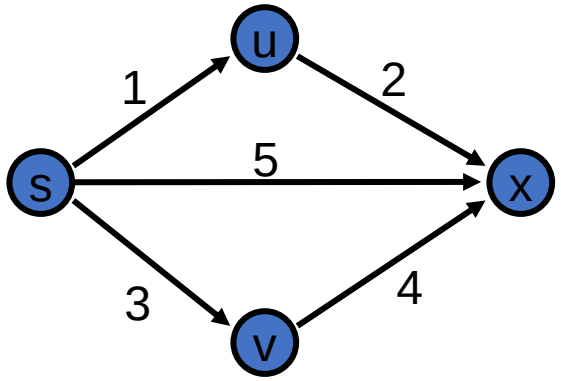
Lecture on

Single Source Shortest Path

Single Source Shortest Path

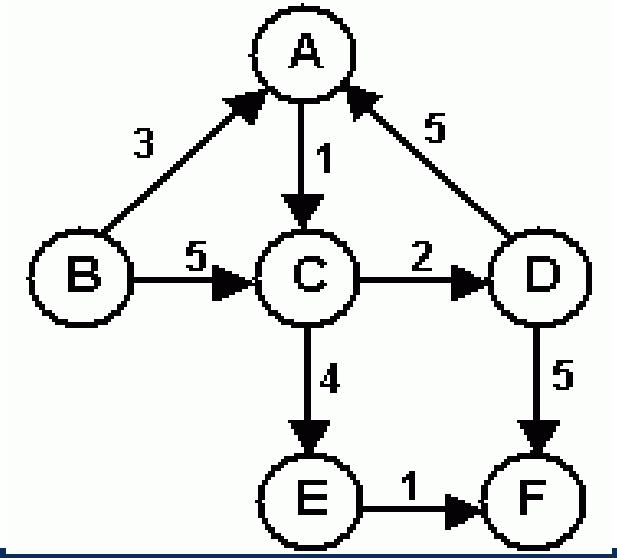
Given a graph and a start vertex s

- Determine distance of every vertex from s
- Identify shortest paths to each vertex
  - Express concisely as a “shortest paths tree”
  - Each vertex has a pointer to a predecessor on shortest path



Is the shortest path problem well defined?

- If all the edges in a graph have non-negative weights, then it is possible to find the shortest path from any two vertices.
- For example, in the figure below, the shortest path from B to F is { B, A, C, E, F } with a total cost of nine.
- Thus, the problem is well defined for a graph that contains non-negative weights.



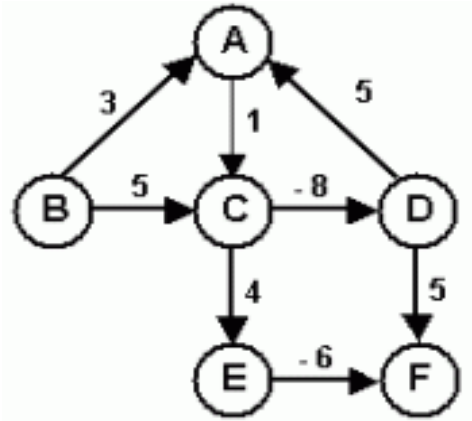
Is the shortest path problem well defined?

Things get difficult for a graph with negative weights.

For example, the path D, A, C, E, F costs 4 even though the edge (D, A) costs 5 -- the longer the less costly.

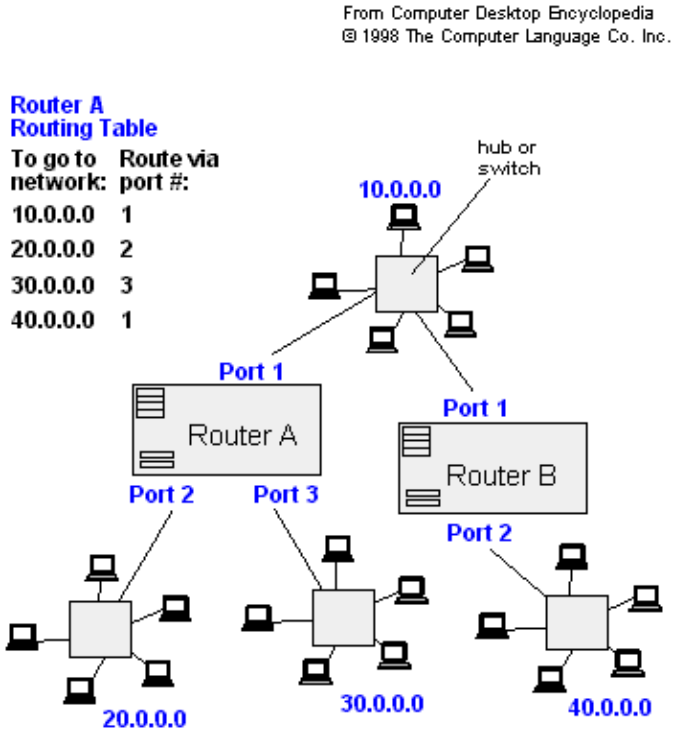
The problem gets even worse if the graph has a negative cost cycle. e.g. {D, A, C, D}

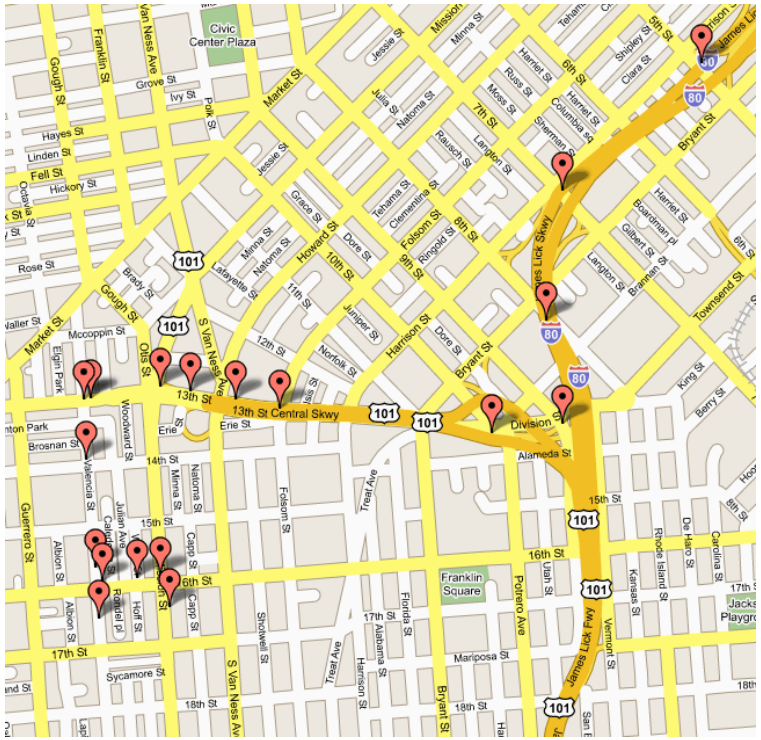
A solution can be found even for negative-weight graphs (**Bellman Ford Algorithm**) but not for graphs involving negative cost cycles.



Application

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





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∈V, such that all edge weights are nonnegative

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

Dijkstra's Algorithm : Initialization

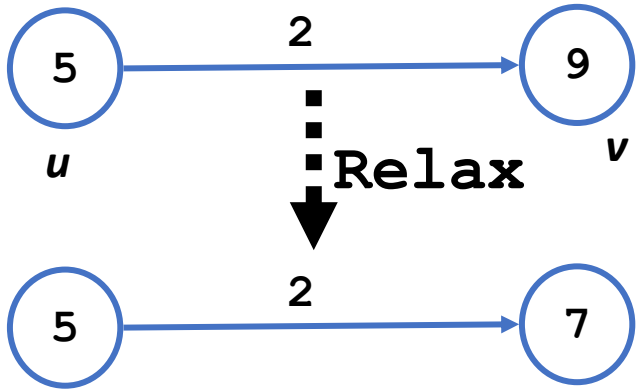
Algorithms keep track of d[v], π[v]. **Initialized** as follows:

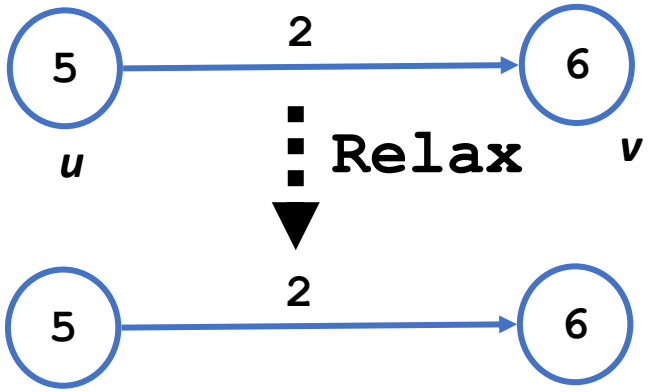
```
Initialize(G, s)
    for each v ∈ V[G] do
        d[v] := ∞;
        π[v] := NIL
    od;
    d[s] := 0
```

Dijkstra's Algorithm : Relaxation

- Maintaining this shortest discovered distance d[*v*] is called **relaxation**:

```
Relax(u,v,w) {
    if (d[v] > d[u]+w) then
        d[v]=d[u]+w;
}
```





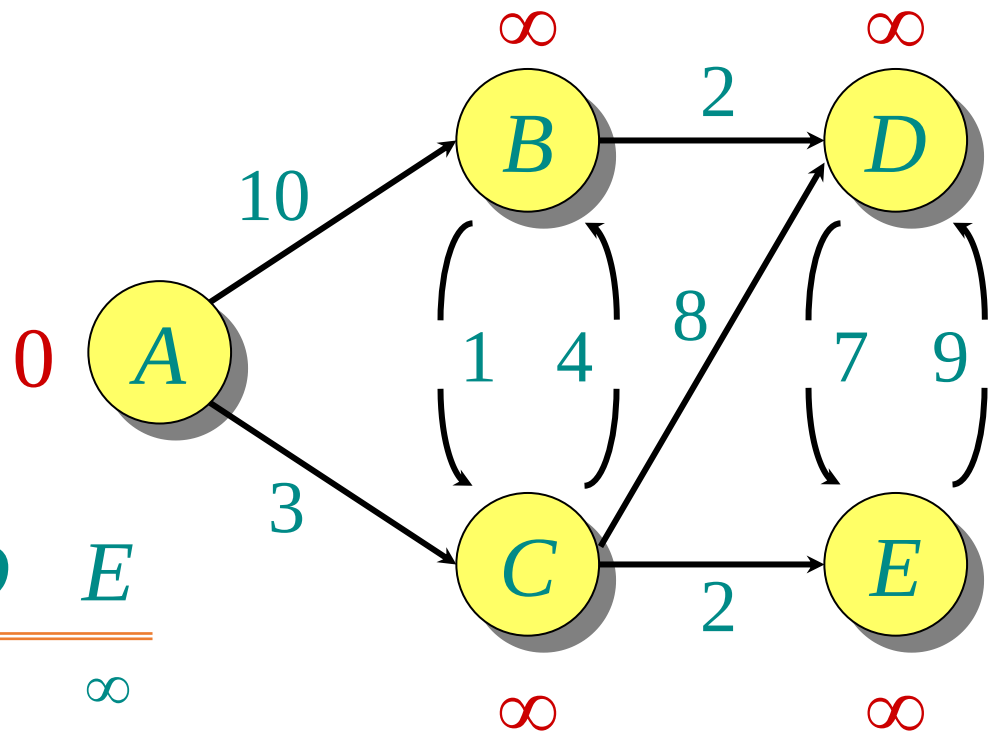
Dijkstra's Algorithm

```
DIJKSTRA(G, w, s)
 1 for each v ∈ V
 2       do d[v] ← ∞
 3 d[s] ← 0
 4 S ← ∅   ▷ Set of discovered nodes
 5 Q ← V
 6 while Q ≠ ∅
 7       do u ← EXTRACT-MIN(Q)
 8          S ← S ∪ {u}
 9          for each v ∈ Adj[u]
10                do if d[v] > d[u] + w(u, v)
11                      then d[v] ← d[u] + w(u, v)
```

relaxing edges

Example of Dijkstra's algorithm

Initialize:

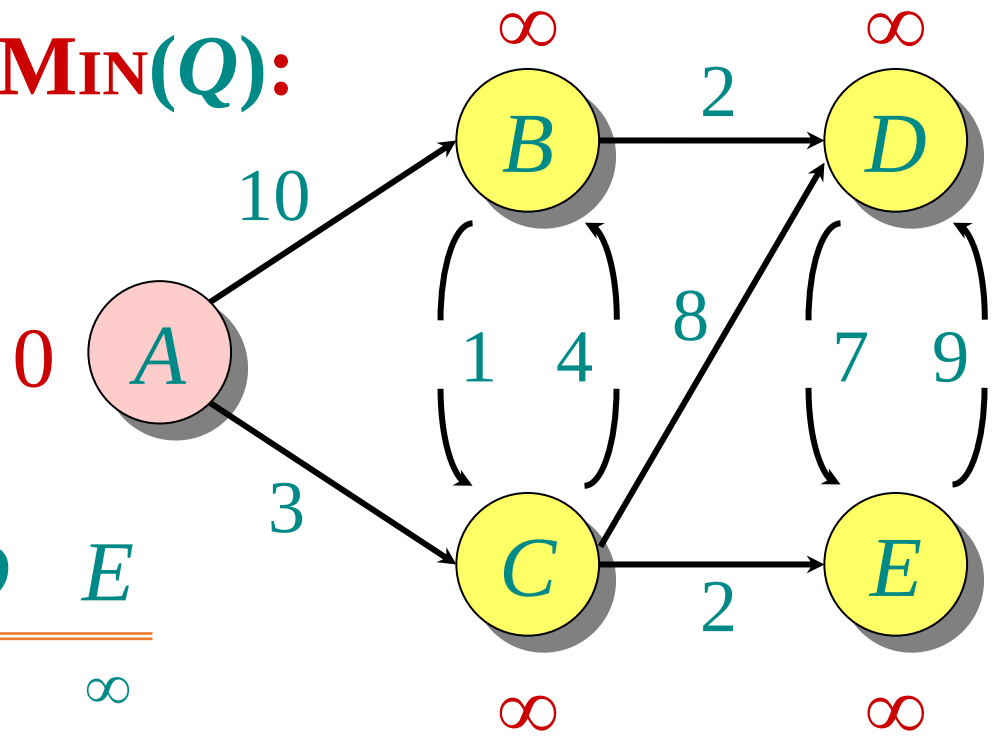


Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: {}

Example of Dijkstra's algorithm

"*A*" ← **EXTRACT-MIN**(*Q*):

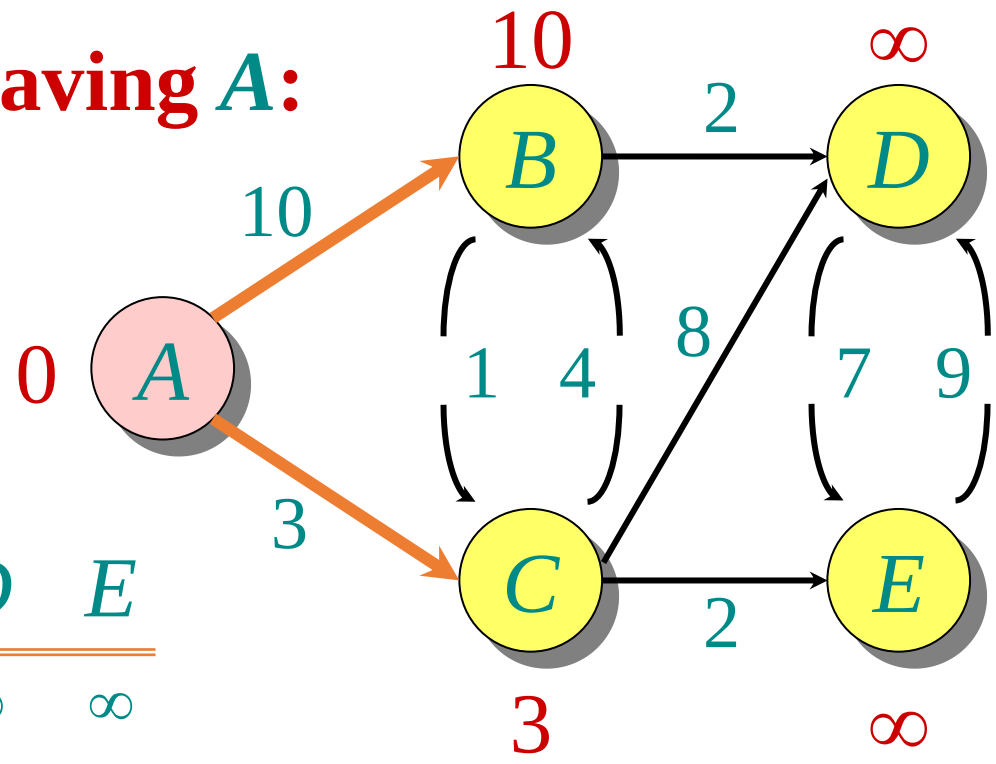


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞

S: { *A* }

Example of Dijkstra's algorithm

Relax all edges leaving *A*:

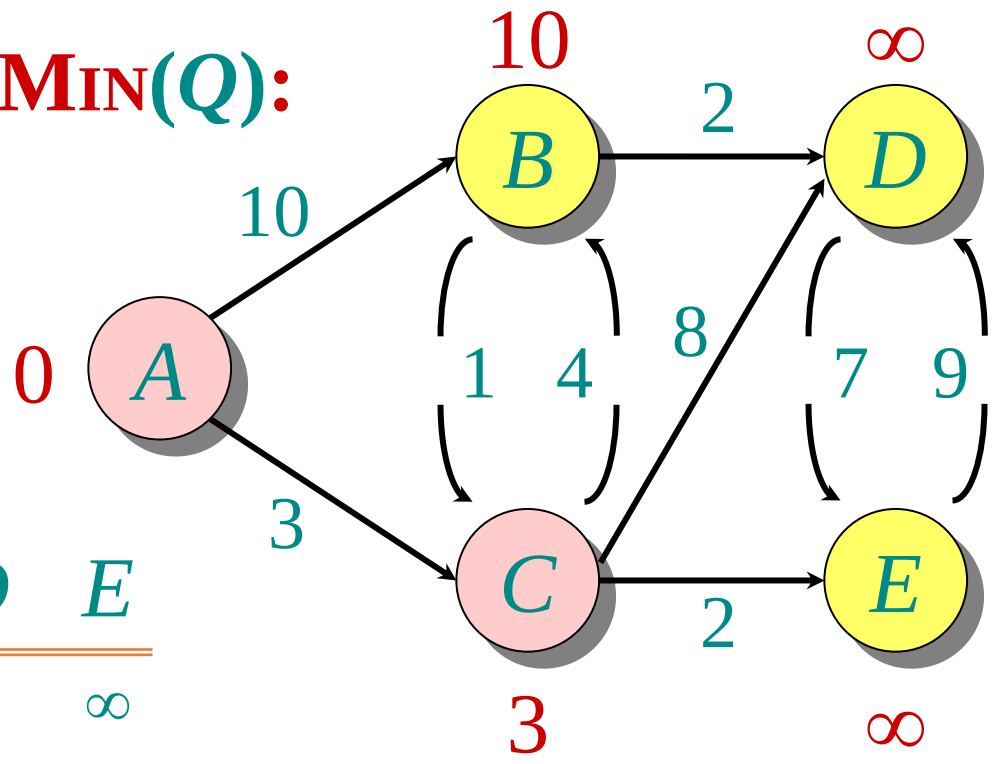


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { *A* }

Example of Dijkstra's algorithm

"*C*" ← **EXTRACT-MIN**(*Q*):

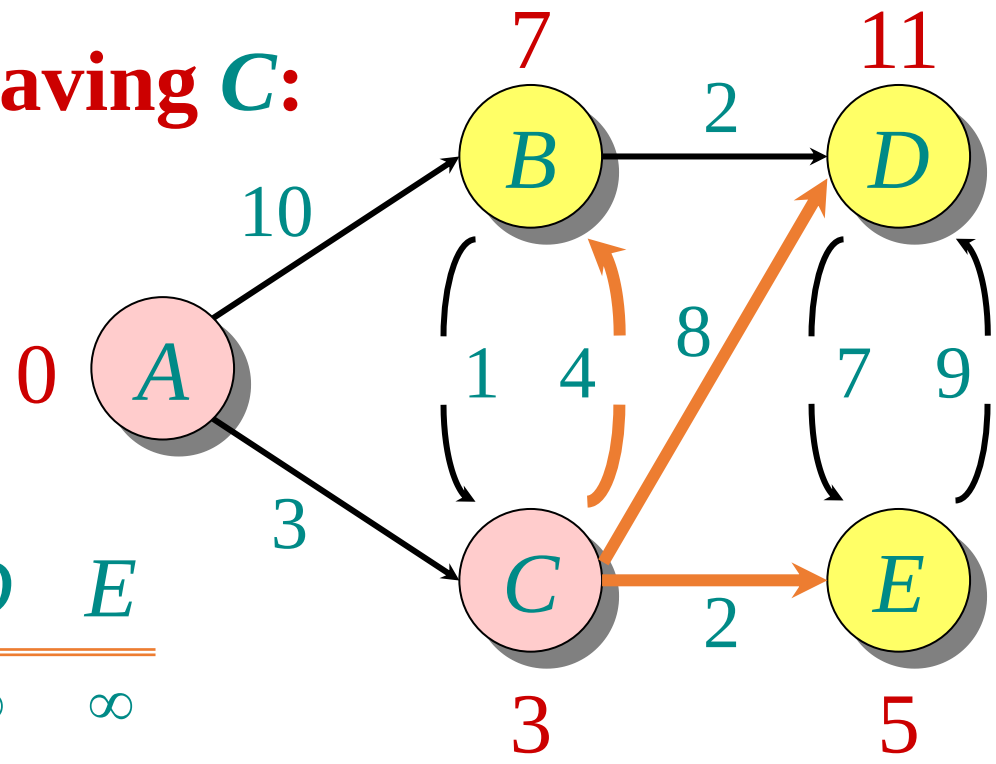


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { *A*, *C* }

Example of Dijkstra's algorithm

Relax all edges leaving *C*:

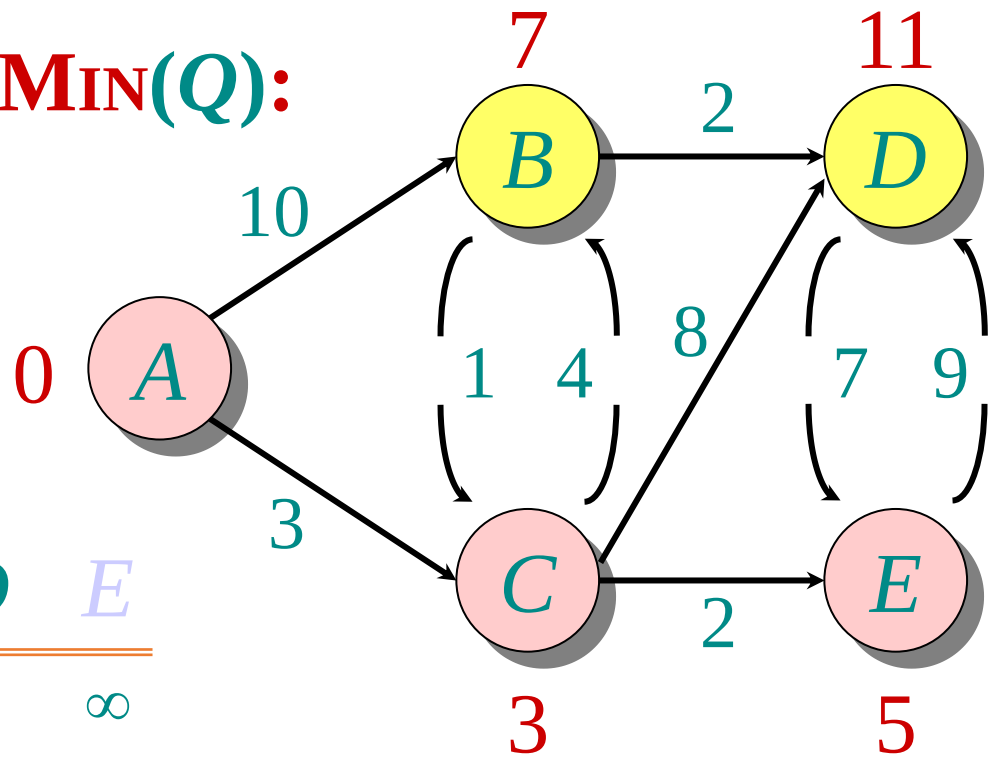


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { *A*, *C* }

Example of Dijkstra's algorithm

"E" ← **EXTRACT-MIN(*Q*):**

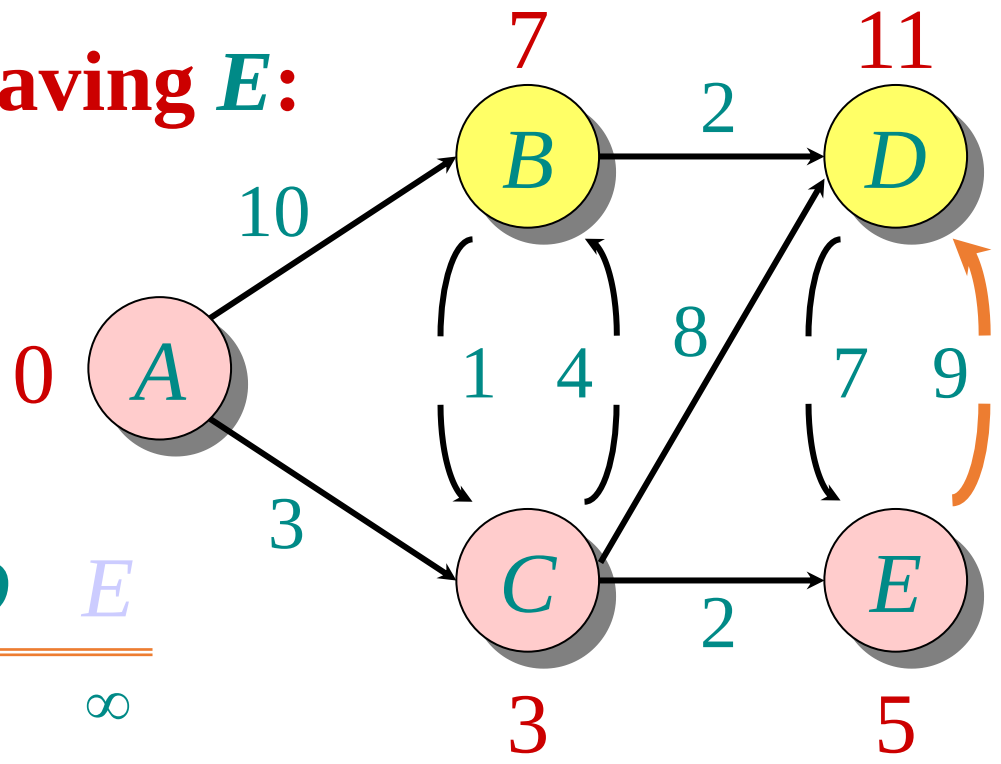


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

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

Example of Dijkstra's algorithm

Relax all edges leaving *E*:

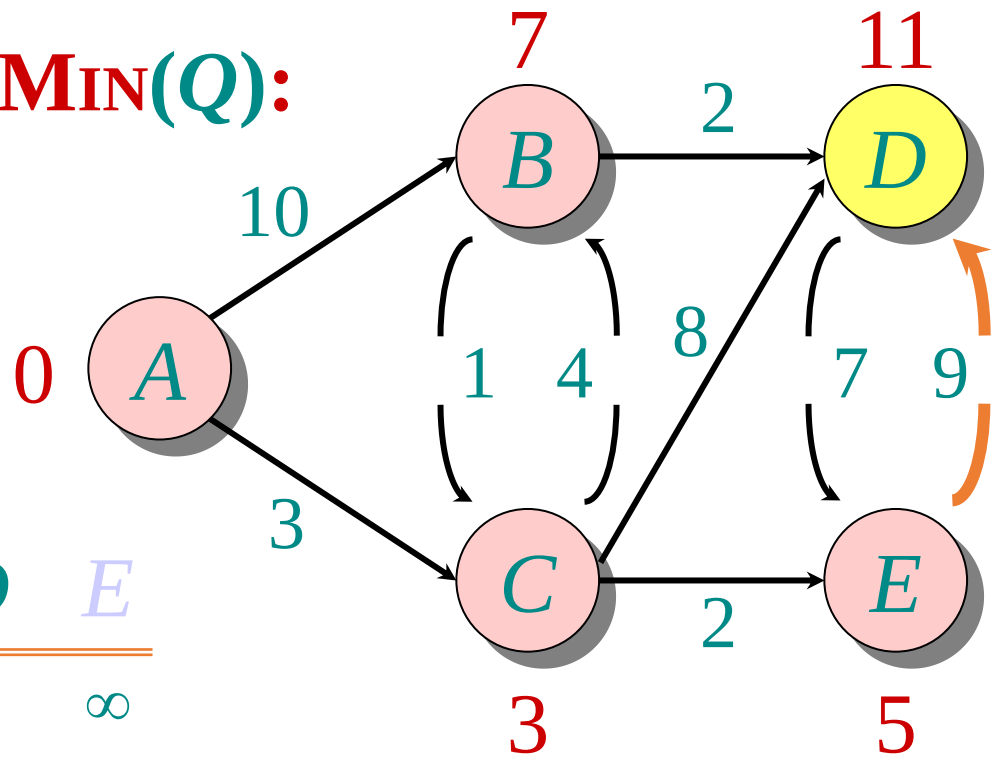


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

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

Example of Dijkstra's algorithm

"B" ← **EXTRACT-MIN(*Q*):**

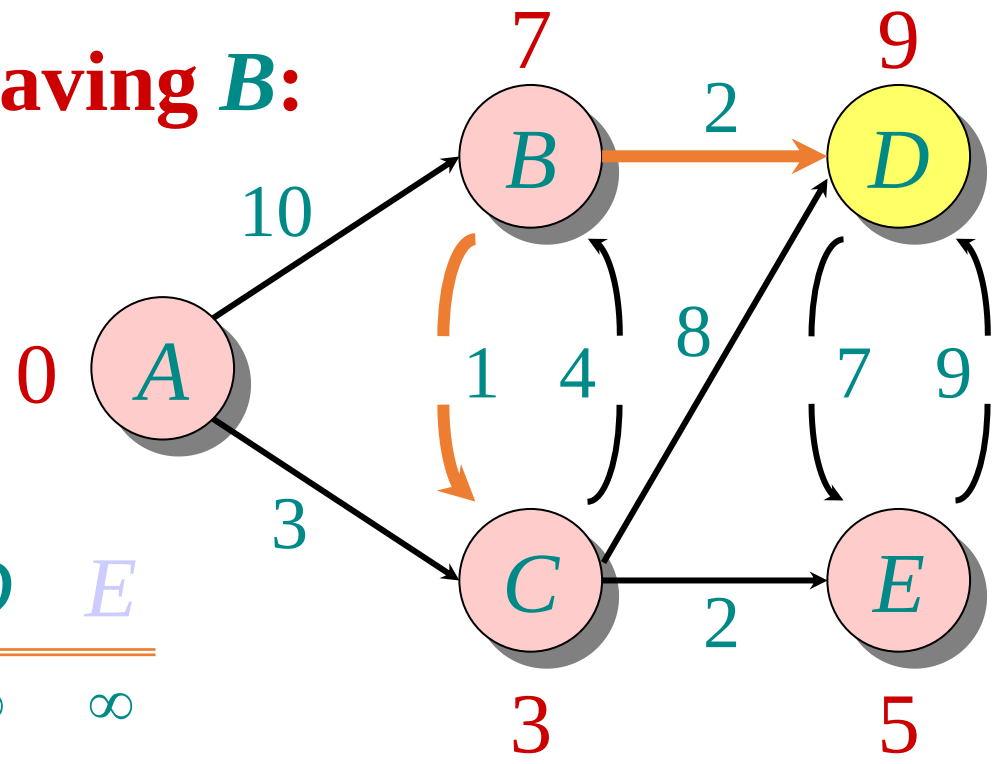


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

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

Example of Dijkstra's algorithm

Relax all edges leaving *B*:

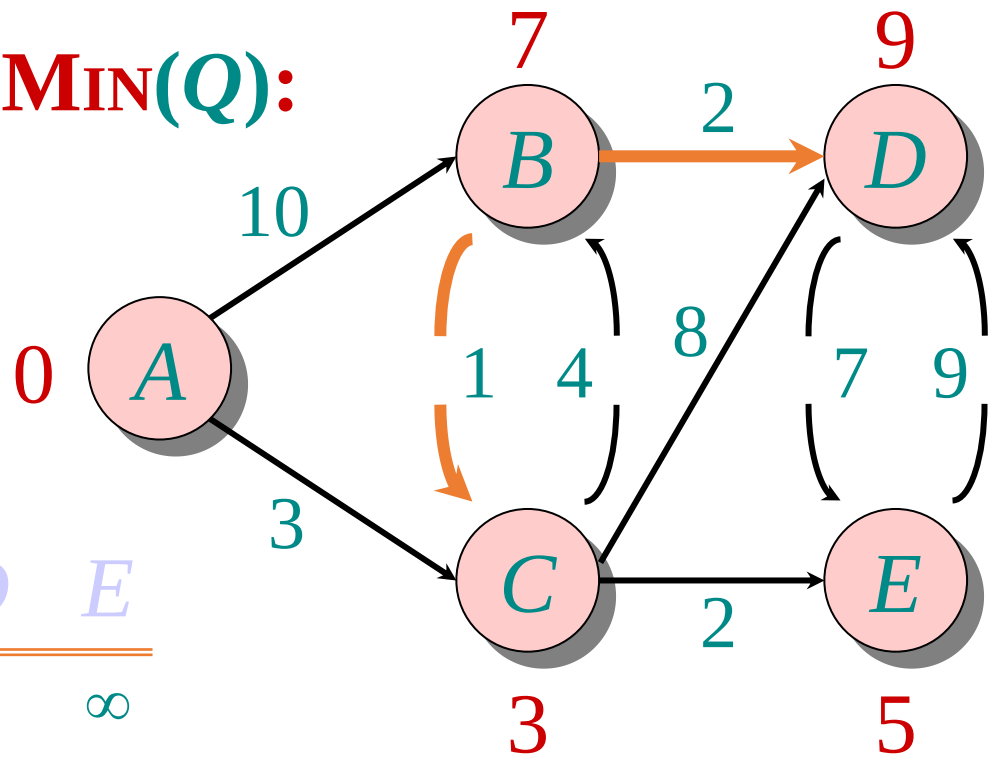


Q:	*A*	*B*	*C*	*D*	*E*
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

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

Example of Dijkstra's algorithm

"*D*" ← EXTRACT-MIN(*Q*):



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

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

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

Thank You