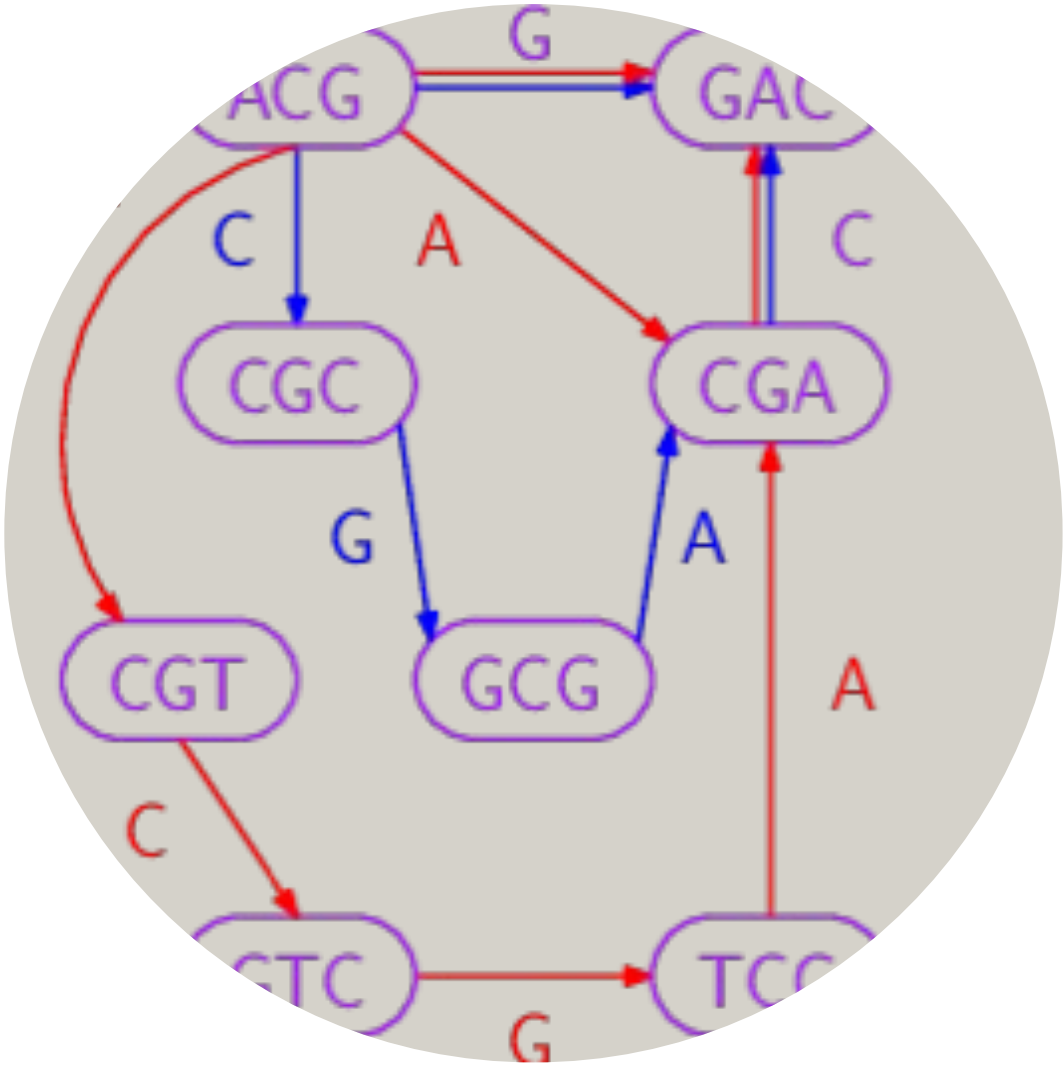
Graph algorithms

BioSB Algorithms for Genomics

12-10-2023

Sandra Smit

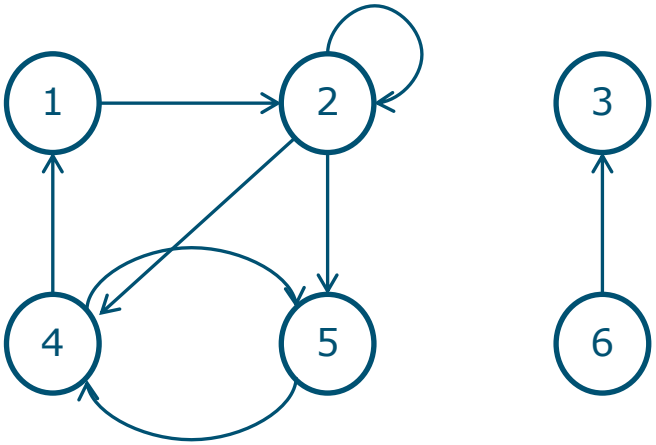


Terminology

- Graph, vertices, edges
- Directed versus undirected graphs
- Degree(v) of a vertex, indegree(v), outdegree(v)
- Connected versus disconnected graphs
- Complete graphs
- Weighted graphs
- Trees

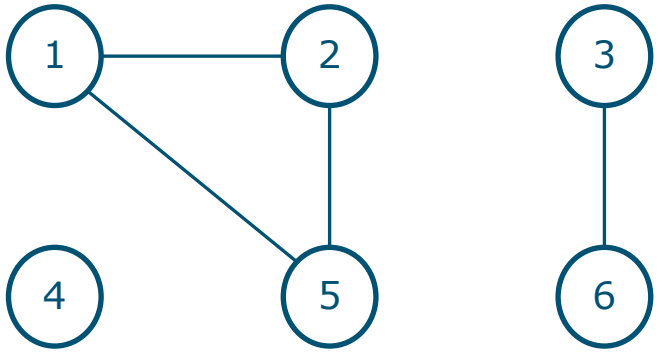
Directed graph

- A directed graph G is a pair (V,E), where V is a finite set and E is a binary relation on V.
  - V: vertex set, with vertices as elements
  - E: edge set, with edges as elements
- Self-loops are possible
- Vertex set {1,2,3,4,5,6}



Undirected graph

- In an undirected graph G=(V,E), the edge set E consists of unordered pairs of vertices, rather than ordered pairs
  - Edge (u,v) where u and v are elements of V and u ≠ v
  - (u,v) is considered to be the same as (v,u)
- Self-loops are forbidden



Incident and adjacent

- If (u,v) is an edge in an undirected graph G=(V,E), we say that (u,v) is incident on vertices u and v.
- In a directed graph:
  - Leaving edge: incident from
  - Entering edge: incident to
- If (u,v) is an edge in a graph G=(V,E), we say that vertex v is adjacent to vertex u
- In an undirected graph, the adjacency relation is symmetric (but not necessarily in a directed graph!)
- In a directed graph G=(V,E), a neighbor of a vertex u is any vertex that is adjacent to u in the undirected version of G

Degree

- The degree of a vertex in an undirected graph is the number of edges incident on it.
- In a directed graph:
  - out-degree of a vertex = the number of edges leaving it
  - in-degree = the number of edges entering it
  - degree = in-degree + out-degree

Balanced, semi-balanced

- A vertex v in a graph is balanced if the number of edges entering v equals the number of edges leaving v
  - indegree(v) == outdegree(v)
- A vertex v in a graph is called semibalanced if the indegree differs from the outdegree by 1
  - | indegree(v) – outdegree(v) | = 1

Paths and cycles

- A path of length k from a vertex u to a vertex u’ in a graph G=(V,E) is a sequence $<v_0, v_1, v_2, \ldots, v_k>$ of vertices such that $u=v_0$, $u' = v_k$, and (v_{i-1},v_i) is an element of *E* for $i = 1,2,\ldots,k$
- Length of the path = number of edges in the path
- In a directed graph, a path $<v_0, v_1, v_2, \ldots, v_k>$ forms a cycle if $v_0=v_k$ and the path contains at least one edge
- A graph with no cycles is acyclic

Connected and complete

- A graph is called connected if all pairs of vertices can be connected by a path, which is a continuous sequence of edges, where each successive edge begins where the previous one left off
- Disconnected graphs: graphs that are not connected
    - Can be partitioned into connected components
- A graph is called complete if there is an edge between every two vertices

Weighted

- Weighted graphs: ordered triple $G=(V,E,w)$ where w is a weight function defined for every edge e in E.
  - $w(e)$ is a number reflecting the weight of edge e

Tree

- A (free) tree is a connected, acyclic, undirected graph
- A forest is an undirected, acyclic, but disconnected graph

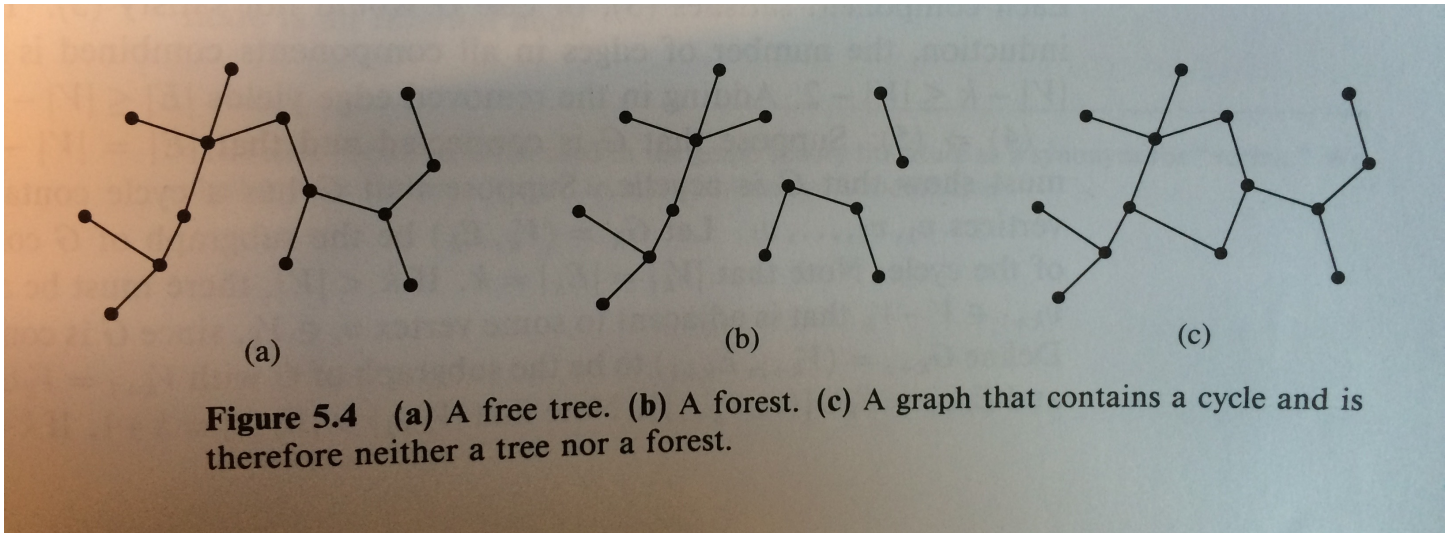


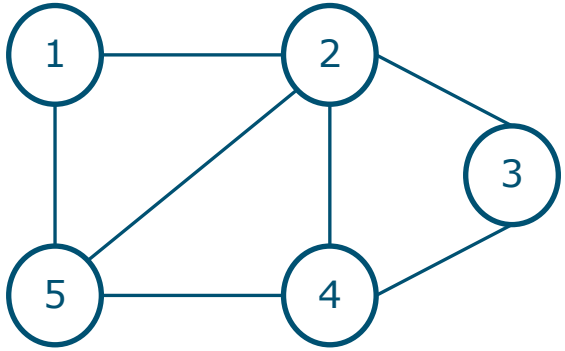
Figure 5.4 **(a)** A free tree. **(b)** A forest. **(c)** A graph that contains a cycle and is therefore neither a tree nor a forest.

Graph representations

- Collection of adjacency lists
  - Used for sparse graphs: $|E|$ much less than $|V|^2$
- Adjacency matrix
  - Used for dense graphs: $|E|$ close to $|V|^2$
  - Used when need to test quickly whether two vertices are connected

Adjacency list

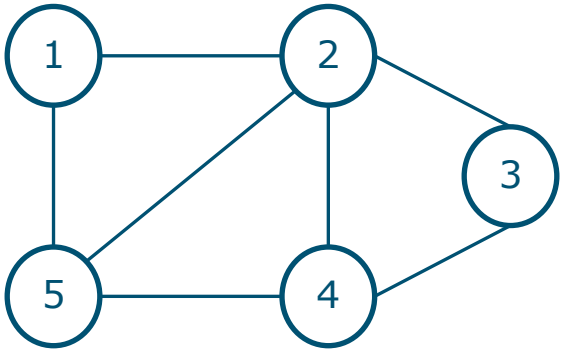
- Array Adj of |V| lists, one for each vertex in V
- For each u ∈ V, the adjacency list Adj[u] contains all vertices v such that there is an edge (u,v) ∈ E




```
{1:[2,5],
 2:[1,5,3,4],
 3:[2,4],
 4:[2,5,3],
 5:[4,1,2]}
```

Adjacency matrix

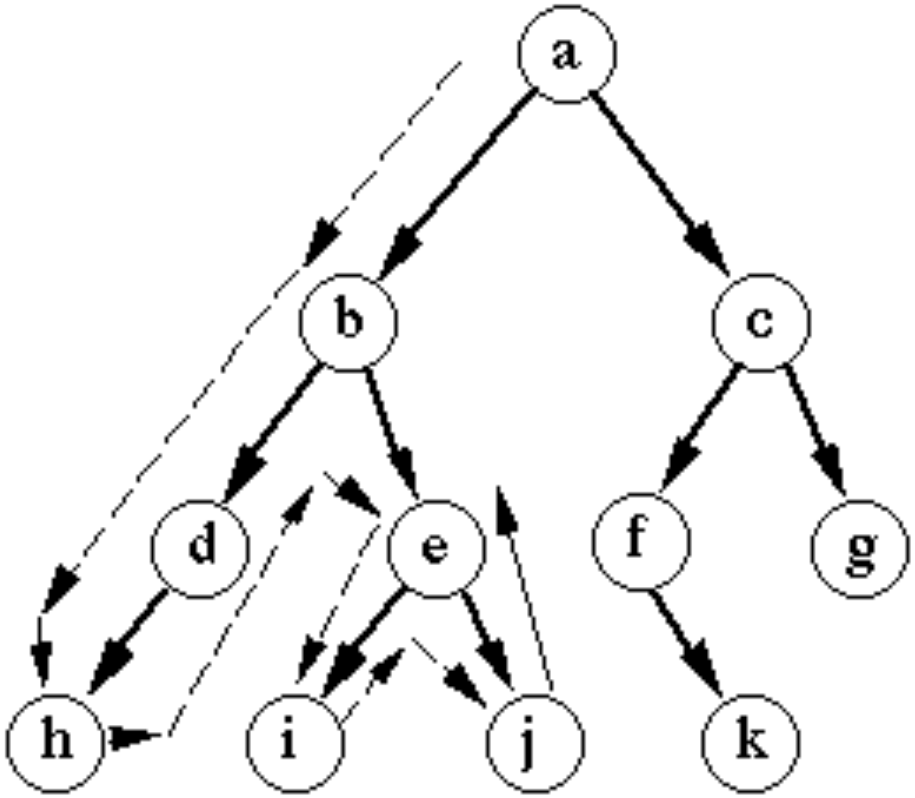
- Assumption: vertices are numbered 1,2,…,|V| in some arbitrary manner
- Graph *G* then consists of a |V| x |V| matrix A = (a_{ij}) such that
  - a_{ij} = 1 if $(i,j) \in E$
  - a_{ij} = 0 otherwise



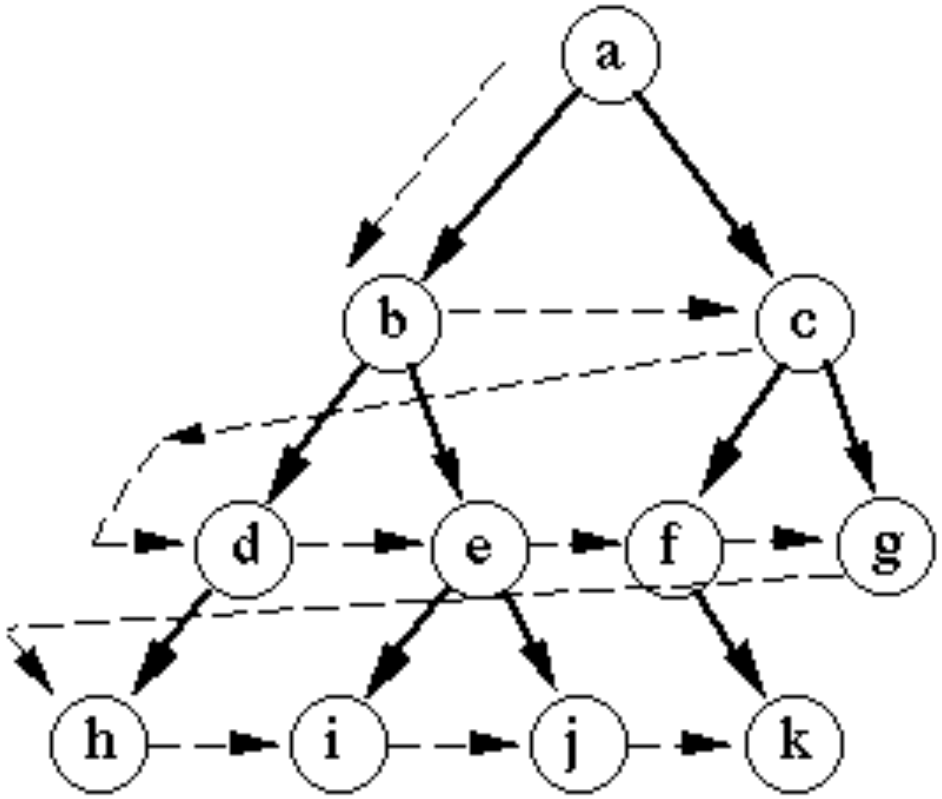
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Searching a graph

- Systematically following the edges of the graph so as to visit the vertices of the graph
- Breadth-first search
  - One of the simplest search algorithms
  - Expands the frontier between discovered and undiscovered vertices uniformly across the breath of the frontier
- Depth-first search
  - Searches deeper into the graph whenever possible



Depth-first search

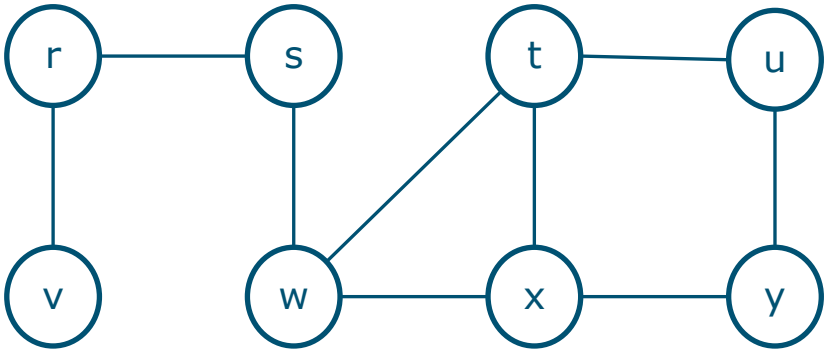


Breadth-first search

Breadth-first search (BFS)

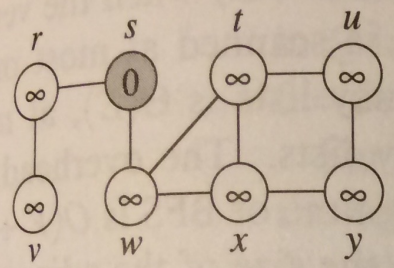
- Given a graph G=(V,E) and a distinguished source vertex , BFS systematically explores the edges of G to discover every vertex that is reachable from s.
- The algorithm discovers all vertices at distance k from s before discovering any vertices at distance k+1
- Uses white, gray, black vertices
    - White: undiscovered vertex
    - Gray: discovered for the first time
    - Black: discovered (done)

BSF



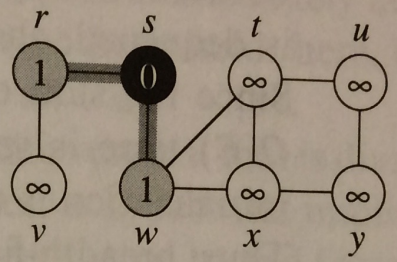

```
{r:[s,v],
 s:[w,r],
 t:[w,x,u],
 u:[t,y],
 v:[r],
 w:[s,t,x],
 x:[w,t,y],
 y:[x,u]
}
```

(a)



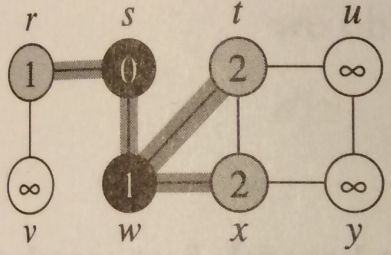
Q | s |
0

(b)



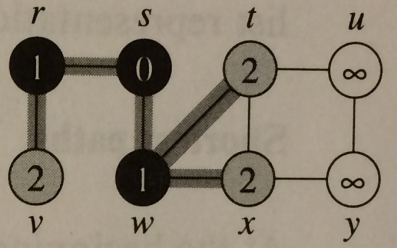
Q | w | r |
1 1

(c)



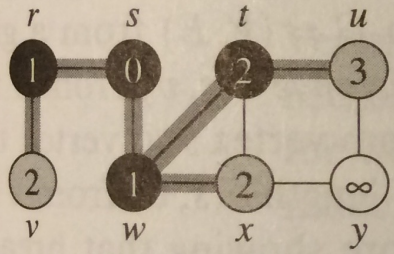
Q | r | t | x |
1 2 2

(d)



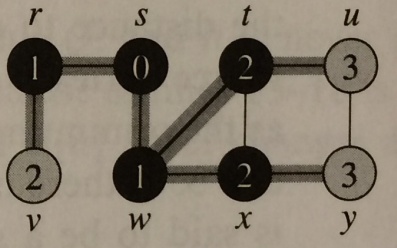
Q | t | x | v |
2 2 2

(e)



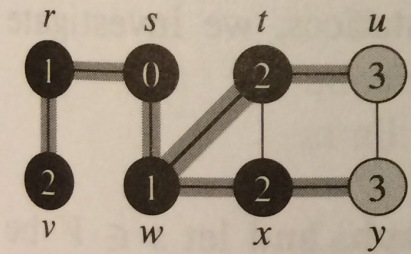
Q | x | v | u |
2 2 3

(f)



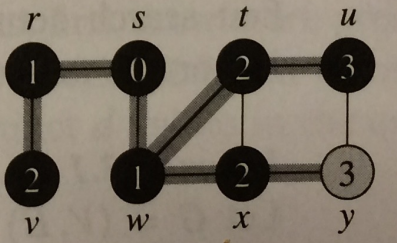
Q | v | u | y |
2 3 3

(g)



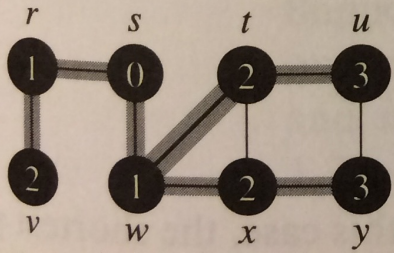
Q | u | y |
3 3

(h)



Q | y |
3

(i)



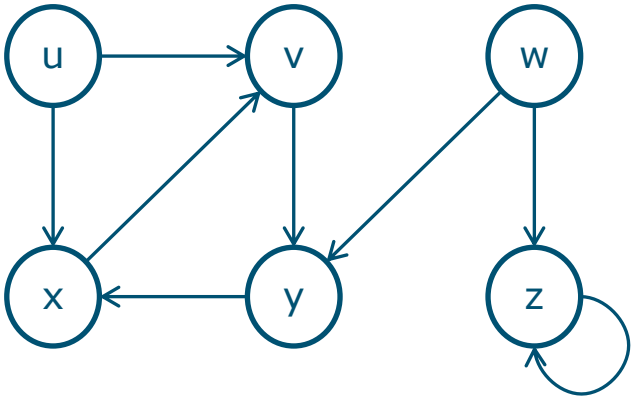
Q $\varnothing$

```
BFS(G, s)
 1  for each vertex u ∈ V[G] − {s}
 2       do color[u] ← WHITE
 3          d[u] ← ∞
 4          π[u] ← NIL
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← {s}
 9  while Q ≠ ∅
10       do u ← head[Q]
11          for each v ∈ Adj[u]
12               do if color[v] = WHITE
13                    then color[v] ← GRAY
14                         d[v] ← d[u] + 1
15                         π[v] ← u
16                         ENQUEUE(Q, v)
17          DEQUEUE(Q)
18          color[u] ← BLACK
```

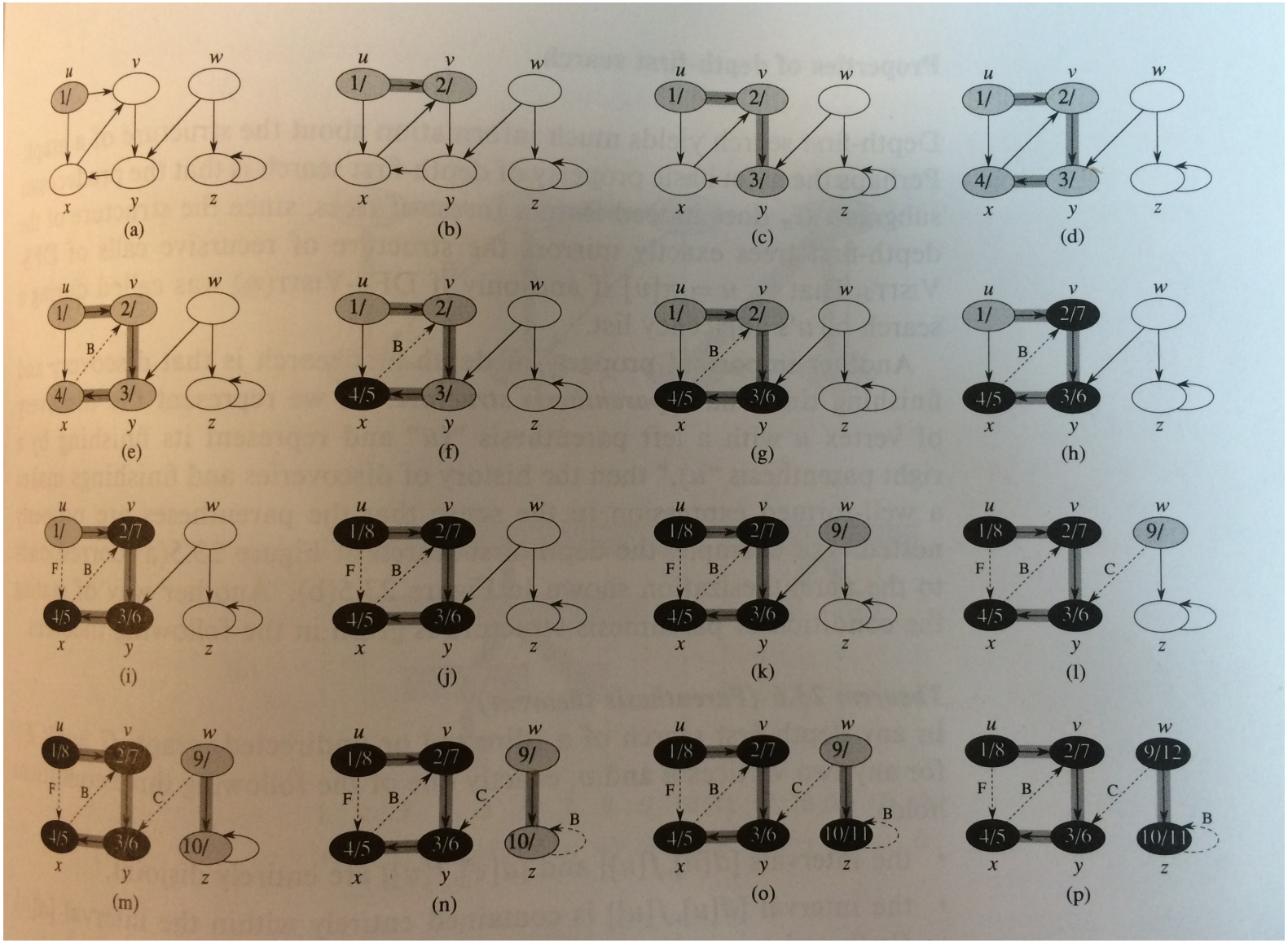
Depth-first search (DFS)

- In DFS, edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it.
- Uses white, gray, black vertices
  - White: undiscovered vertex
  - Gray: discovered for the first time
  - Black: finished (its adjacency list has been examined completely)
- Vertices are also timestamped: one for discovery, one for finishing

DSF




```
{u:[v,x],
 v:[y],
 w:[y,z],
 x:[v],
 y:[x],
 z:[z]
}
```



Graph challenges

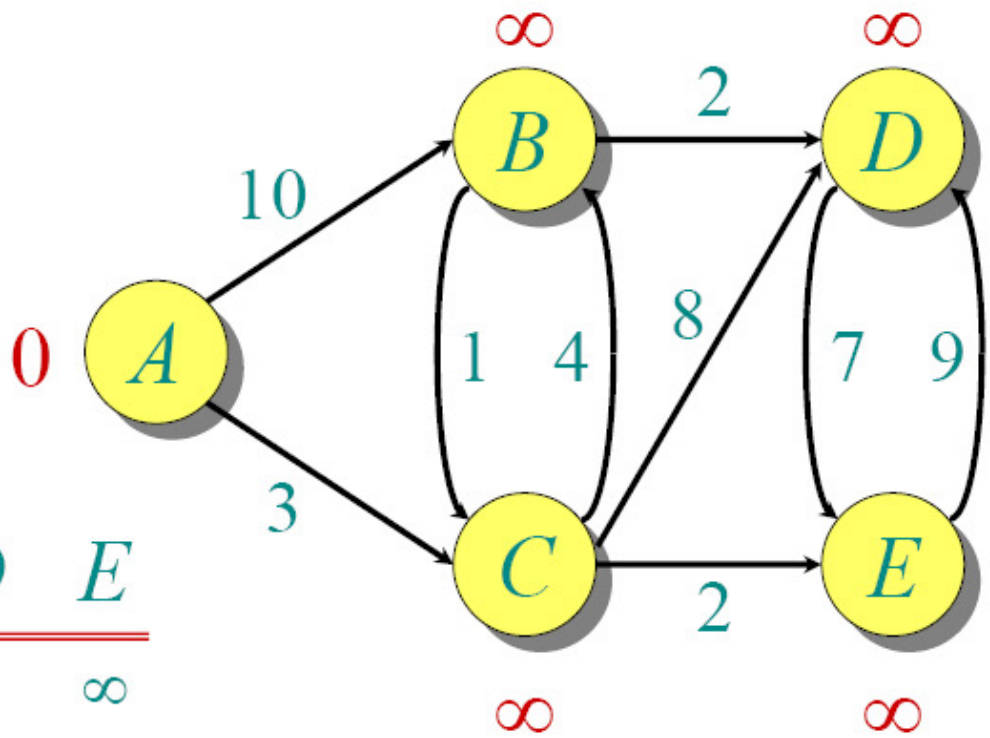
- Shortest Path Problem: given a weighted graph and two vertices, find the shortest distance between them
- Hamiltonian Cycle problem: find a cycle in a graph that visits every vertex exactly once.
- Eulerian Cycle Problem: find cycle in graph that visits each edge exactly once

Shortest path

- Dijkstra's algorithm: solution to the single source shortest path problem
  - Works for directed and undirected graphs
  - All edges must have nonnegative weights
  - Graphs must be connected
- Calculate the shortest path between one node (you pick which one) and every other node in the graph

Dijkstra Animated Example

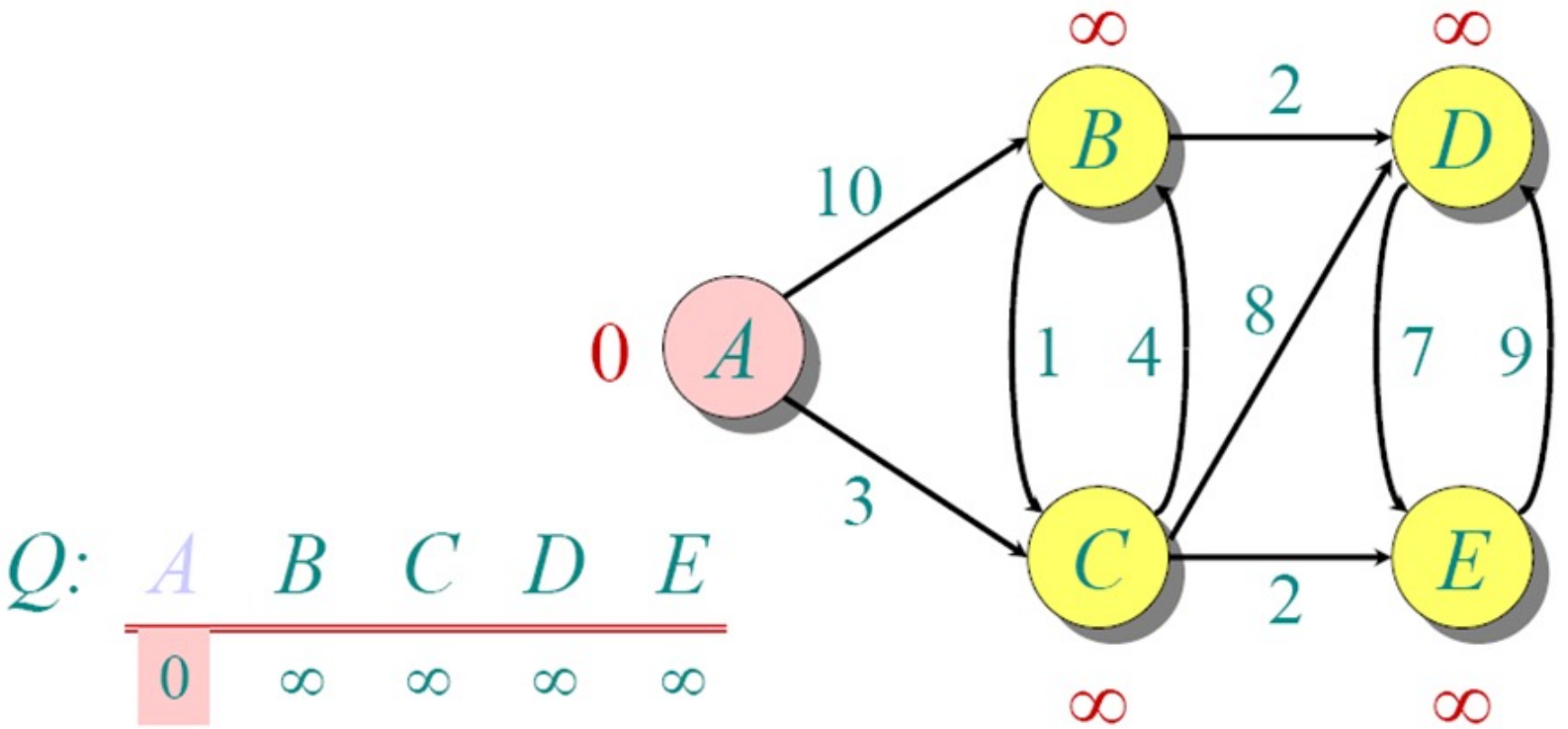
Initialize:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

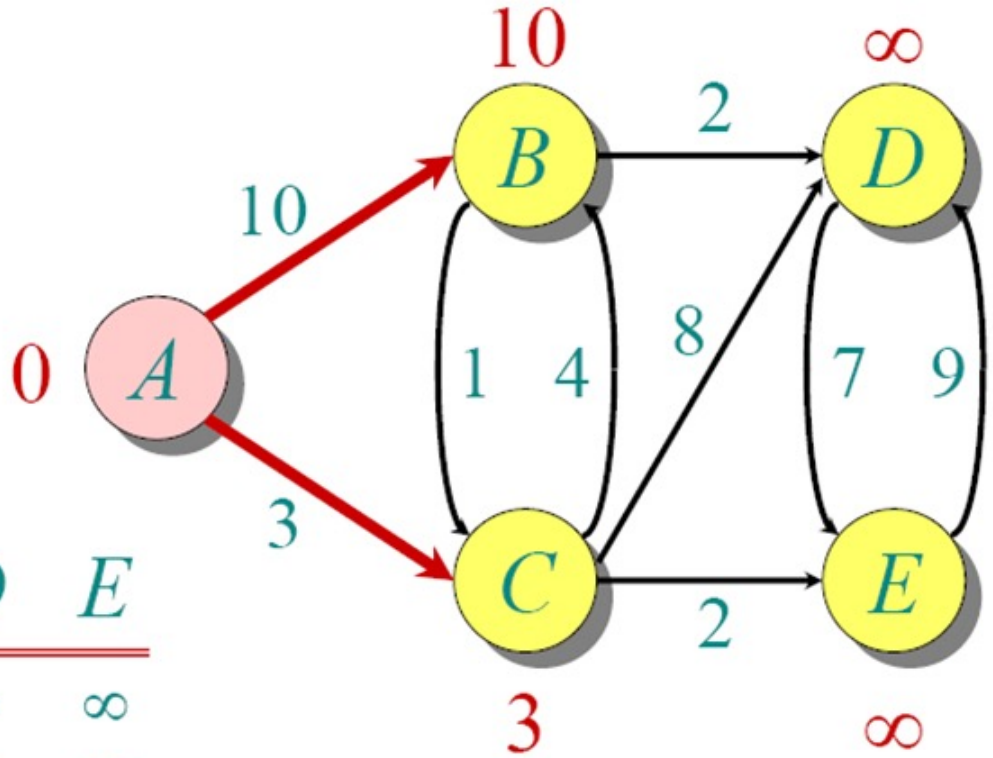
S: {}

Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

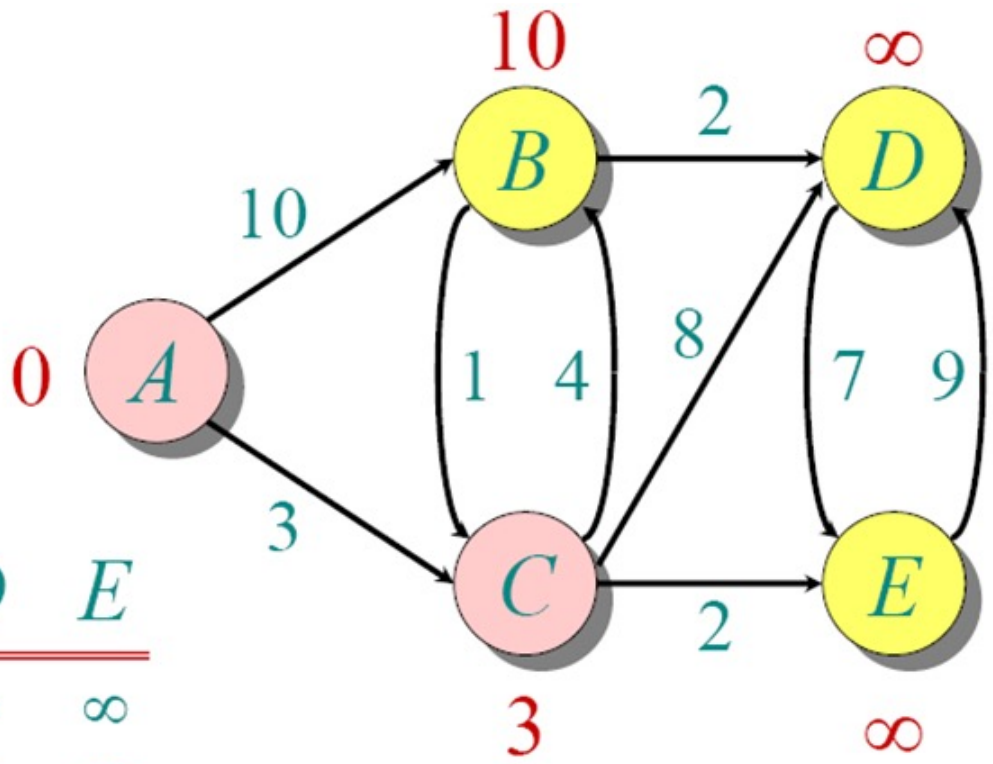
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

$S: \{ A \}$

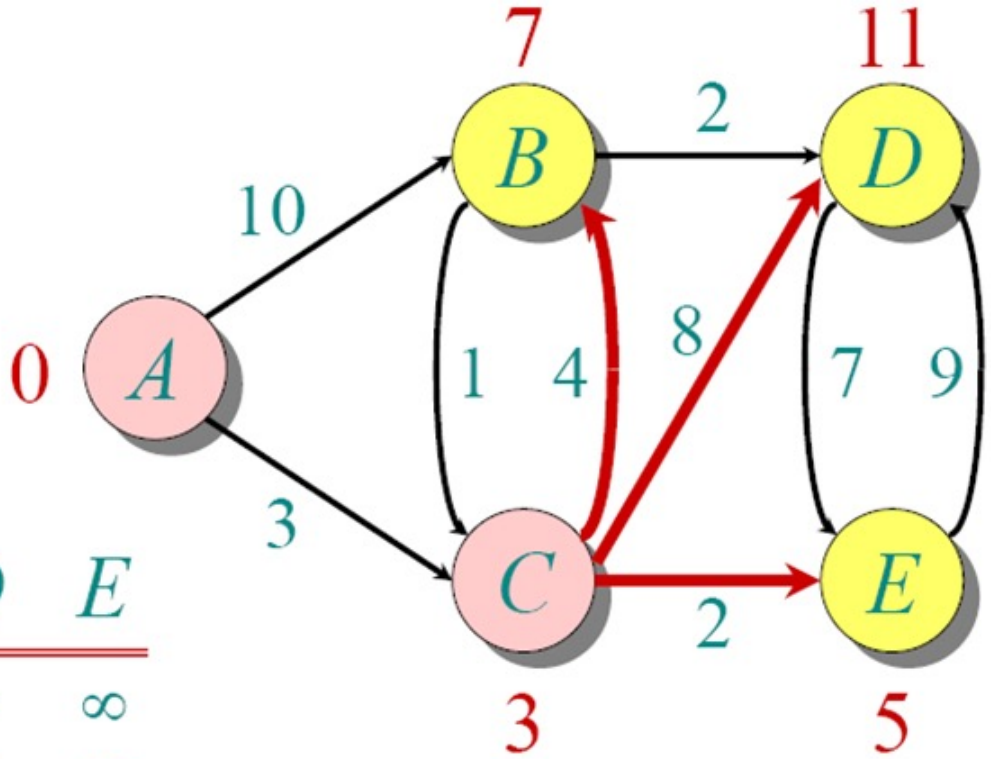
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

$S: \{ A, C \}$

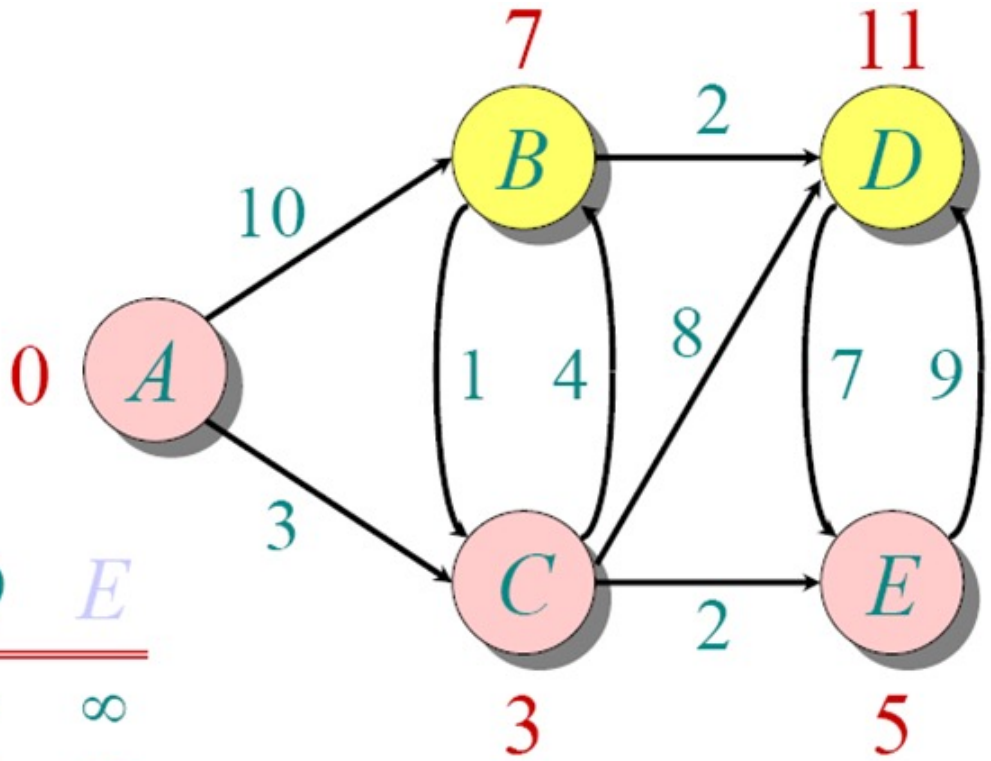
Dijkstra Animated Example



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

$S: \{ A, C \}$

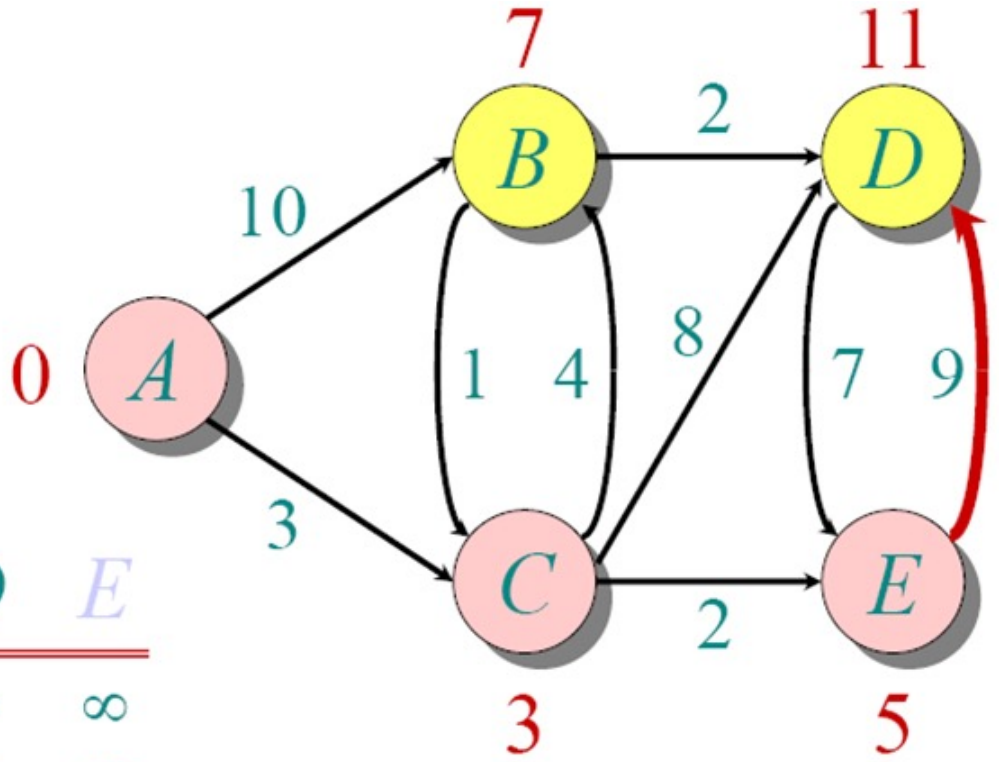
Dijkstra Animated Example



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

$S: \{ A, C, E \}$

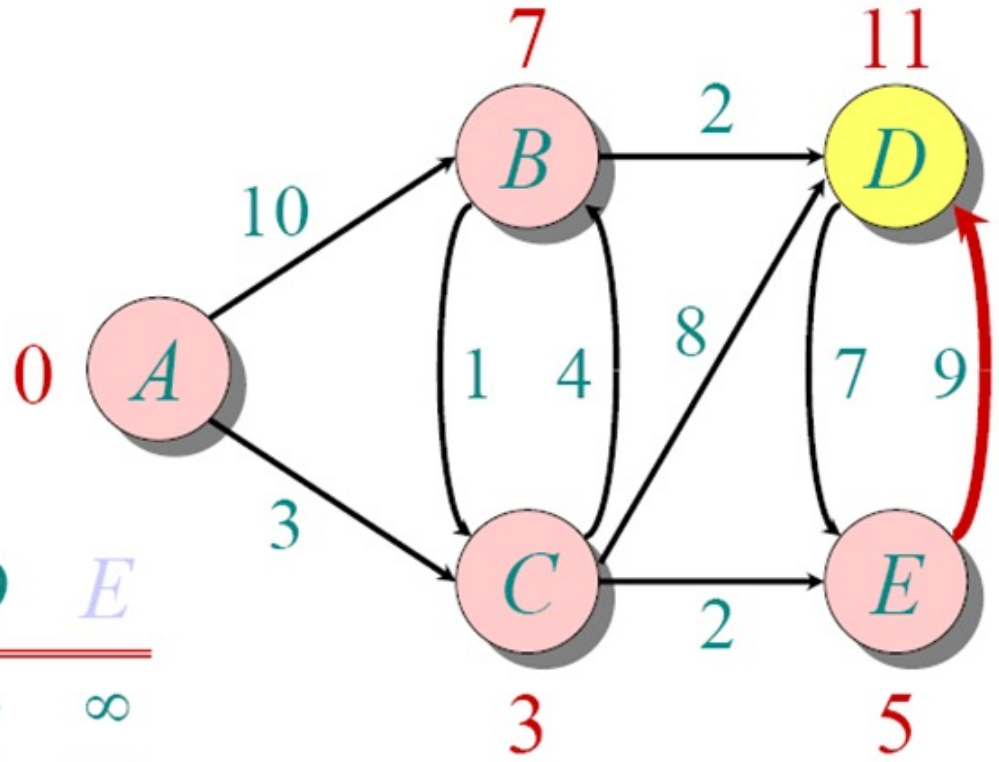
Dijkstra Animated Example



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

$S: \{ A, C, E \}$

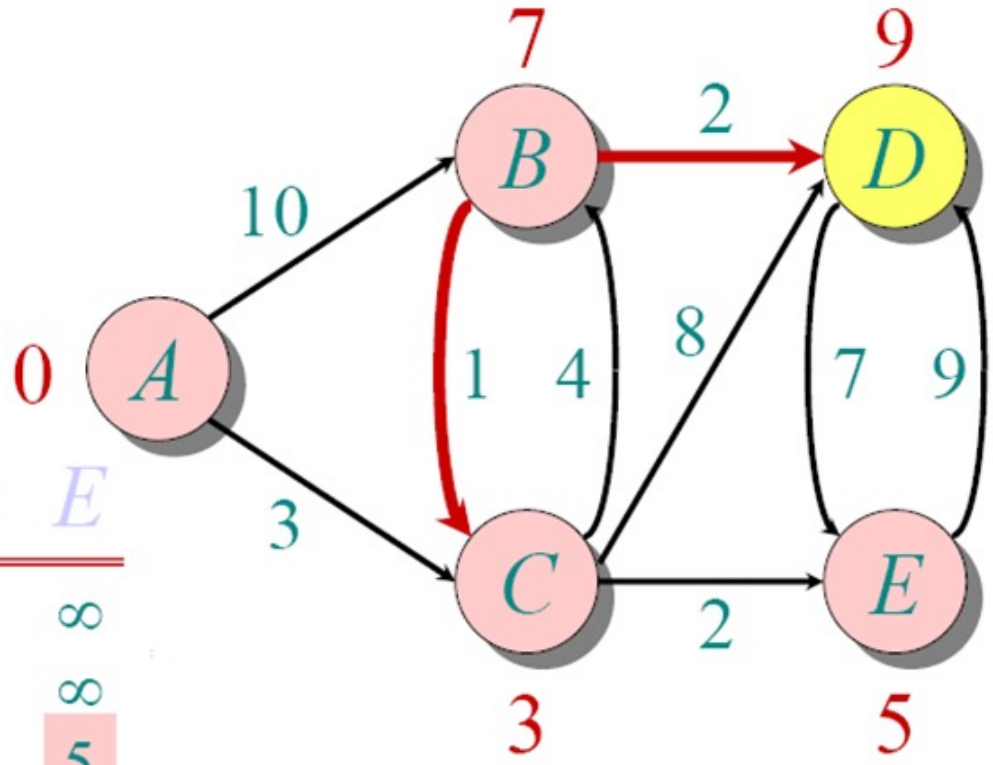
Dijkstra Animated Example



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

S: { A, C, E, B }

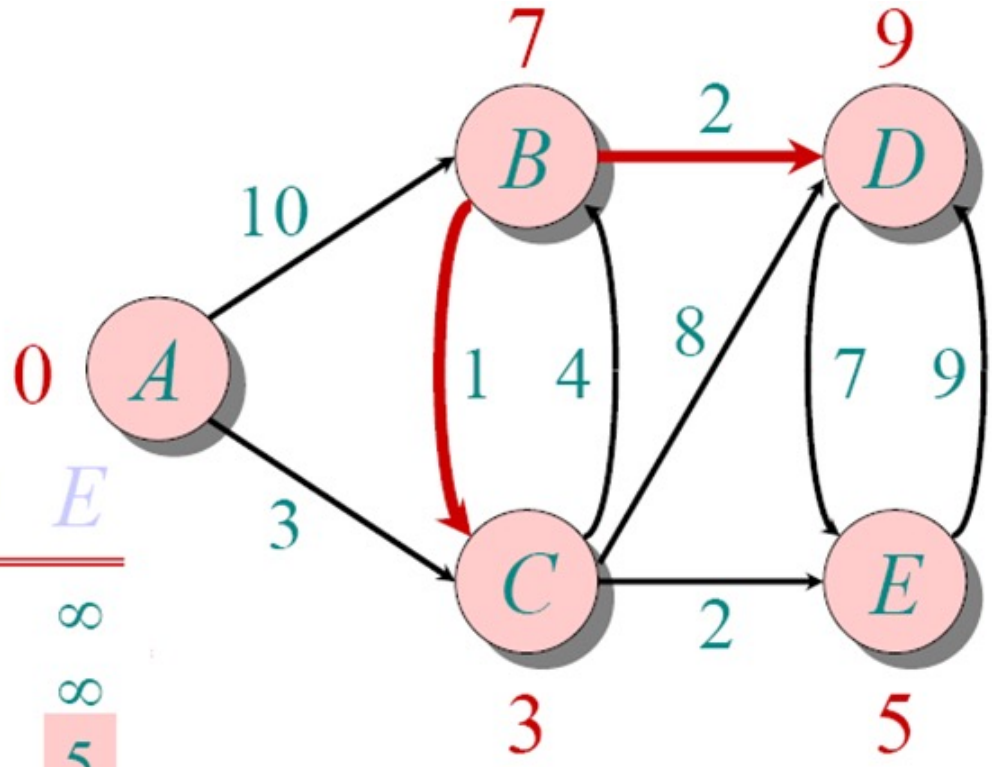
Dijkstra Animated Example



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

$S: \{ A, C, E, B \}$

Dijkstra Animated Example



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

$S: \{ A, C, E, B, D \}$

Pseudocode

```
dist[s] ←o                                  (distance to source vertex is zero)
for  all v ∈ V–{s}
      do  dist[v] ←∞                        (set all other distances to infinity)
S←∅                                         (S, the set of visited vertices is initially empty)
Q←V                                         (Q, the queue initially contains all vertices)
while Q ≠∅                                  (while the queue is not empty)
do   u ← mindistance(Q,dist)                (select the element of Q with the min. distance)
     S←S∪{u}                                (add u to list of visited vertices)
      for all v ∈ neighbors[u]
            do  if   dist[v] > dist[u] + w(u, v)        (if new shortest path found)
                then     d[v] ←d[u] + w(u, v)           (set new value of shortest path)
                                                        (if desired, add traceback code)

return dist
```