5

Example: Continued...

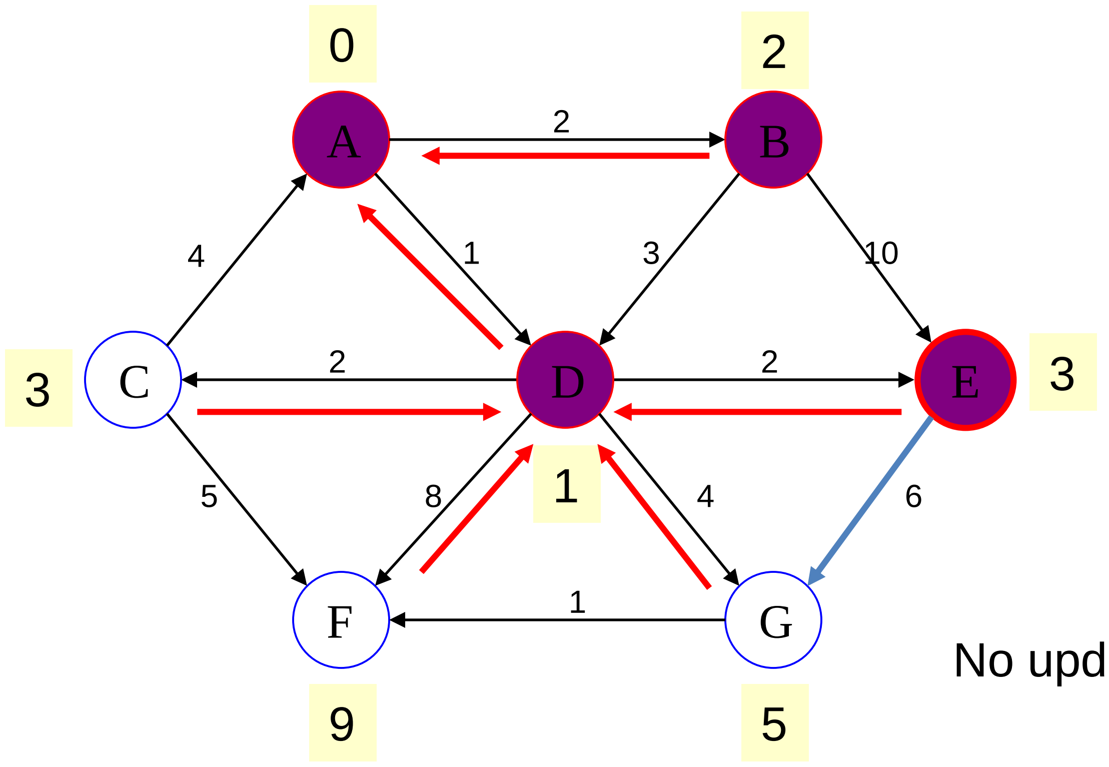
Pick vertex in List with minimum distance (B) and update neighbors



Note : distance(D) not updated since D is already known and distance(E) not updated since it is larger than previously computed

Example: Continued...

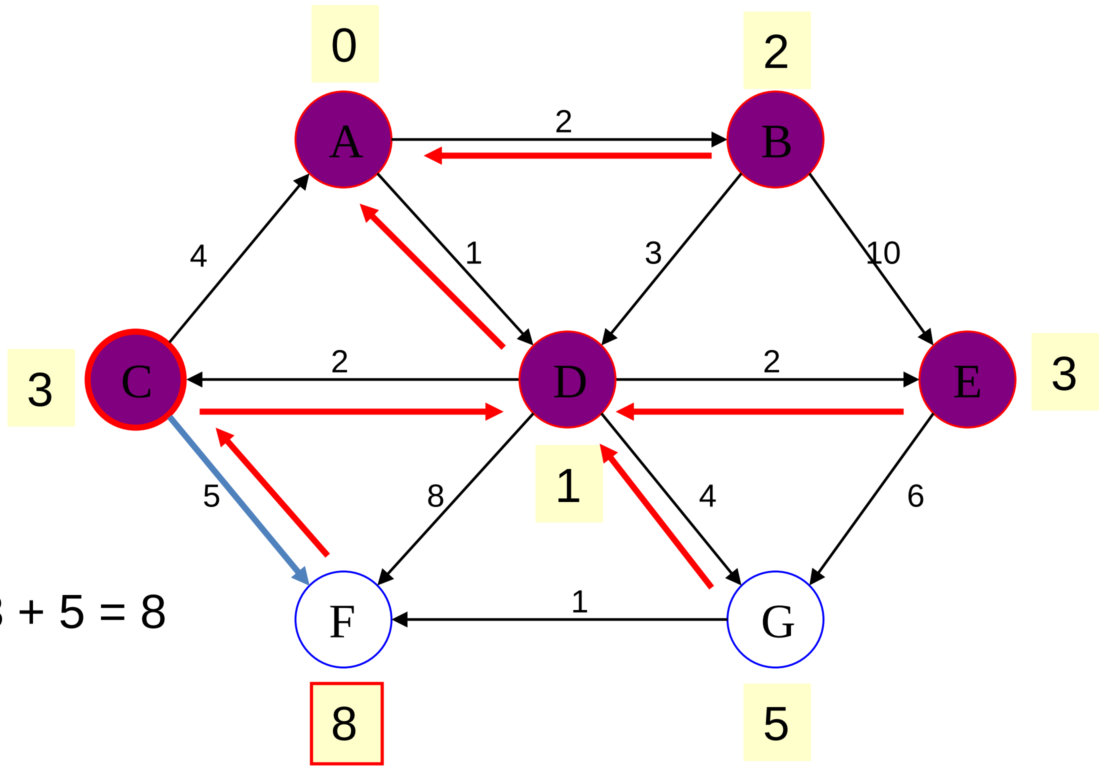
Pick vertex List with minimum distance (E) and update neighbors



No updating

Example: Continued...

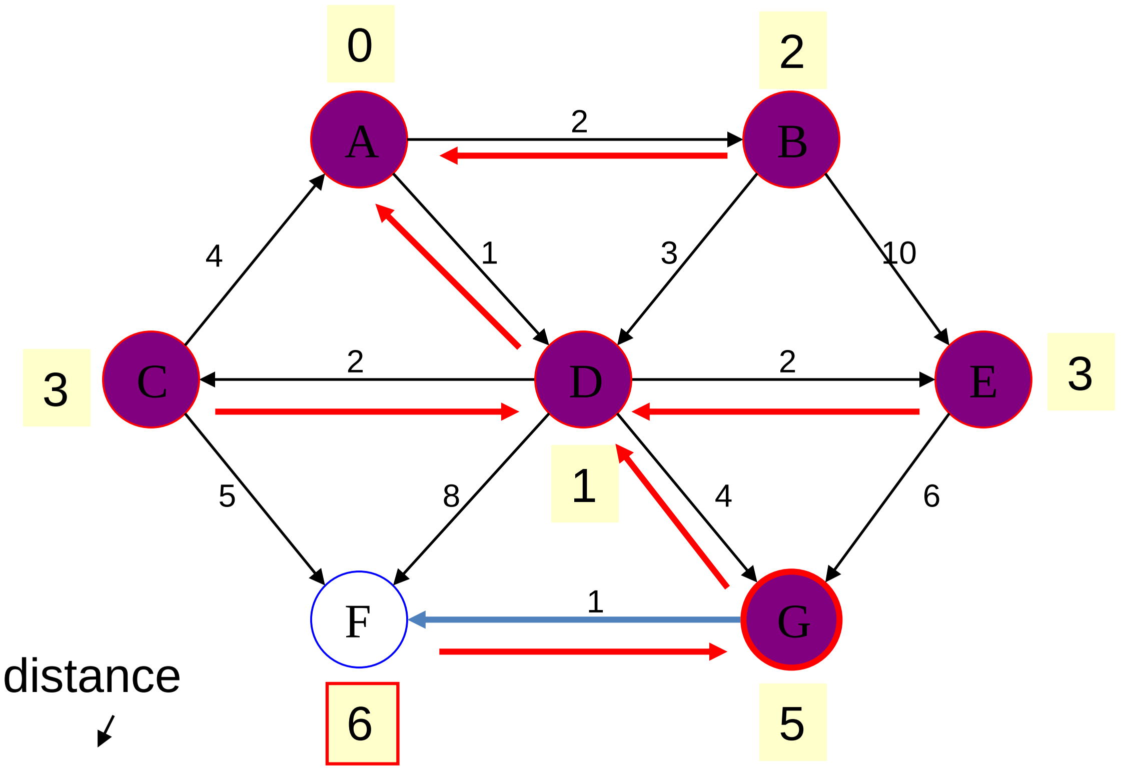
Pick vertex List with minimum distance (C) and update neighbors



Distance(F) = 3 + 5 = 8

Example: Continued...

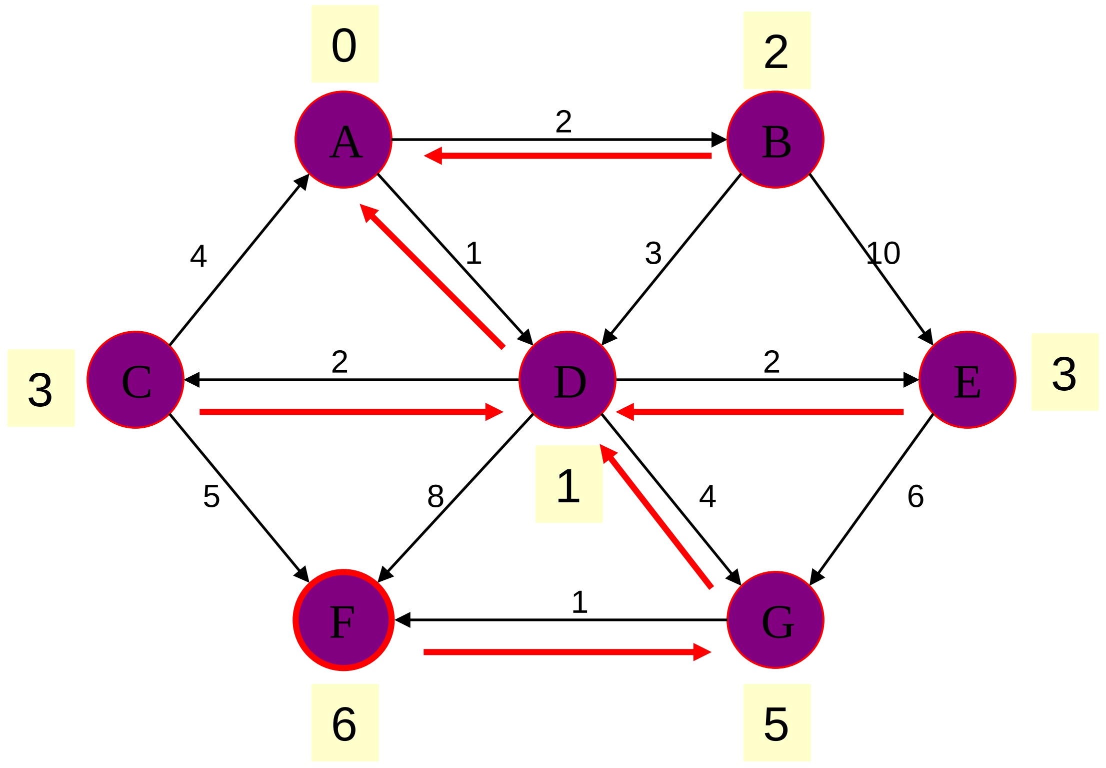
Pick vertex List with minimum distance (G) and update neighbors



Previous distance

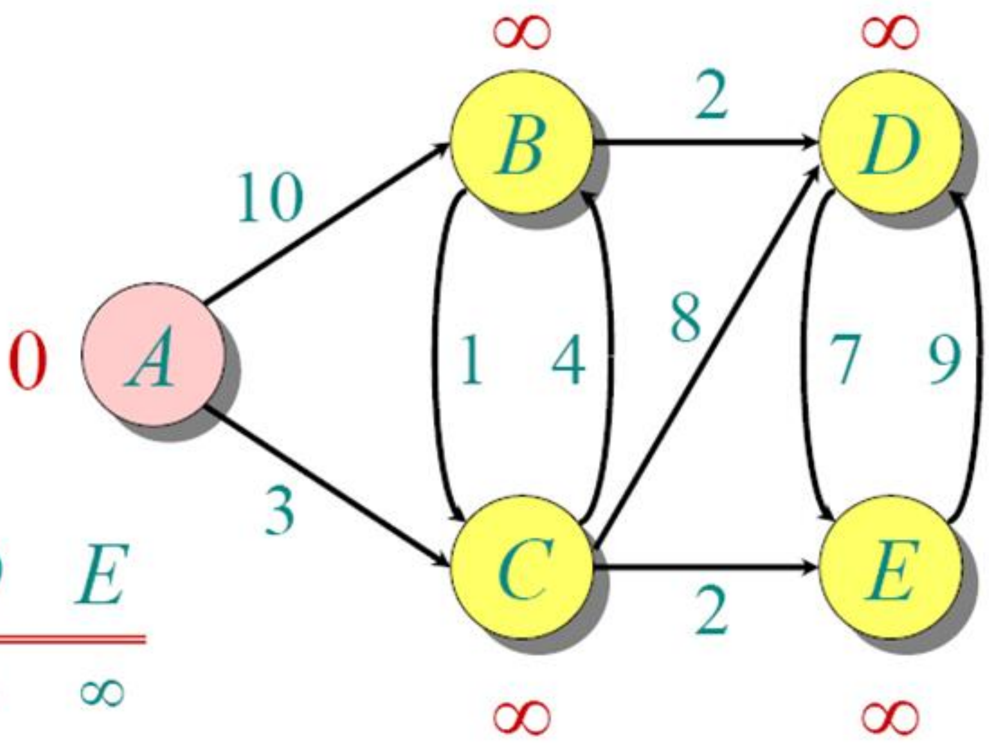
Distance(F) = min (8, 5+1) = 6

Example (end)



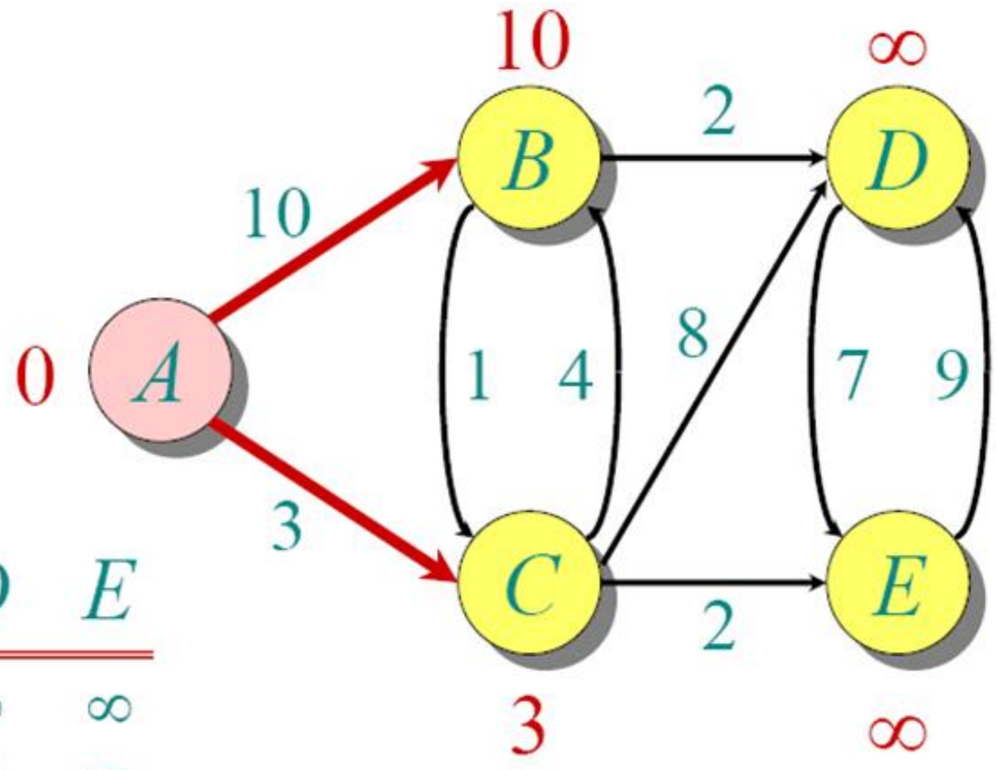
Pick vertex not in S with lowest cost (F) and update neighbors

Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

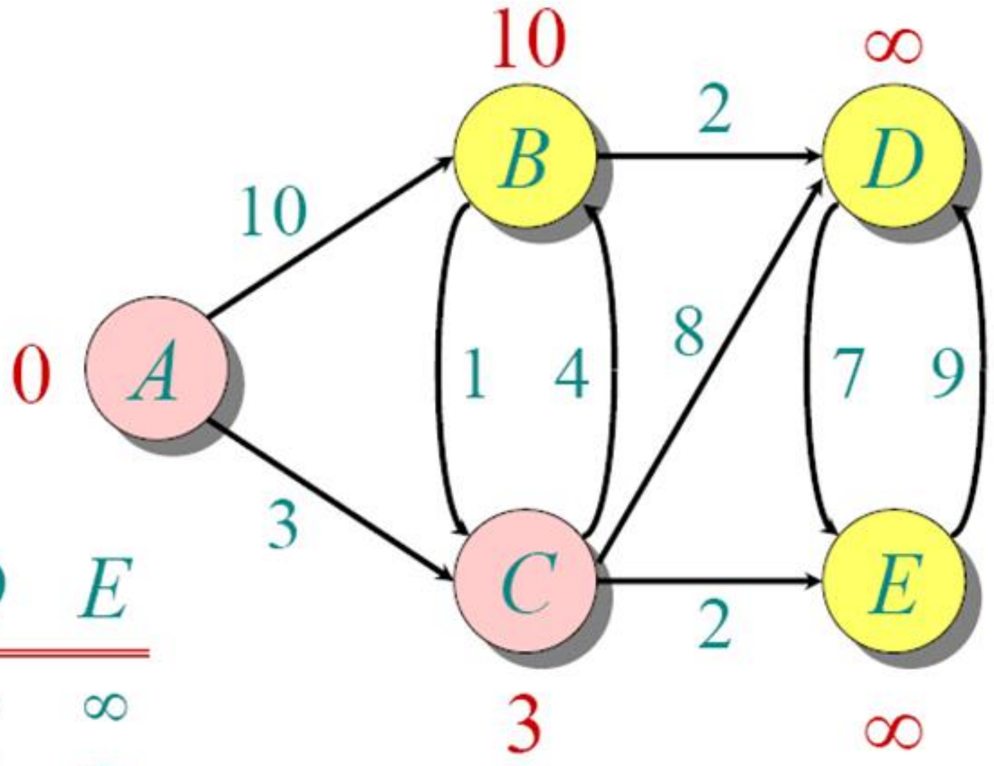
Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

$S: \{ A \}$

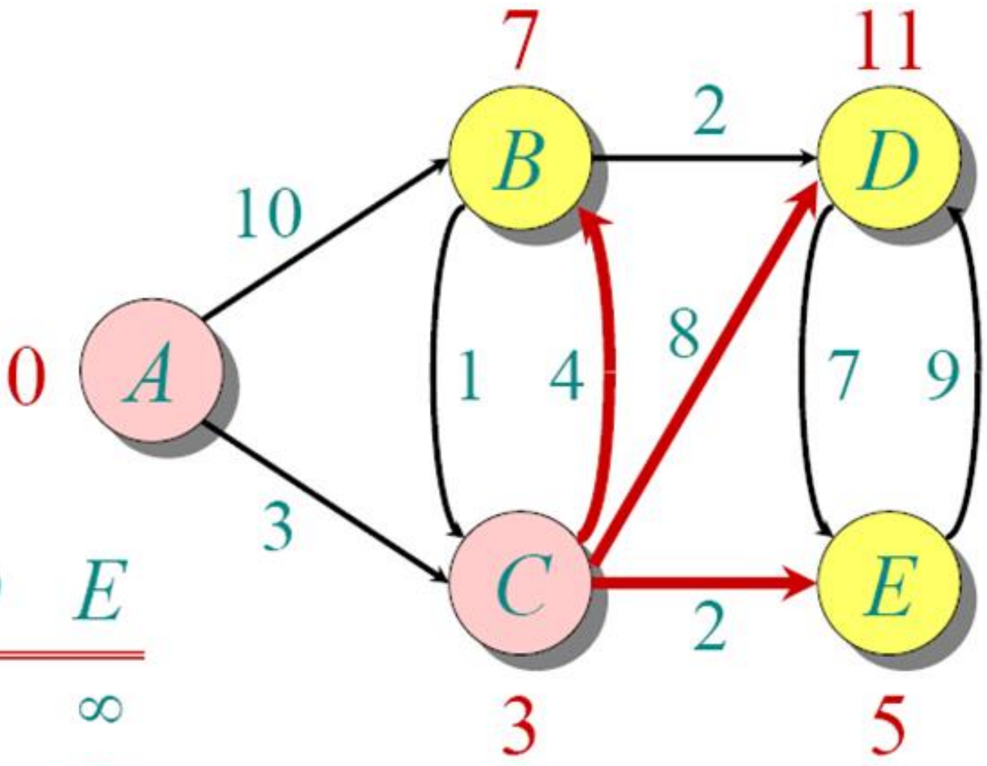
Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

$S: \{ A, C \}$

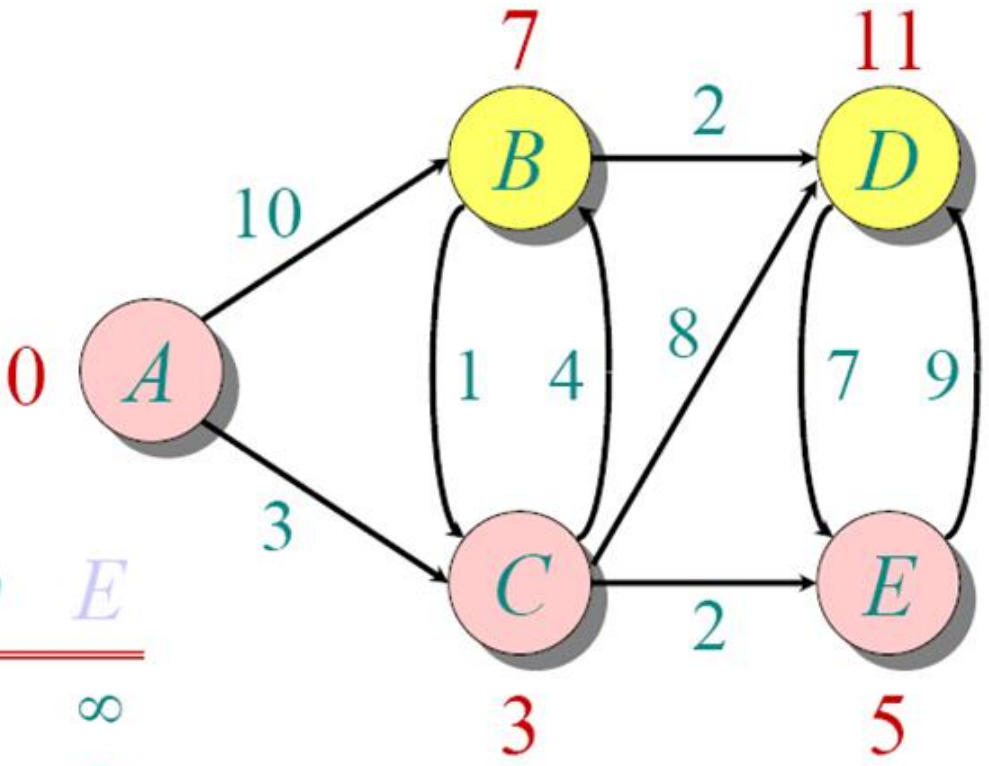
Another Example



Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { *A*, *C* }

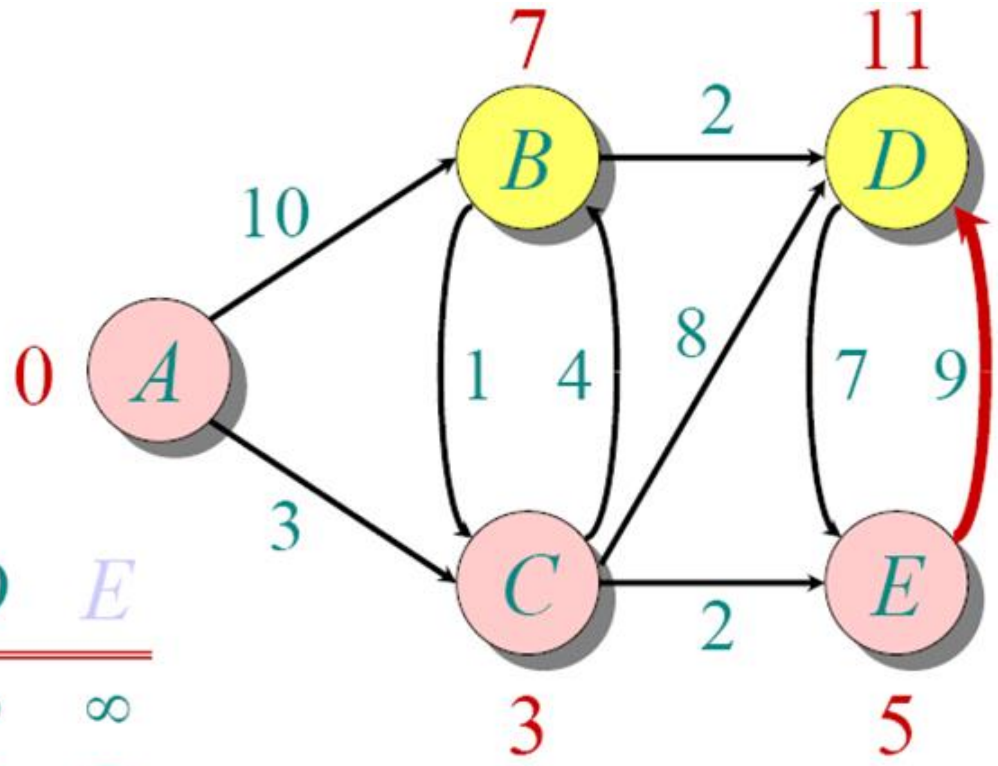
Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { *A*, *C*, *E* }

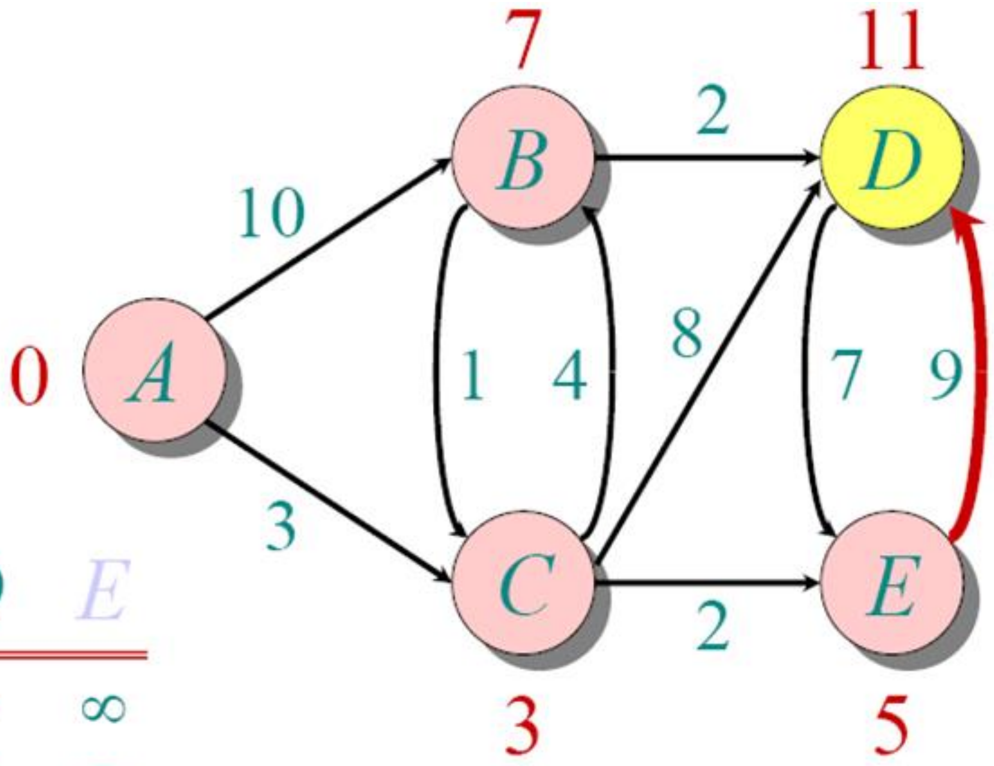
Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { *A*, *C*, *E* }

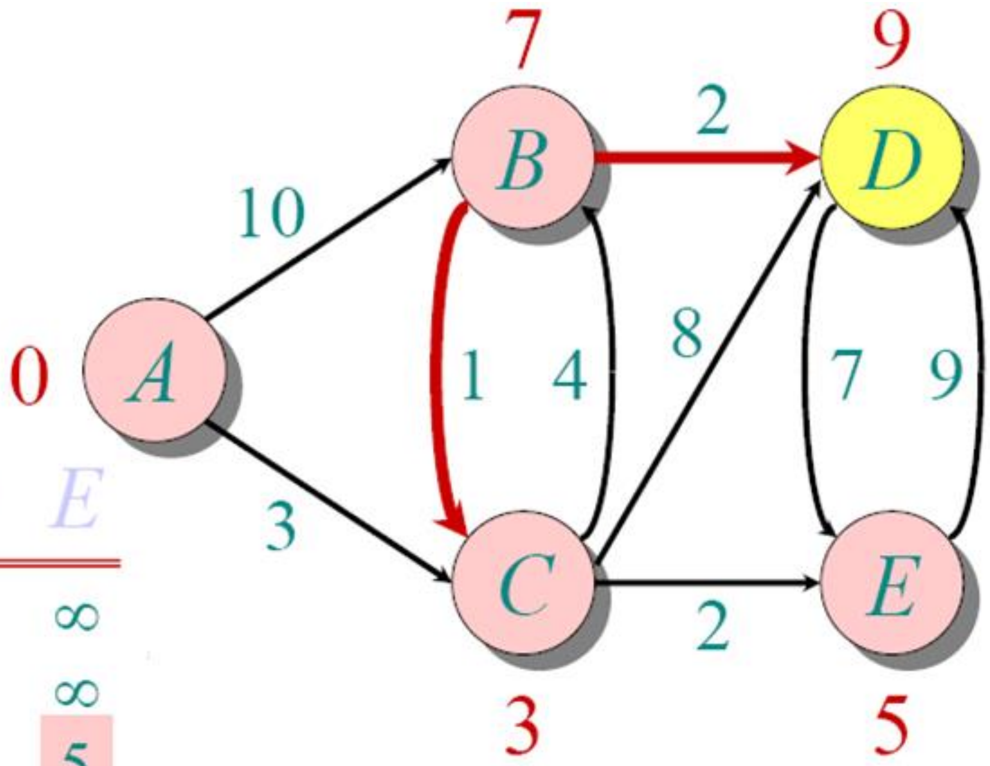
Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { *A*, *C*, *E*, *B* }

Another Example



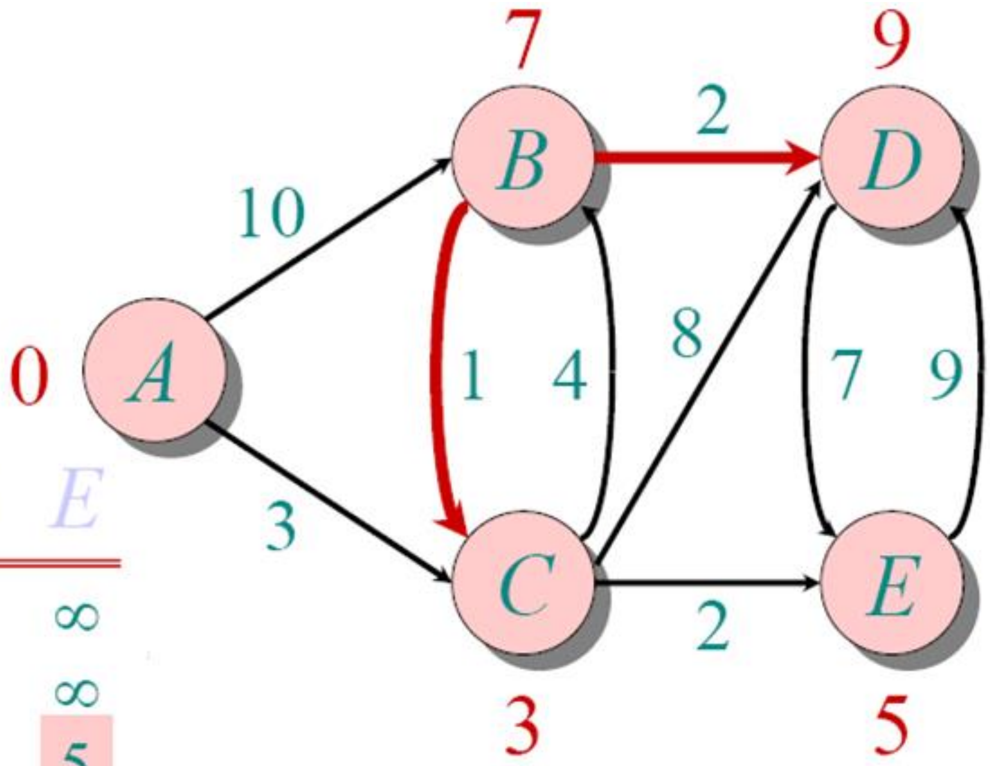
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { A, C, E, B }

Another Example

Q:

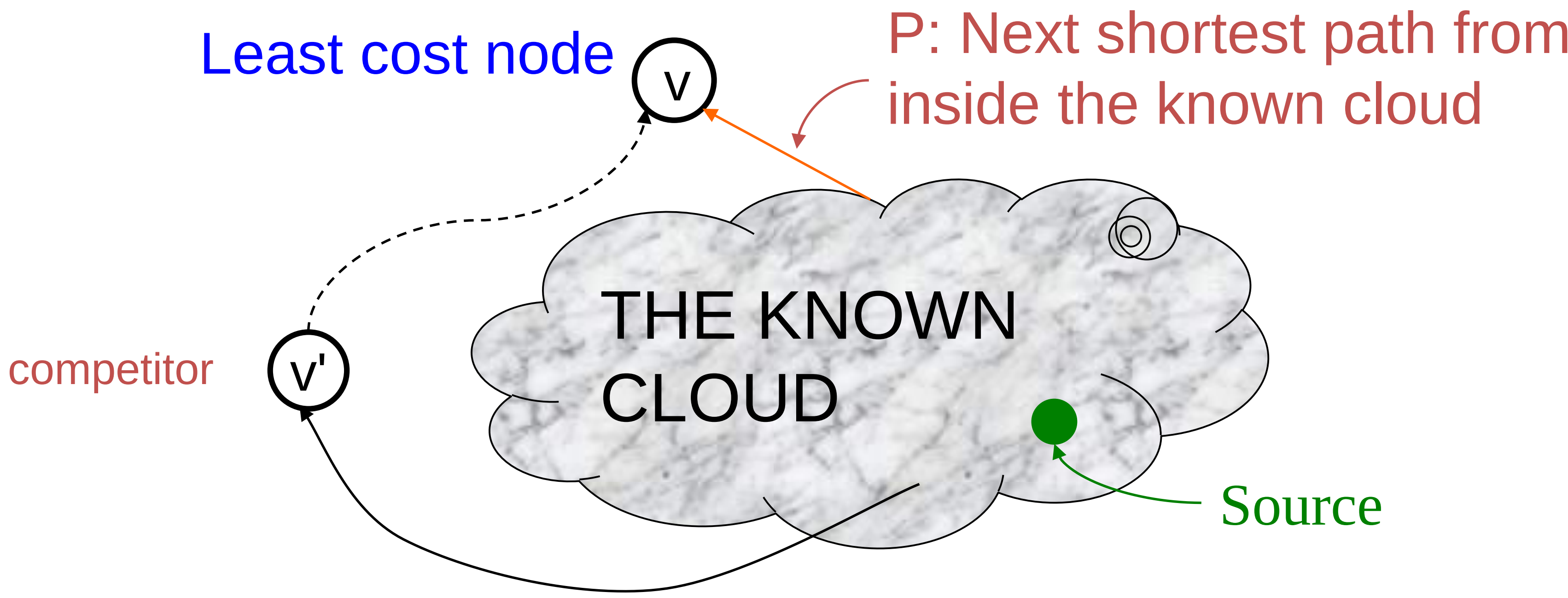
A	*B*	*C*	*D*	*E*
0	∞	∞	∞	∞
	10	3	∞	∞
	7		11	5
	7		11	
			9	



S: { *A*, *C*, *E*, *B*, *D* }

Correctness :“Cloudy” Proof

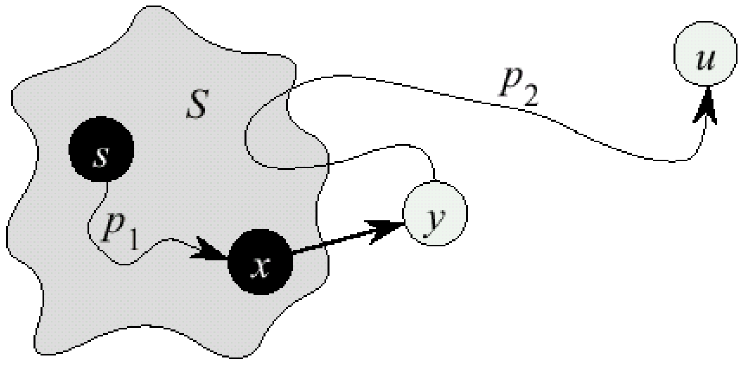
When a vertex is added to the cloud, it has shortest distance to source.



- If the path to v is the next shortest path, the path to v' must be at least as long. Therefore, any path through v' to v cannot be shorter!

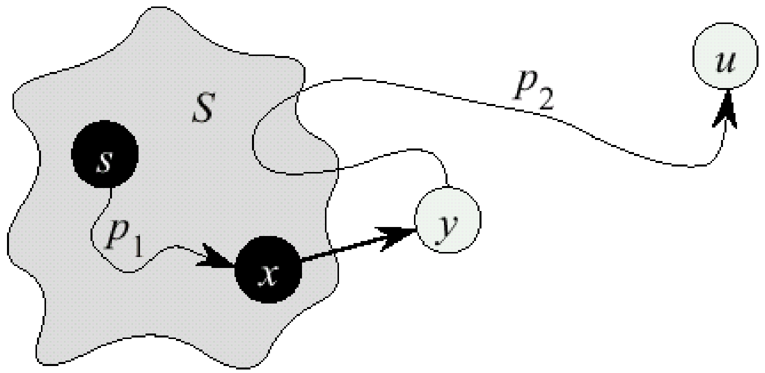
Dijkstra’s Correctness

- We will prove that **whenever *u* is added to *S***, $d[u] = \delta(s,u)$, i.e., that *d[u]* is minimum, and that equality is maintained thereafter
- Proof
  - Note that for all *v not in S*, $d[v] \geq \delta(s,v)$
  - Let *u* be the first **vertex picked** such that there is a shorter path than $d[u]$, i.e., that $d[u] > \delta(s,u)$
  - We will show that this assumption leads to a contradiction



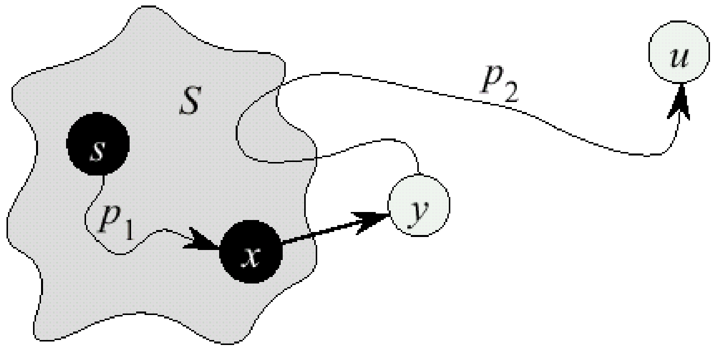
Dijkstra Correctness (2)

- Let *y* be the first vertex in *V* – *S* on the actual shortest path from *s* to *u*, then it must be that *d*[*y*] = δ(*s*,*y*) because
  - *d*[*x*] is set correctly for *y*'s predecessor *x* in *S* on the shortest path (by choice of *u* as the first vertex for which *d* is set incorrectly)
  - when the algorithm inserted *x* into *S*, it relaxed the edge (*x*,*y*), assigning *d*[*y*] the correct value



Dijkstra Correctness (3)

$$\begin{aligned} d[u] &> \delta(s,u) && \text{(initial assumption)} \\ &= \delta(s,y) + \delta(y,u) && \text{(optimal substructure)} \\ &= d[y] + \delta(y,u) && \text{(correctness of } d[y]\text{)} \\ &\geq d[y] && \text{(no negative weights)} \end{aligned}$$



- But if *d*[*u*] > *d*[*y*], the algorithm would have chosen *y* (from the Q) to process next, not *u* –– Contradiction
- Thus d[u] = δ(*s*,*u*) at time of insertion of *u* into *S*, and Dijkstra's algorithm is correct

Dijkstra’s Pseudo Code

- Graph *G*, weight function *w*, root *s*

```
DIJKSTRA($G, w, s$)
 1  for each $v \in V$
 2        do $d[v] \leftarrow \infty$
 3  $d[s] \leftarrow 0$
 4  $S \leftarrow \emptyset$    $\triangleright$ Set of discovered nodes
 5  $Q \leftarrow V$
 6  while $Q \neq \emptyset$
 7        do $u \leftarrow$ EXTRACT-MIN($Q$)
 8           $S \leftarrow S \cup \{u\}$
 9           for each $v \in Adj[u]$
10                do if $d[v] > d[u] + w(u, v)$
11                      then $d[v] \leftarrow d[u] + w(u, v)$
```

relaxing edges

Time Complexity: Using List

The simplest implementation of the Dijkstra's algorithm stores vertices in an ordinary linked list or array

- Good for dense graphs (many edges)

- $|V|$ vertices and $|E|$ edges
- Initialization $O(|V|)$
- While loop $O(|V|)$
  - Find and remove min distance vertices $O(|V|)$
- Potentially $|E|$ updates
    - Update costs $O(1)$

Total time $O(|V^2| + |E|) = O(|V^2|)$

Time Complexity: Priority Queue

For sparse graphs, (i.e. graphs with much less than $|V^2|$ edges) Dijkstra's implemented more efficiently by *priority queue*

- Initialization O(|V|) using O(|V|) buildHeap
- While loop O(|V|)
  - Find and remove min distance vertices O(log |V|) using O(log |V|) deleteMin
- Potentially |E| updates
  - Update costs O(log |V|) using decreaseKey

Total time O(|V|log|V| + |E|log|V|) = O(|E|log|V|)

- |V| = O(|E|) assuming a connected graph