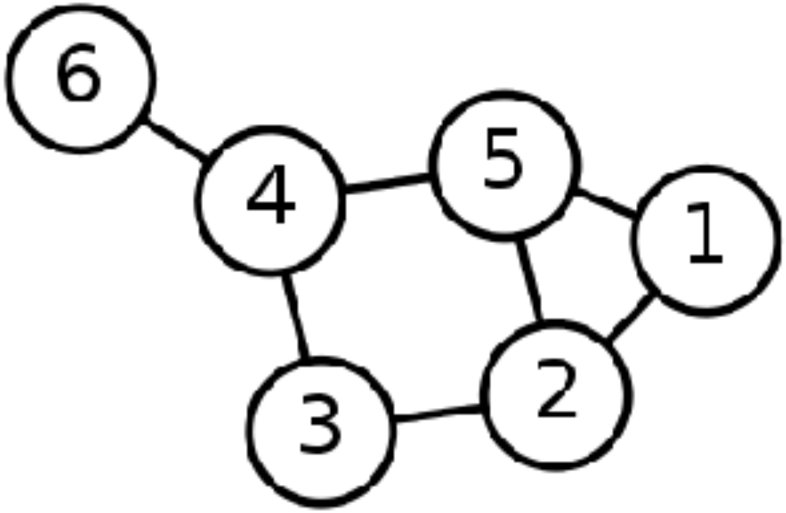
Dijkstra’s Algorithm

Single-Source Shortest Path Problem

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



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 $v \in V$, such that all edge weights are nonnegative

Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices

Approach

- The algorithm computes for each vertex u the distance to u from the start vertex v, that is, the weight of a shortest path between v 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 v and u. The algorithm will update a D[u] value when it finds a shorter path from v 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 v and vertex u.

Dijkstra's Pseudo Code

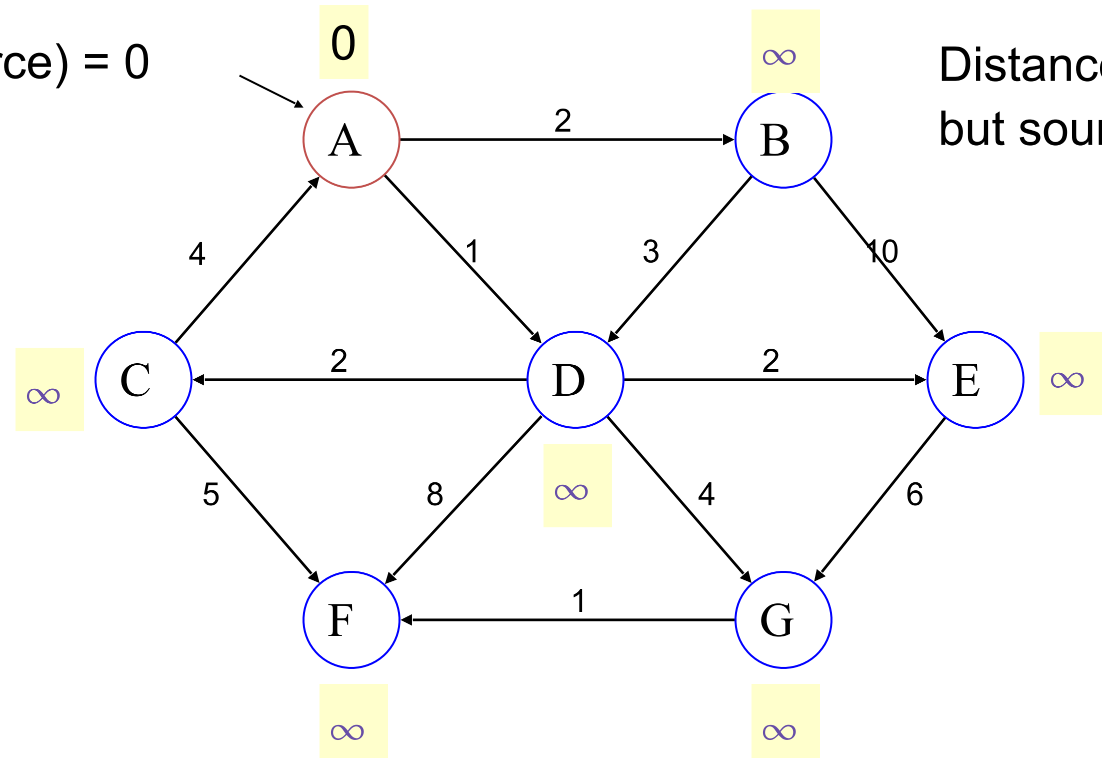
- Graph *G*, weight function *w*, root *s*

```
DIJKSTRA(G, w, s)
for each v ∈ V
        do d[v] ← ∞
d[s] ← 0
S ← ∅    ▷ Set of discovered nodes
Q ← V
while Q ≠ ∅
        do u ← EXTRACT-MIN(Q)
           S ← S ∪ {u}
           for each v ∈ Adj[u]
               do if d[v] > d[u] + w(u, v)
                     then d[v] ← d[u] + w(u, v)
```

relaxing edges

Example: Initialization

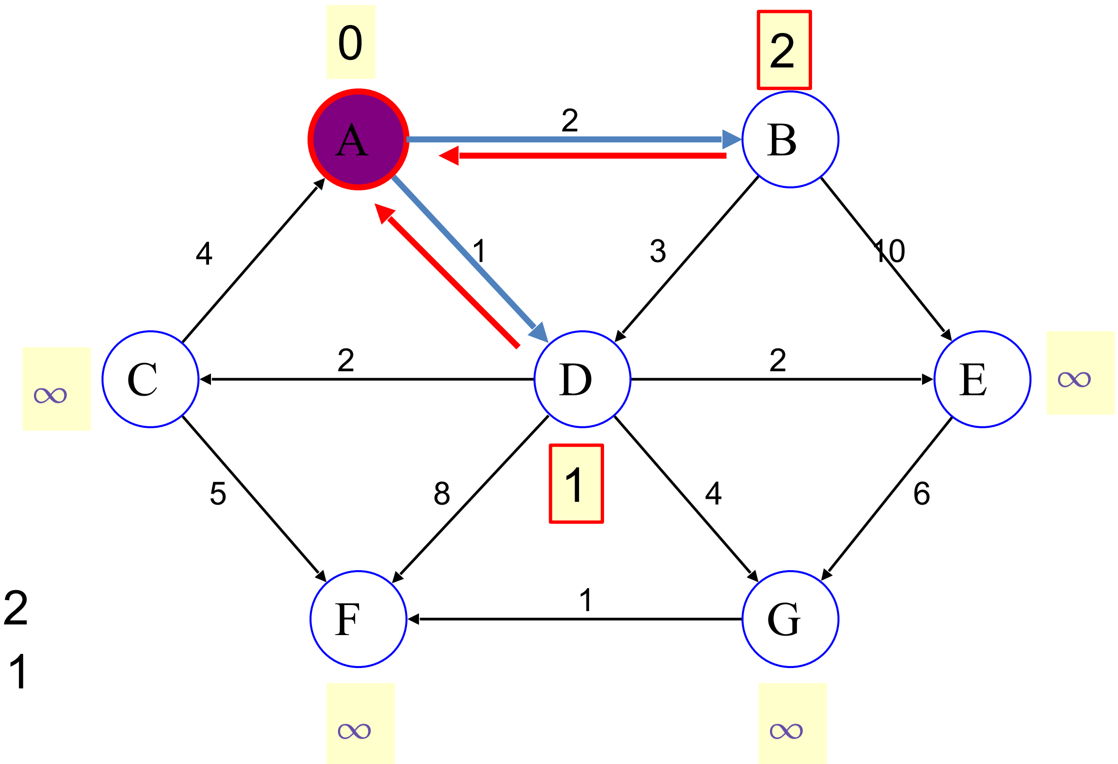
Distance(source) = 0



Distance (all vertices but source) = ∞

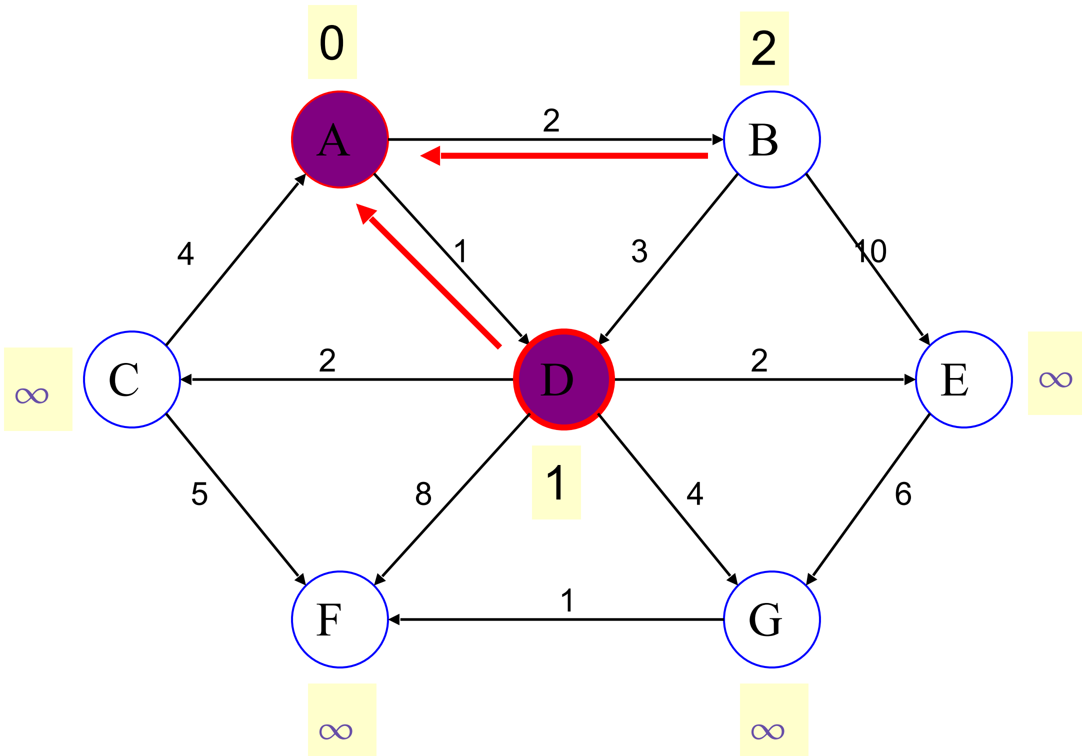
Pick vertex in List with minimum distance.

Example: Update neighbors' distance



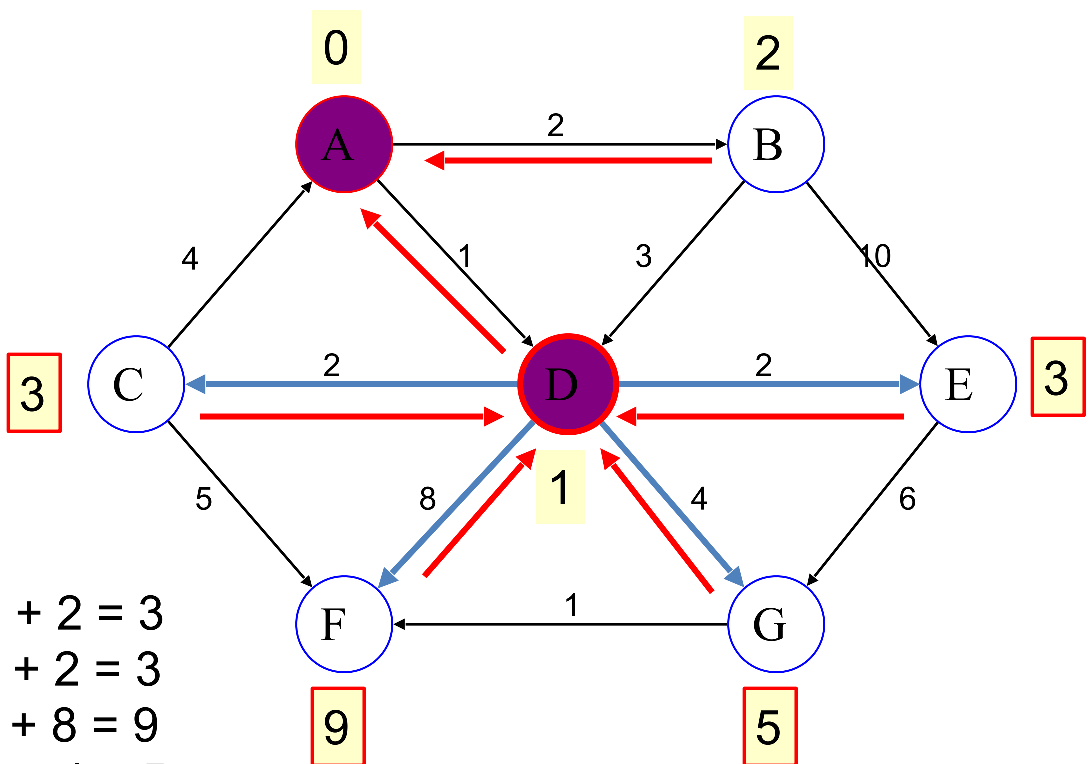
Distance(B) = 2
Distance(D) = 1

Example: Remove vertex with minimum distance



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

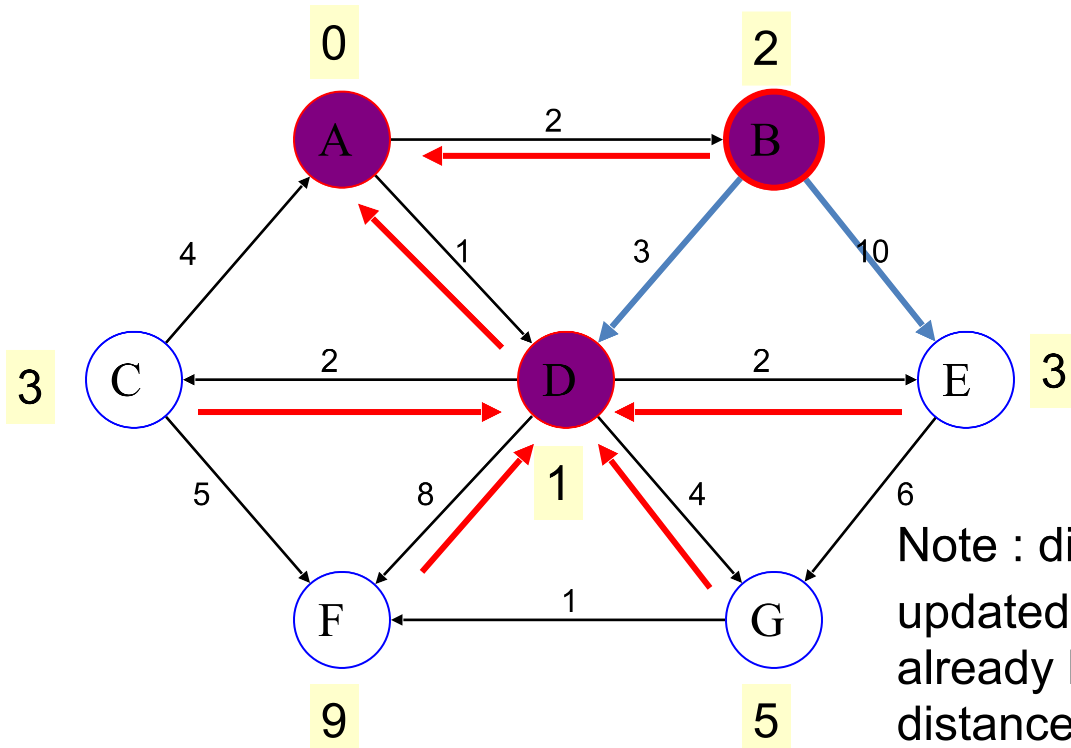
Example: Update neighbors



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

Example: Continued...

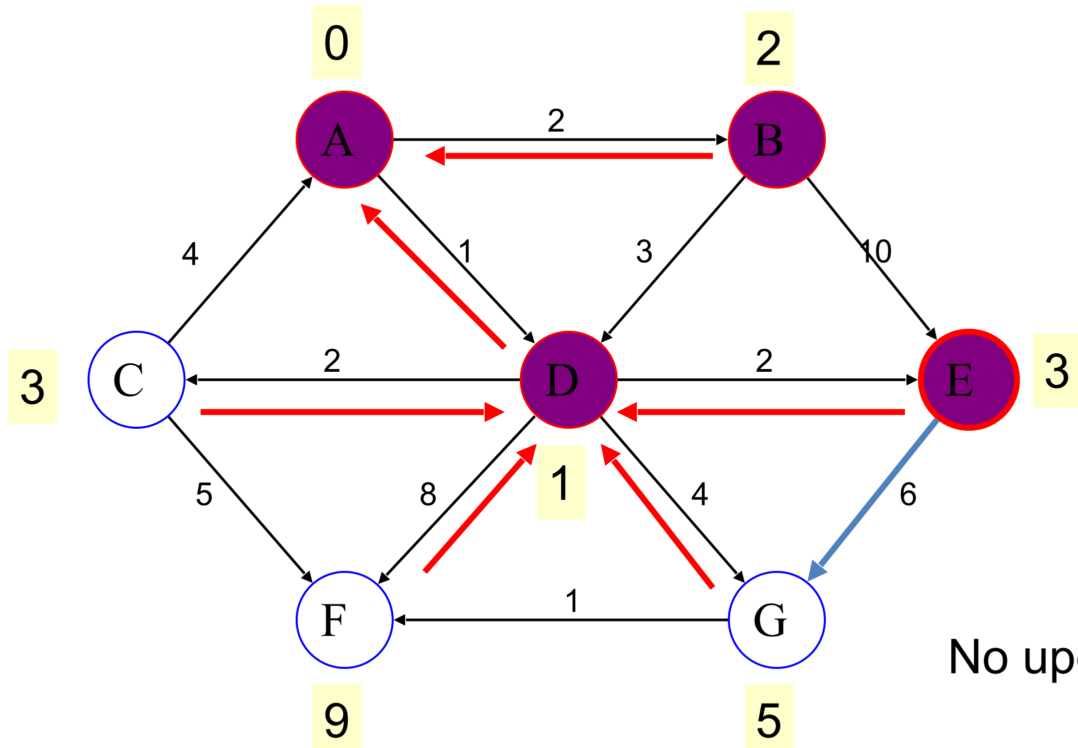
Pick vertex in List with minimum distance (B) and update neighbors



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

Example: Continued...

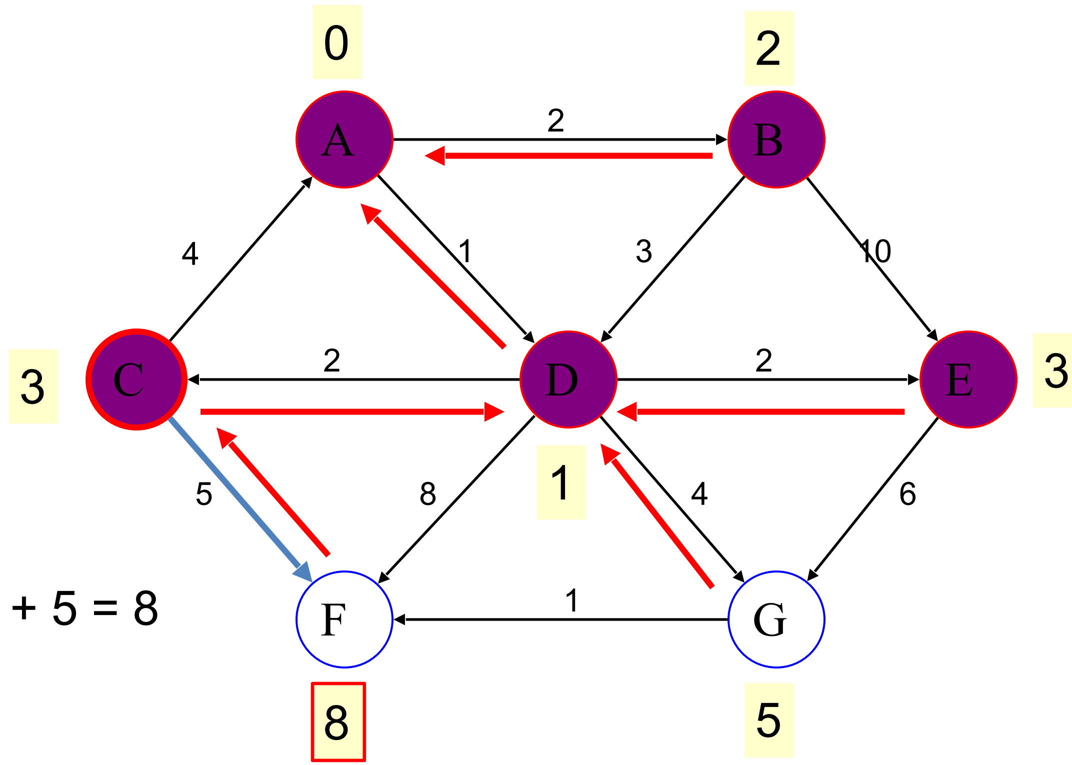
Pick vertex List with minimum distance (E) and update neighbors



No updating

Example: Continued...

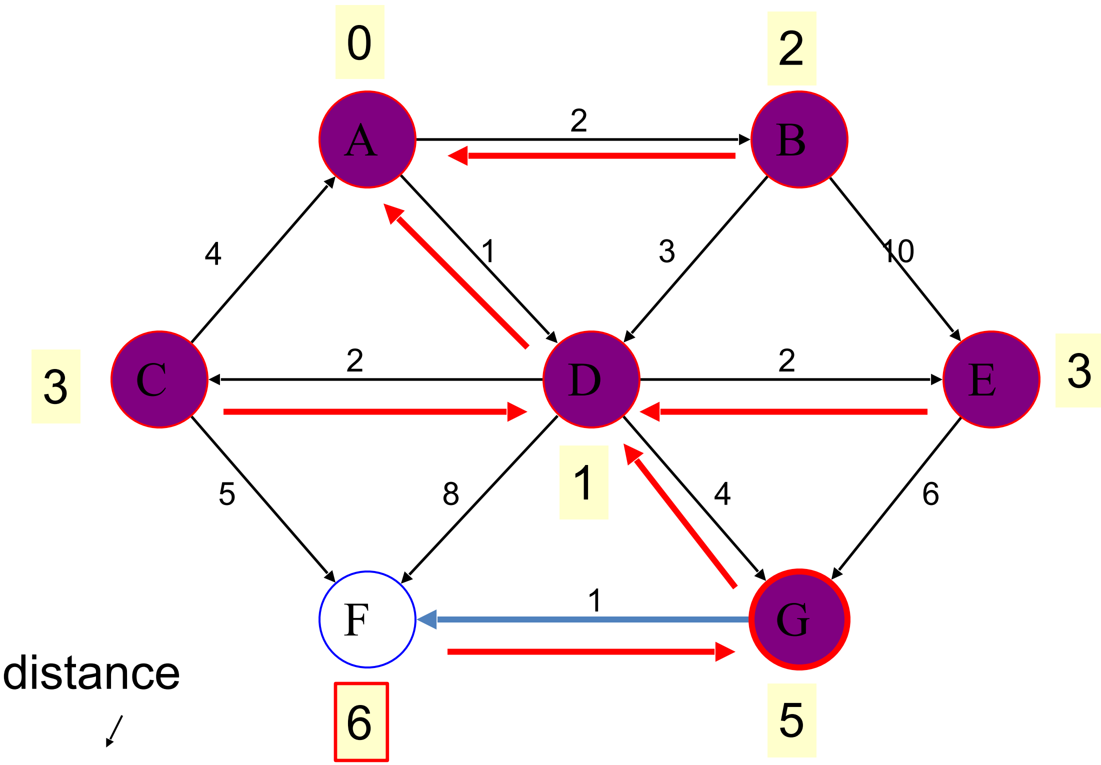
Pick vertex List with minimum distance (C) and update neighbors



Distance(F) = 3 + 5 = 8

Example: Continued...

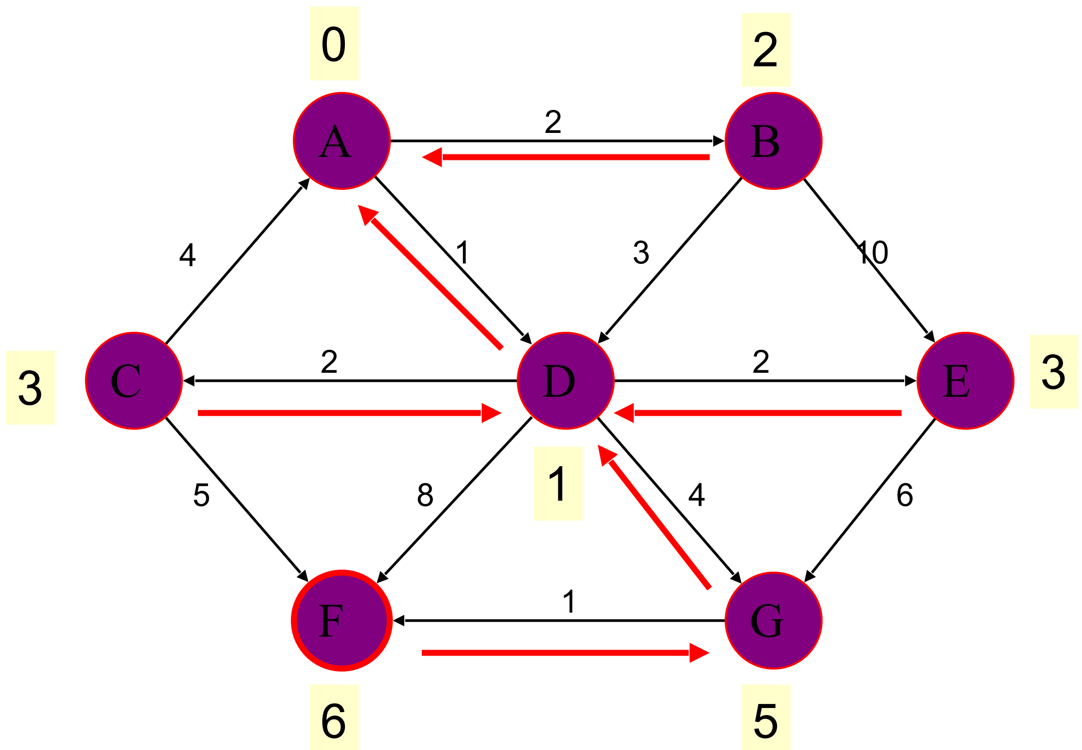
Pick vertex List with minimum distance (G) and update neighbors



Previous distance

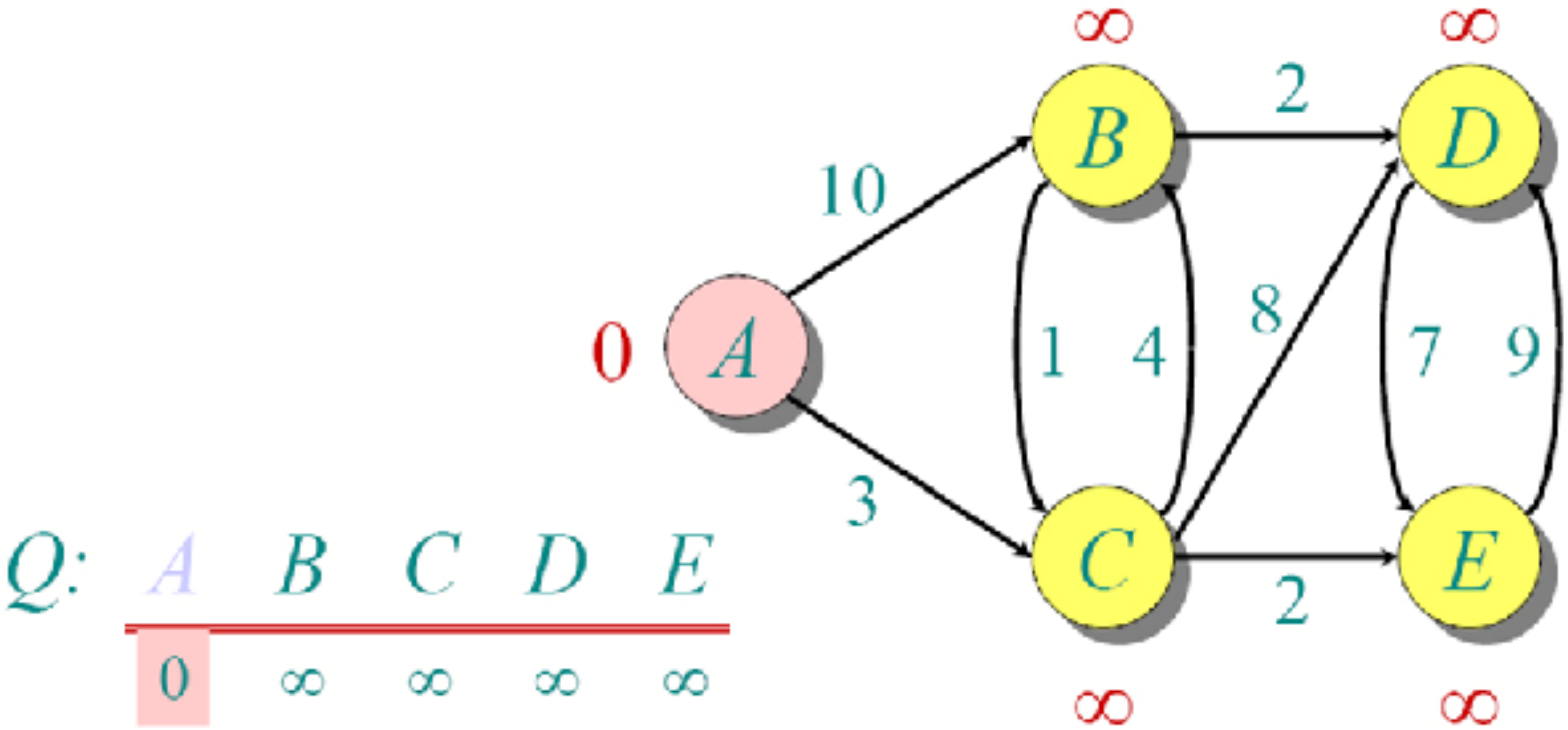
Distance(F) = min (8, 5+1) = 6

Example (end)

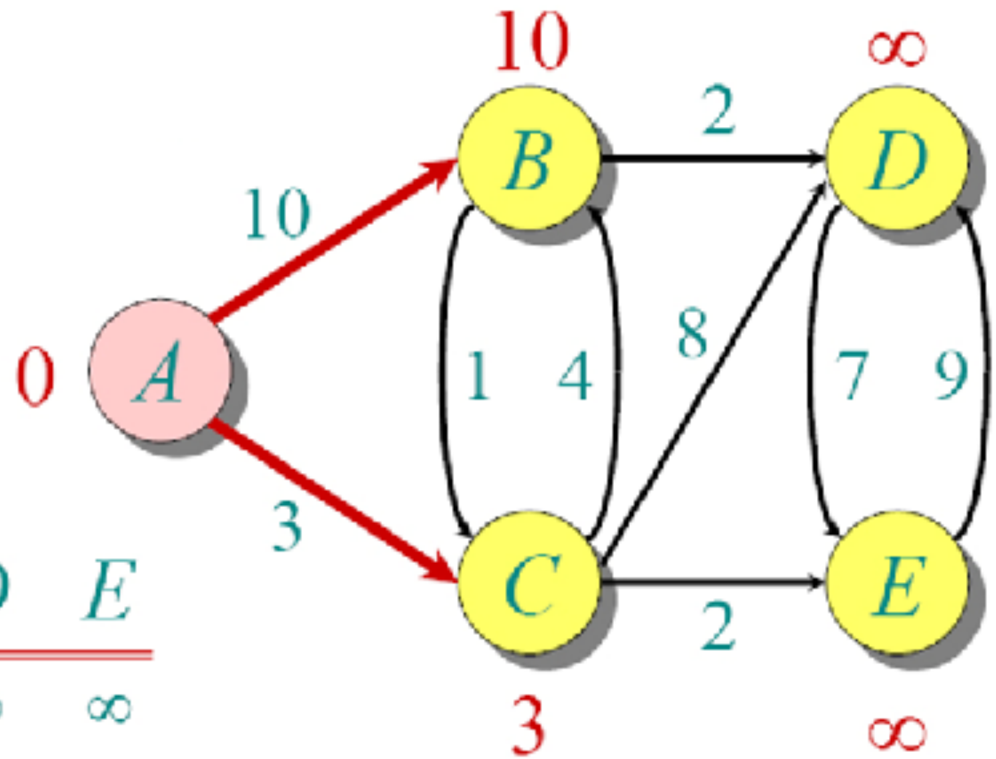


Pick vertex not in S with lowest cost (F) and update neighbors

Another Example



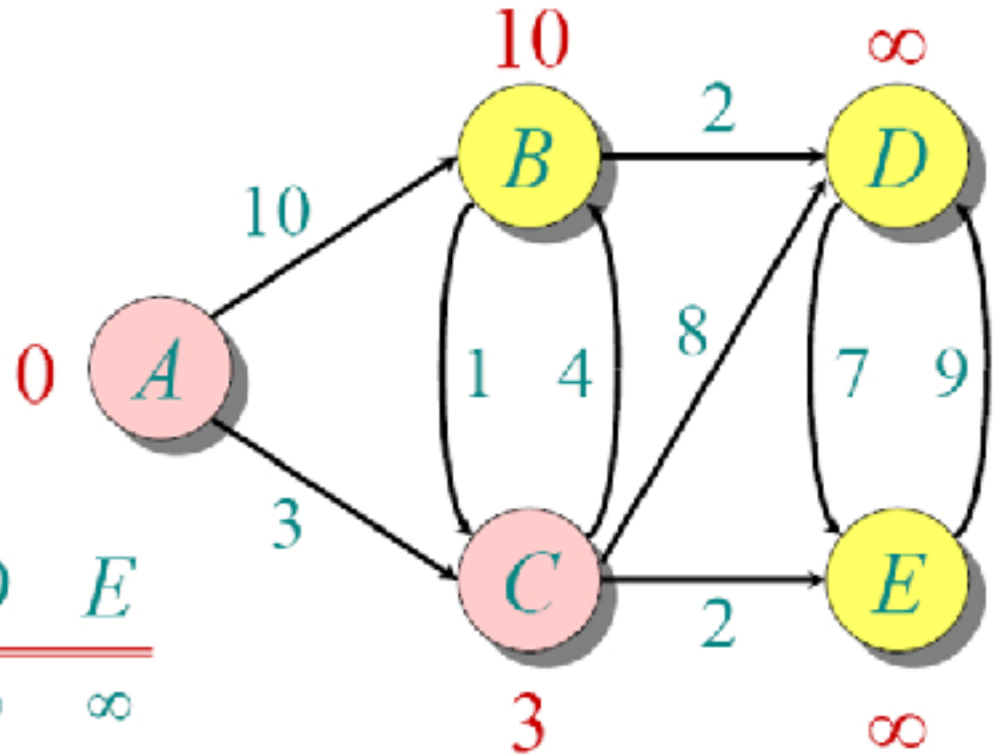
Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }

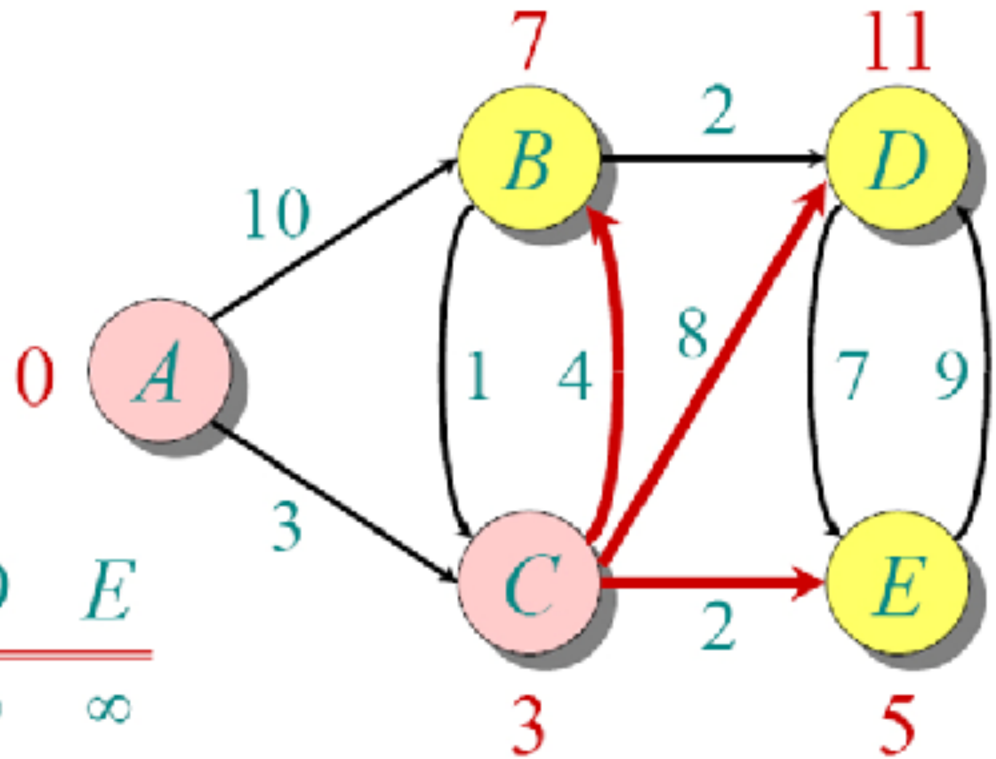
Another Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }

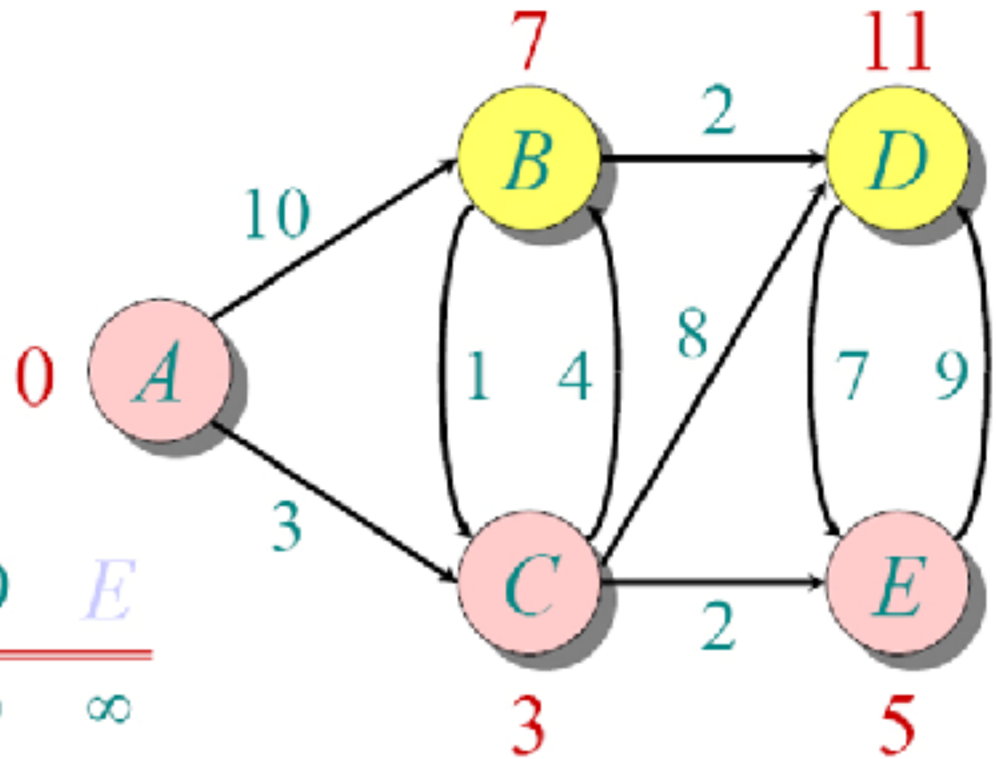
Another Example



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

S: { A, C }

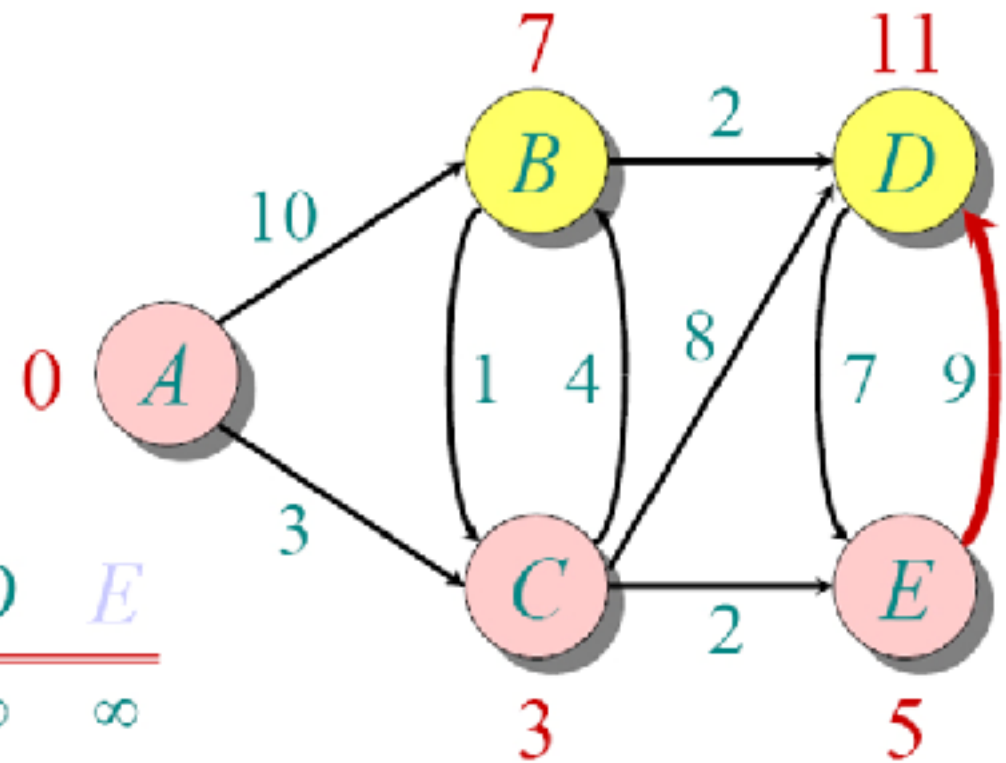
Another Example



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

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

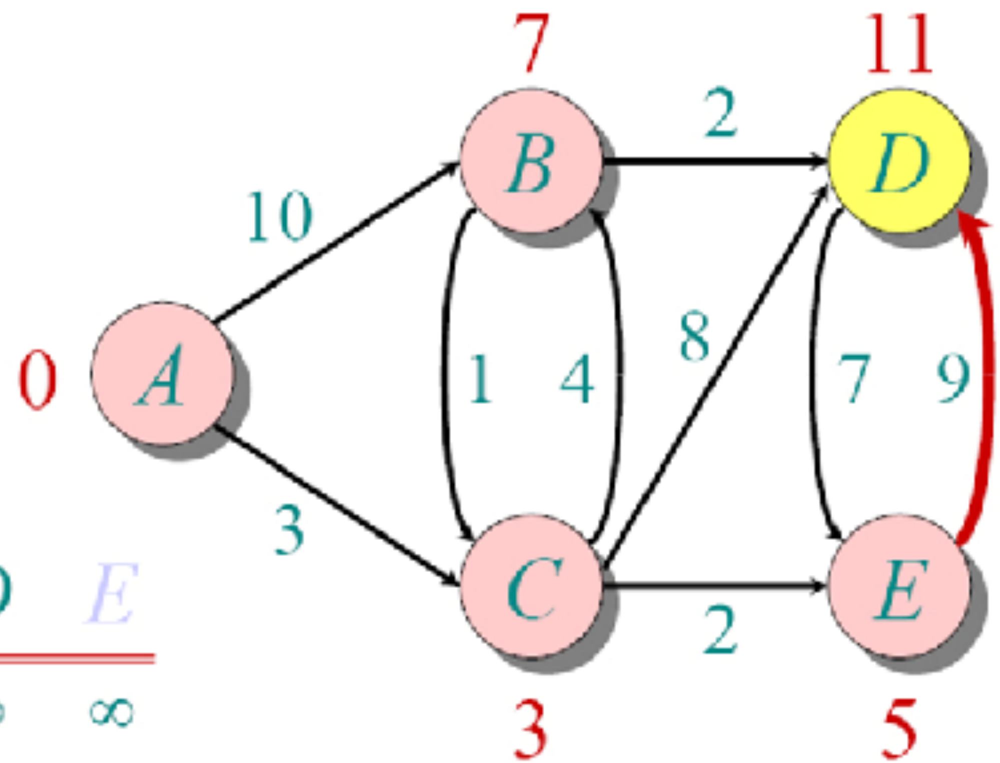
Another Example



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

S: { A, C, E }

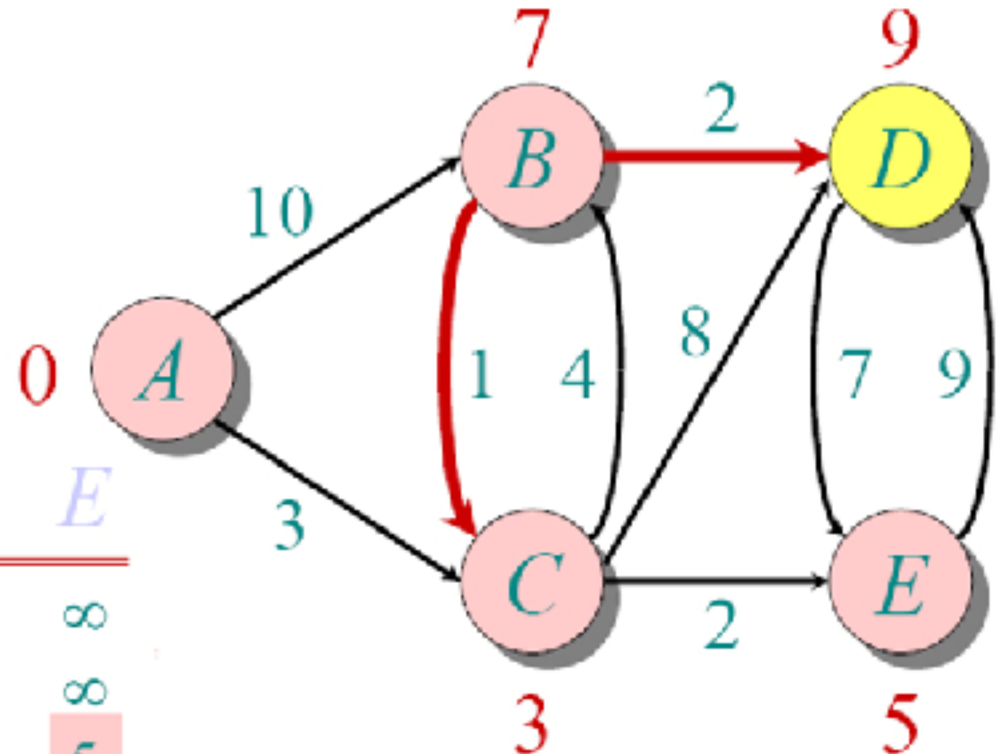
Another Example



Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	$\boxed{5}$
		$\boxed{7}$		11	

S: { A, C, E, B }

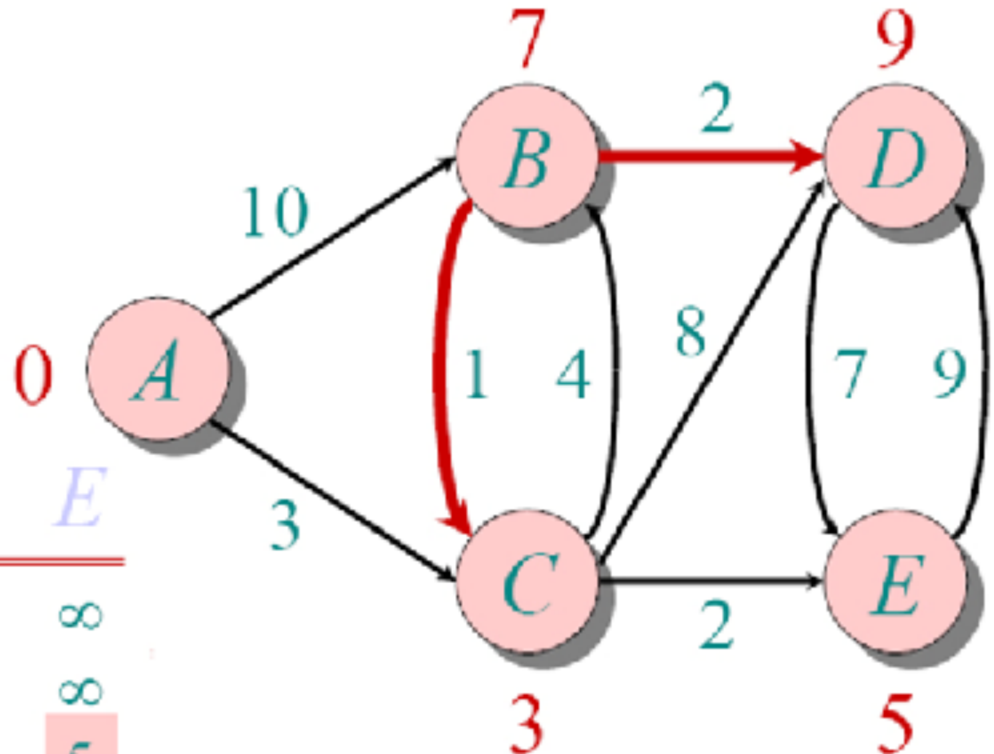
Another Example



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
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	$\boxed{5}$
		$\boxed{7}$		11	
				$\boxed{9}$	

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

Is Dijkstra’s algorithm correct?

- Invariant: For every vertex removed from the heap, dist[v] is the actual shortest distance from s to v
  - The only time a vertex gets visited is when the distance from s to that vertex is smaller than the distance to any remaining vertex
  - Therefore, there cannot be any other path that hasn’t been visited already that would result in a shorter path

Bounding the distance

- Another invariant: For each vertex v, dist[v] is an upper bound on the actual shortest distance
  - start of at ∞
  - only update the value if we find a shorter distance
- An update procedure

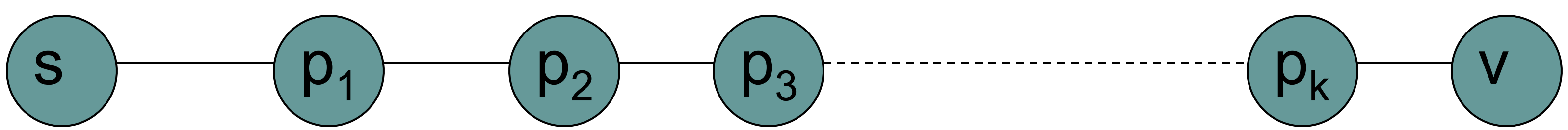
$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- Can we ever go wrong applying this update rule?
  - We can apply this rule as many times as we want and will never underestimate dist[v]
- When will dist[v] be right?
  - If u is along the shortest path to v and dist[u] is correct

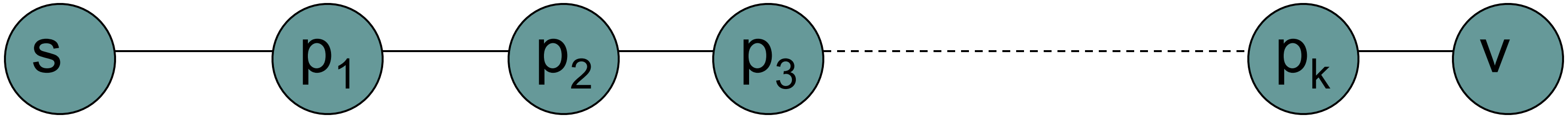
$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- Consider the shortest path from s to v



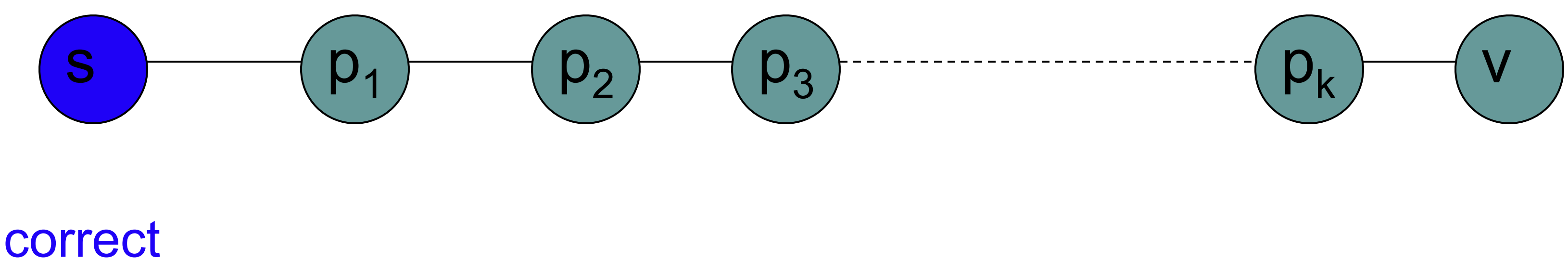
$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- What happens if we update all of the vertices with the above update?



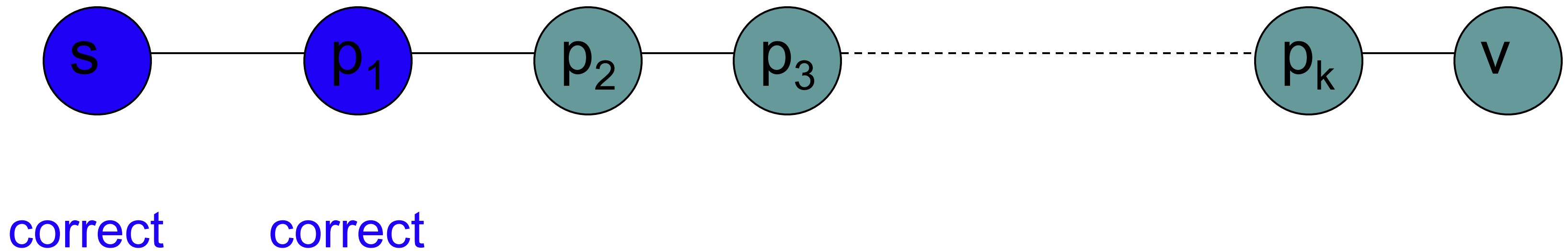
$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- What happens if we update all of the vertices with the above update?



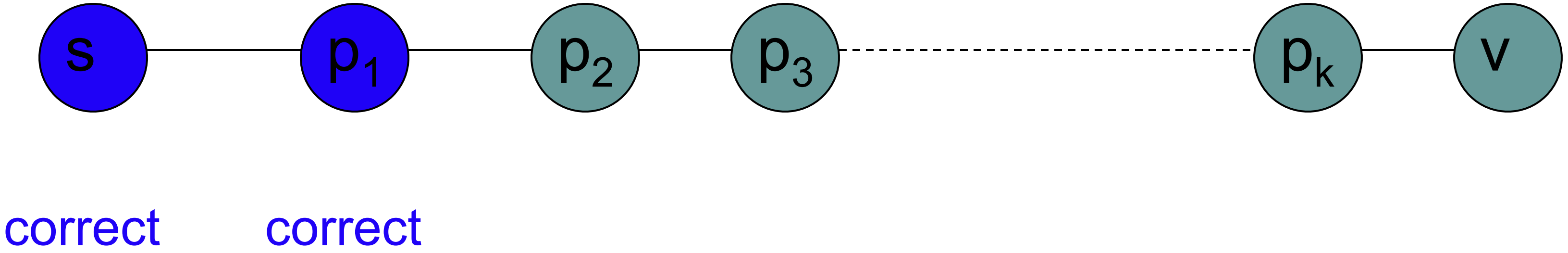
$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- What happens if we update all of the vertices with the above update?



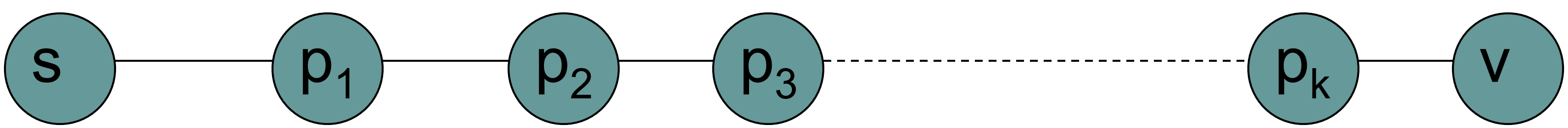
$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- Does the order that we update the vertices matter?



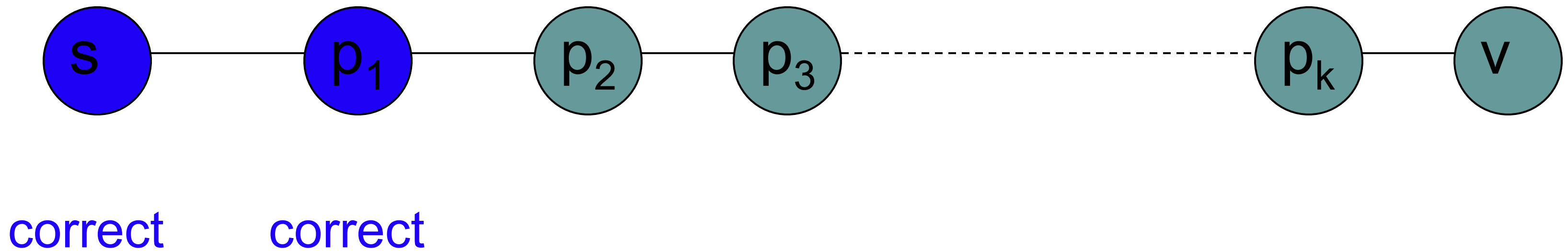
$$dist[v] = \min\{dist[v], dist[u] + w(u, v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- How many times do we have to do this for vertex p_i to have the correct shortest path from s?
  - i times



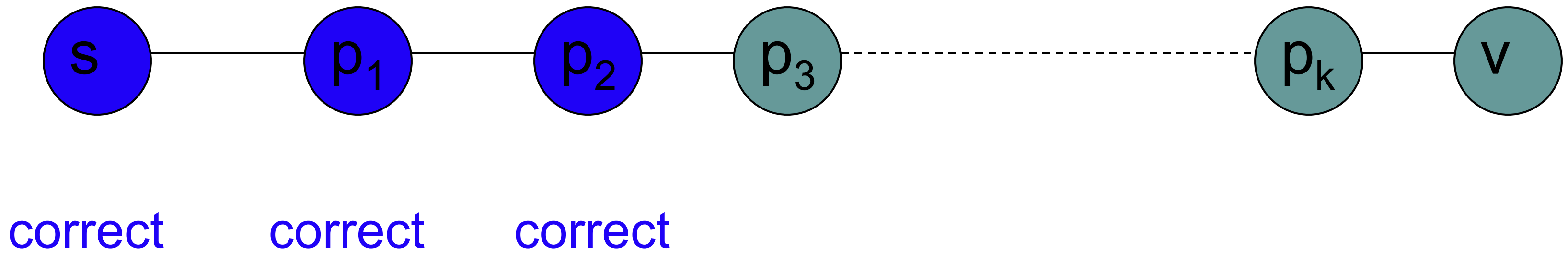
$$dist[v] = \min\{dist[v], dist[u] + w(u, v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- How many times do we have to do this for vertex p_i to have the correct shortest path from s?
  - i times



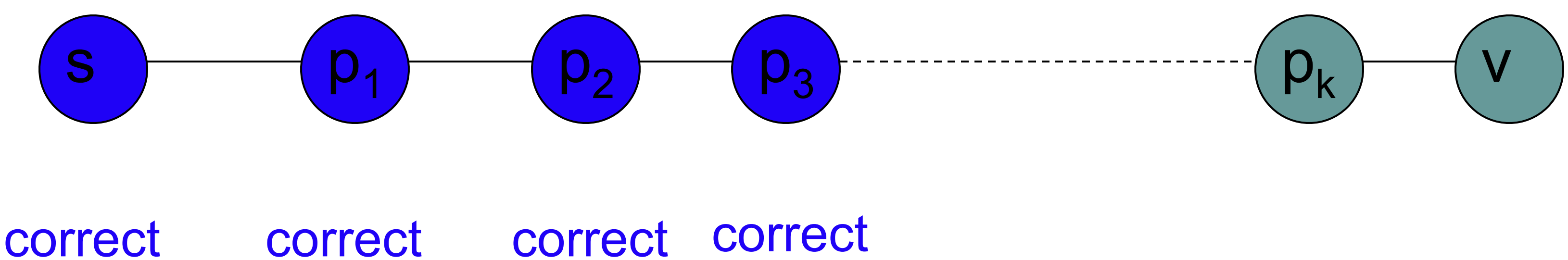
$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- How many times do we have to do this for vertex p_i to have the correct shortest path from s?
  - i times



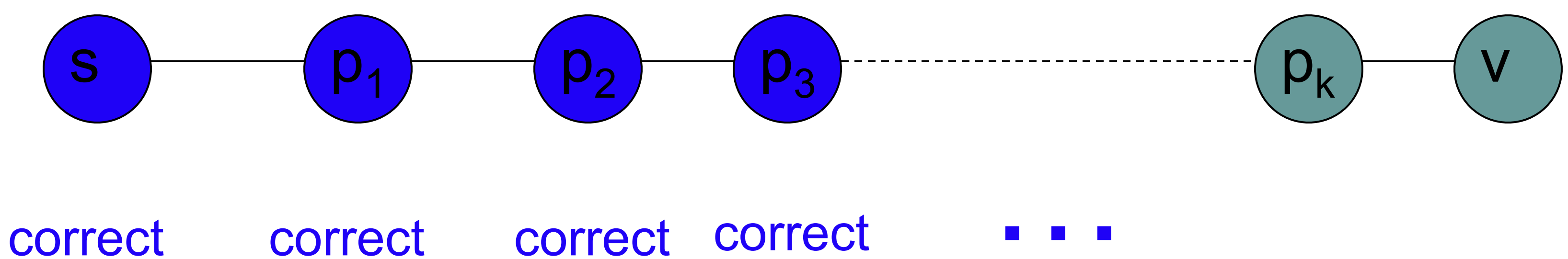
$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- How many times do we have to do this for vertex p_i to have the correct shortest path from s?
  - i times



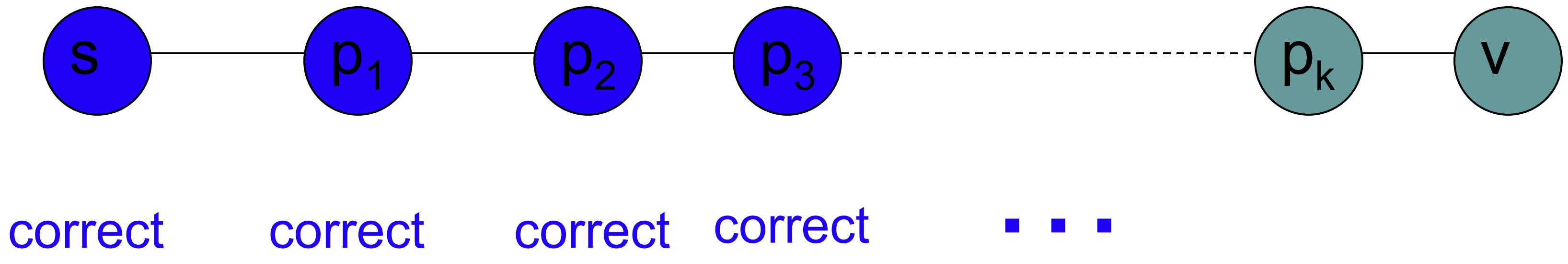
$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- How many times do we have to do this for vertex p_i to have the correct shortest path from s?
  - i times



$$dist[v] = \min\{dist[v], dist[u] + w(u,v)\}$$

- dist[v] will be right if u is along the shortest path to v and dist[u] is correct
- What is the longest (vetex-wise) the path from s to any node v can be?
  - |V| - 1 edges/vertices



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