CMSC 132: Object-Oriented Programming II

Shortest Paths

Quiz 1

One advantage of adjacency list representation over adjacency matrix representation of a graph is that in adjacency list representation, space is saved for sparse graphs.

A. True
B. False

Quiz 1

One advantage of adjacency list representation over adjacency matrix representation of a graph is that in adjacency list representation, space is saved for sparse graphs.

A. True
B. False

Quiz 2

Traversal of a graph is different from tree because

A. There can be a loop in graph so we must maintain a visited flag for every vertex
B. DFS of a graph uses stack, but inorder traversal of a tree is recursive
C. BFS of a graph uses queue, but a time efficient BFS of a tree is recursive.
D. All of the above

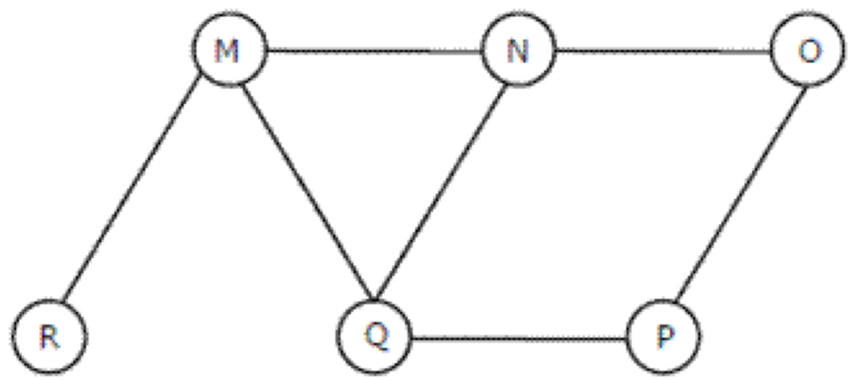
Quiz 2

Traversal of a graph is different from tree because

A. There can be a loop in graph so we must maintain a visited flag for every vertex
B. DFS of a graph uses stack, but inorder traversal of a tree is recursive
C. BFS of a graph uses queue, but a time efficient BFS of a tree is recursive.
D. All of the above

Quiz 3

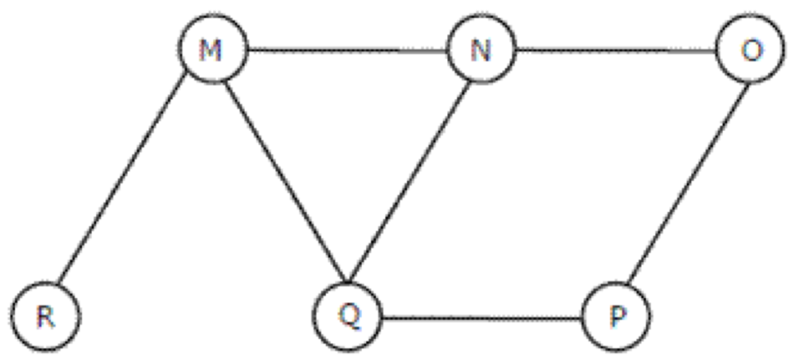
One possible order of Breadth First Search on the following graph



A. MNOPQR
B. NQMPOR
C. QMNPRO
D. QMNPOR

Quiz 3

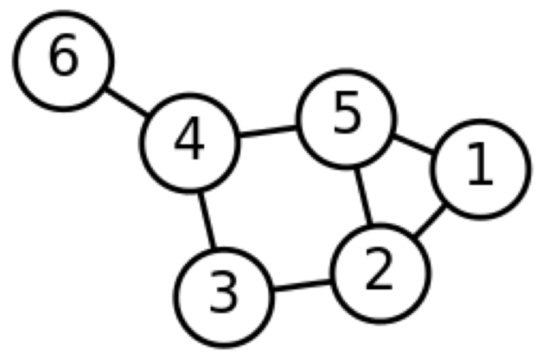
One possible order of Breadth First Search on the following graph



A. MNOPQR
B. NQMPOR
C. QMNPRO
D. QMNPOR

Quiz 4

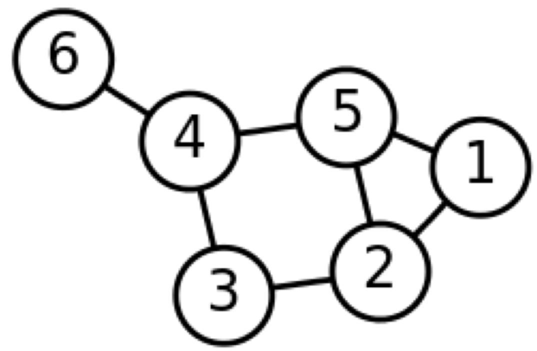
Given two vertices in a graph 1 and 6, which of the two traversals (BFS and DFS) can be used to find if there is path from 1 to 6?



A. Only BFS
B. Only DFS
C. Both BFS and DFS
D. Neither BFS nor DFS

Quiz 4

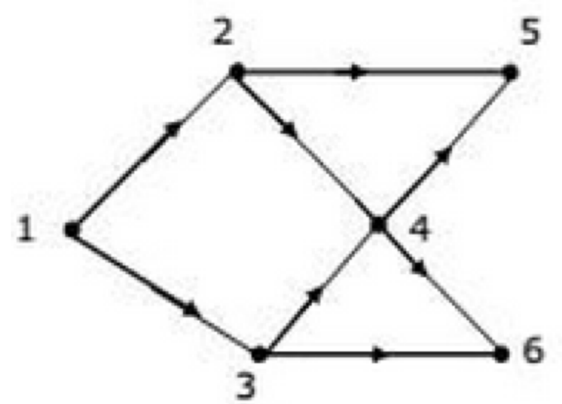
Given two vertices in a graph 1 and 6, which of the two traversals (BFS and DFS) can be used to find if there is path from 1 to 6?



A. Only BFS
B. Only DFS
C. Both BFS and DFS
D. Neither BFS nor DFS

Quiz 5

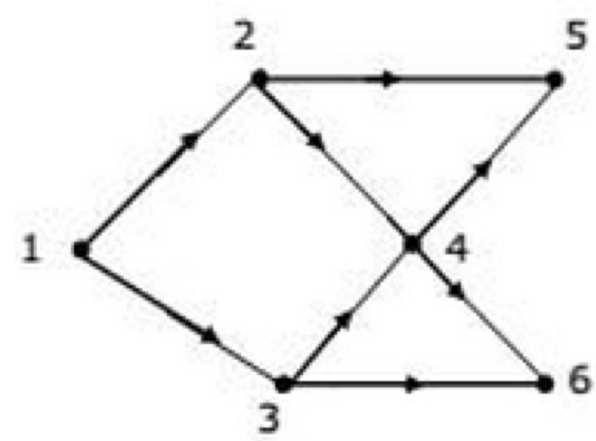
Consider the DAG with Consider V = {1, 2, 3, 4, 5, 6}, shown below. Which of the following is NOT a topological ordering?



A. 1 2 3 4 5 6
B. 1 3 2 4 5 6
C. 1 3 2 4 6 5
D. 3 2 4 1 6 5

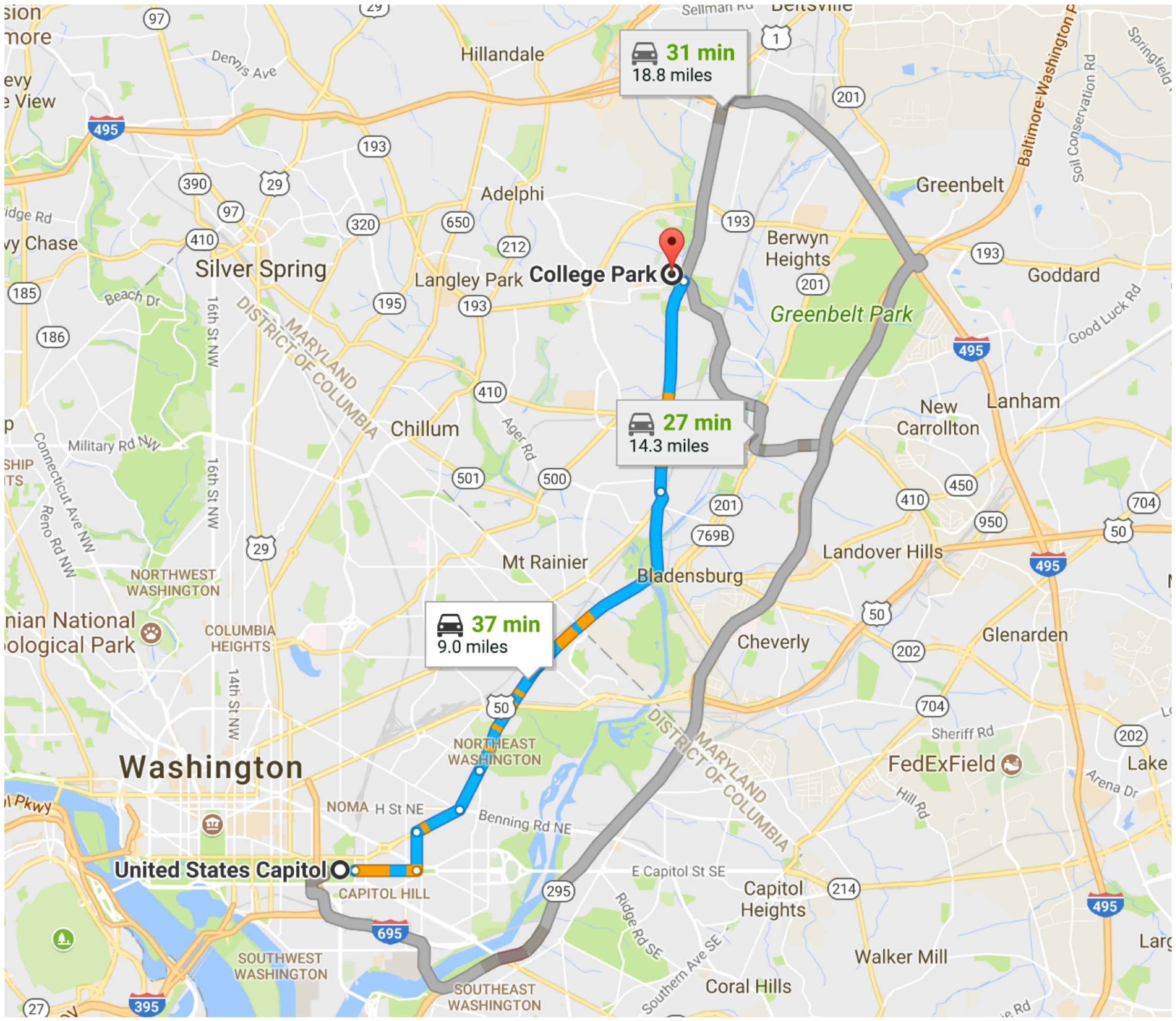
Quiz 5

Consider the DAG with Consider V = {1, 2, 3, 4, 5, 6}, shown below. Which of the following is NOT a topological ordering?



A. 1 2 3 4 5 6
B. 1 3 2 4 5 6
C. 1 3 2 4 6 5
D. 3 2 4 1 6 5

Shortest Paths

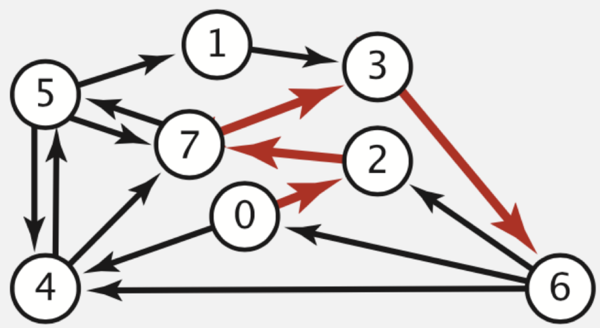


Shortest paths

Given an edge-weighted digraph, find the shortest path from *s* to *t*.

edge-weighted digraph

4->5	0.35
5->4	0.35
4->7	0.37
5->7	0.28
7->5	0.28
5->1	0.32
0->4	0.38
0->2	0.26
7->3	0.39
1->3	0.29
2->7	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93



shortest path from 0 to 6

0->2	0.26
2->7	0.34
7->3	0.39
3->6	0.52

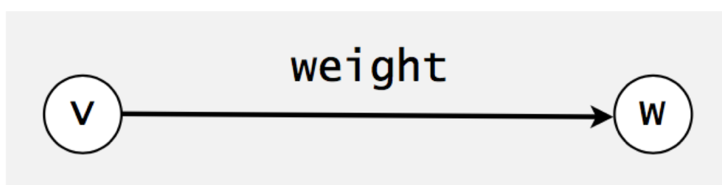
Shortest path variants

- Which vertices?
  - Single source: from one vertex *s* to every other vertex.
  - Source-sink: from one vertex *s* to another *t*.
  - All pairs: between all pairs of vertices.
- Restrictions on edge weights?
  - Nonnegative weights.
  - Euclidean weights.
  - Arbitrary weights.
- Cycles?
  - No directed cycles.
  - No "negative cycles."
- Simplifying assumption: Shortest paths from *s* to each vertex *v* exist.

Weighted directed edge

public class DirectedEdge

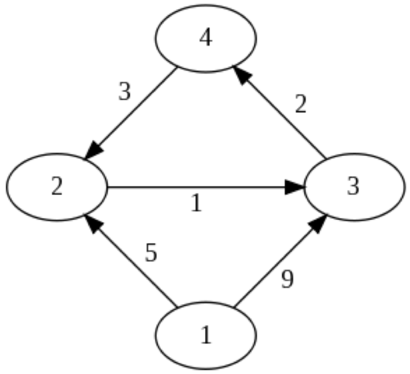
	DirectedEdge(int v, int w, double weight)	*weighted edge v→w*
int	from()	*vertex v*
int	to()	*vertex w*
double	weight()	*weight of this edge*
String	toString()	*string representation*



Idiom for processing an edge e: **int v = e.from(), w = e.to();**

Weighted directed edge implementation

```
public class DirectedEdge{
   private final int v, w;
   private final double weight;

   public DirectedEdge(int v, int w, double weight){
        this.v = v;
        this.w = w;
        this.weight = weight;
   }

   public int from() { return v; }
   public int to() { return w; }
   public double weight() { return weight; }
}
```



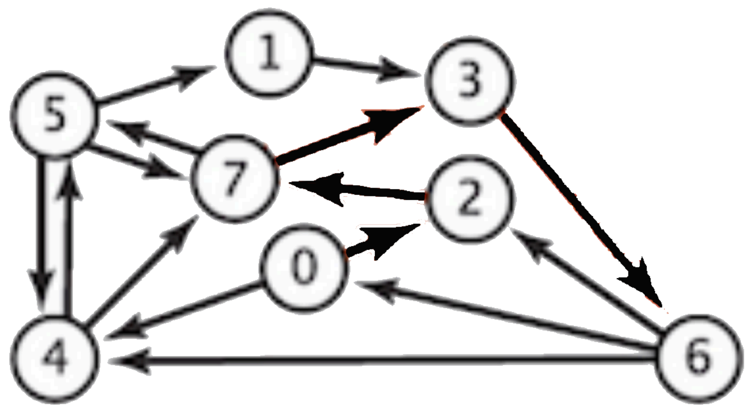
Edge-weighted digraph

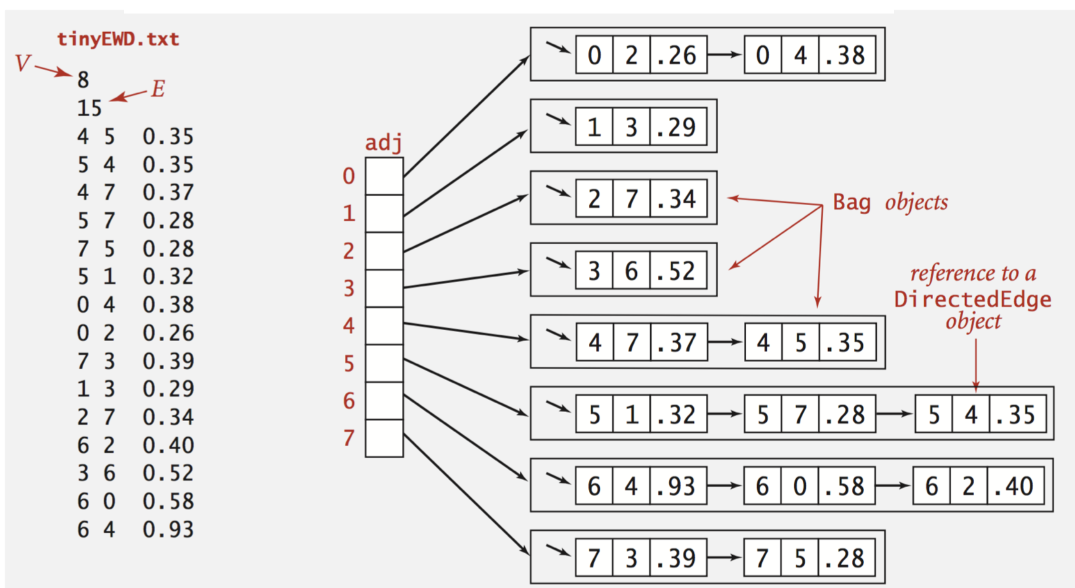
```
public class EdgeWeightedDigraph
```

	`EdgeWeightedDigraph(int V)`	*edge-weighted digraph with V vertices*
`void`	`addEdge(DirectedEdge e)`	*add weighted directed edge e*
`Iterable<DirectedEdge>`	`adj(int v)`	*edges pointing from v*
`int`	`V()`	*number of vertices*
`int`	`E()`	*number of edges*
`Iterable<DirectedEdge>`	`edges()`	*all edges*
`String`	`toString()`	*string representation*

Conventions. Allow self-loops and parallel edges.

Edge-weighted digraph: adjacency-lists representation



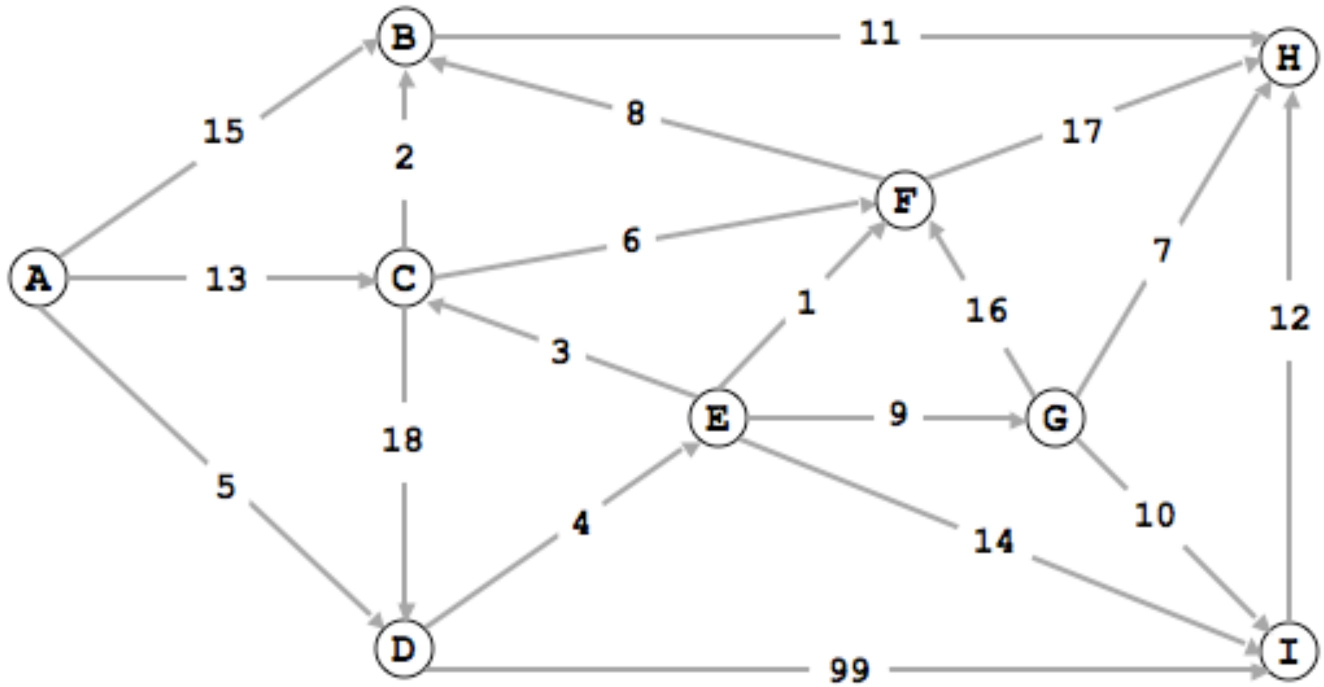


Edge-weighted digraph implementation

```
public class EdgeWeightedDigraph{
  private final int V;
  private final Bag<DirectedEdge>[] adj;

  public EdgeWeightedDigraph(int V){
     this.V = V;
     adj = (Bag<DirectedEdge>[]) new Bag[V];
     for (int v = 0; v < V; v++)
         adj[v] = new Bag<DirectedEdge>();
  }

  public void addEdge(DirectedEdge e){
      int v = e.from();
      adj[v].add(e);
  }

  public Iterable<DirectedEdge> adj(int v){
       return adj[v];
  }
}
```

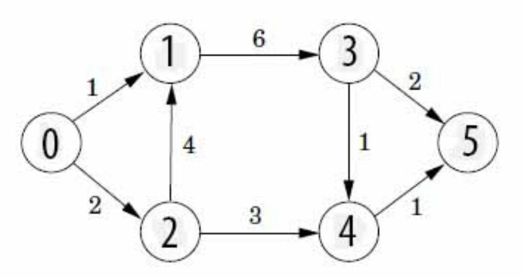
Single-source shortest paths

What is the shortest distance and path from A to H?



Single-source shortest paths

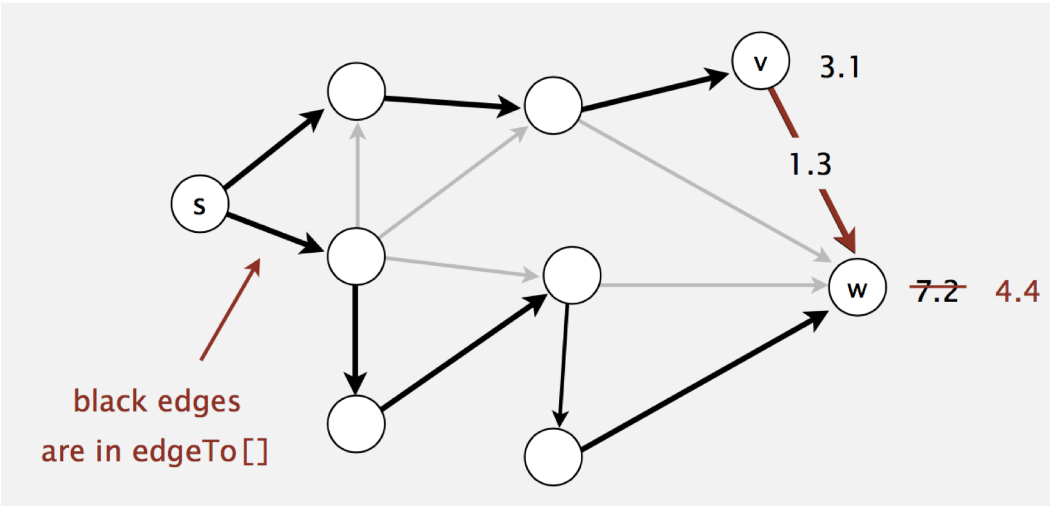
- Data structures: Represent the Shortest Path with two vertex-indexed arrays:
  - distTo[v] is length of shortest path from *s* to *v*.
  - edgeTo[v] is last edge on shortest path from *s* to *v*.




```
public double distTo(int v){
  return distTo[v];
}

public Iterable<DirectedEdge> pathTo(int v){
    Stack<DirectedEdge> path = new Stack<DirectedEdge>();
    DirectedEdge e = edgeTo[v];

    while (e != null){
      path.push(e);
      e = edgeTo[e.from()];
   }
   return path;
}
```

Edge relaxation

- Relax edge e = v→w.
  - distTo[v] is length of shortest known path from **s to v**.
  - distTo[w] is length of shortest known path from **s to w**.
  - edgeTo[w] is last edge on shortest known path from s to w.
  - If **e = v→w** gives shorter path to w through v, update both distTo[w] and edgeTo[w]

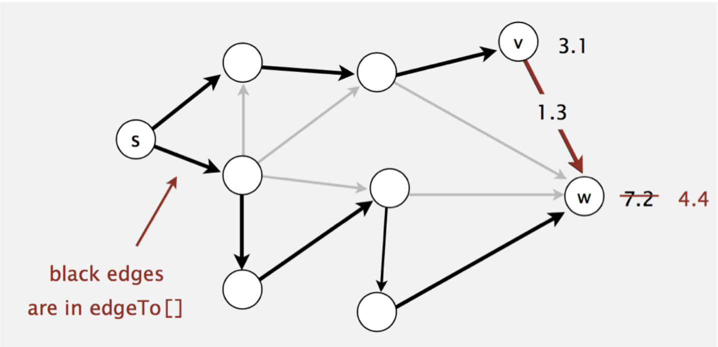


v→w successfully relaxes

Edge relaxation

- Relax edge e = v→w.
  - distTo[v] is length of shortest known path from s to v.
  - distTo[w] is length of shortest known path from s to w.
  - edgeTo[w] is last edge on shortest known path from s to w.
  - If e = v→w gives shorter path to w through v, update both distTo[w] and edgeTo[w]

```
private void relax(DirectedEdge e) {
   int v = e.from(), w = e.to();
   if (distTo[w] > distTo[v] + e.weight()) {
      distTo[w] = distTo[v] + e.weight();
      edgeTo[w] = e;
  }
}
```



Generic shortest-paths algorithm

Generic algorithm (to compute SPT from s)

Initialize distTo[s] = 0 and distTo[v] = ∞ for all other vertices.

Repeat until optimality conditions are satisfied:

Relax any edge.

Efficient implementations: How to choose which edge to relax?

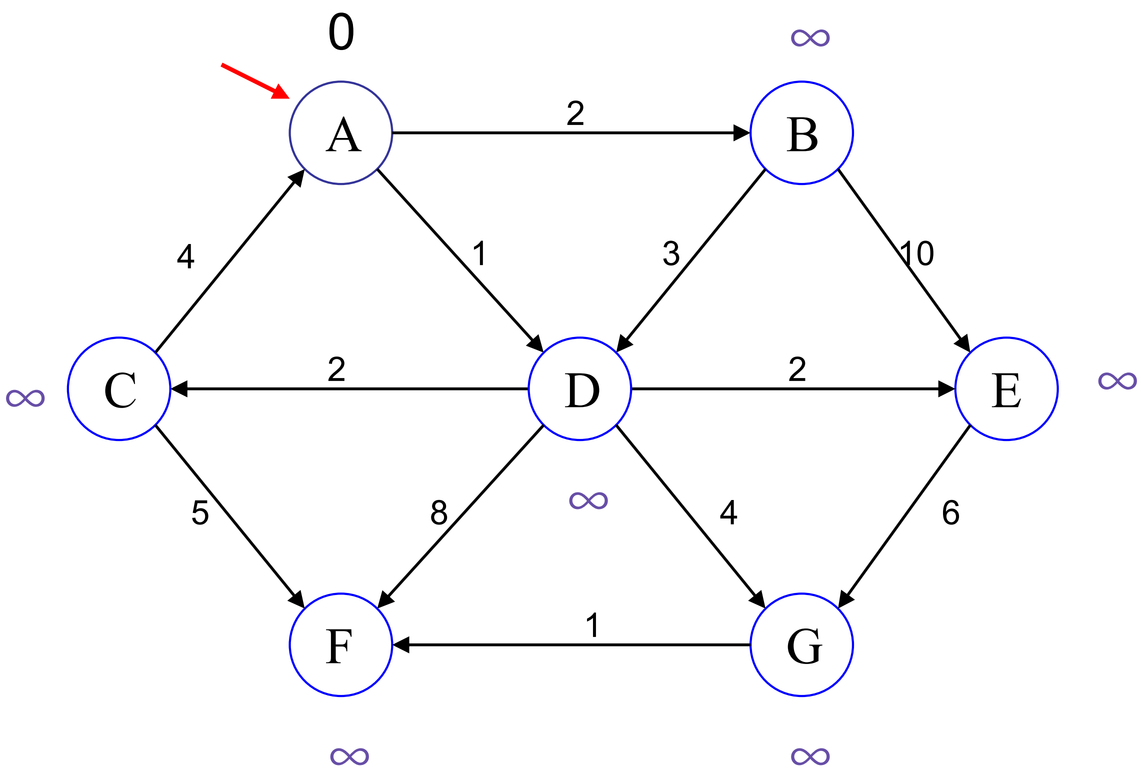
- Dijkstra's algorithm (nonnegative weights).
- Topological sort algorithm (no directed cycles).
- Bellman-Ford algorithm (no negative cycles).

Dijkstra's algorithm

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest distTo[] value).
- Add vertex to tree and relax all edges pointing from that vertex.

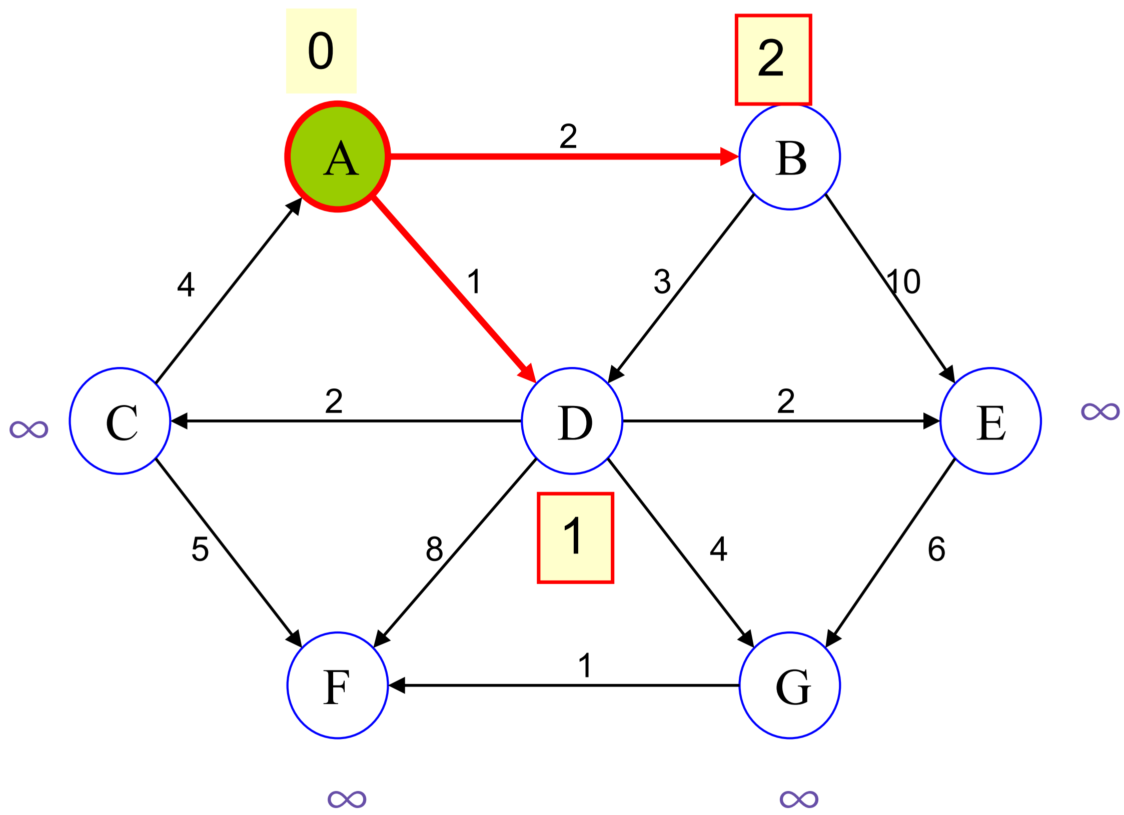
Dijkstra's algorithm Demo

Pick vertex in List with minimum distance.



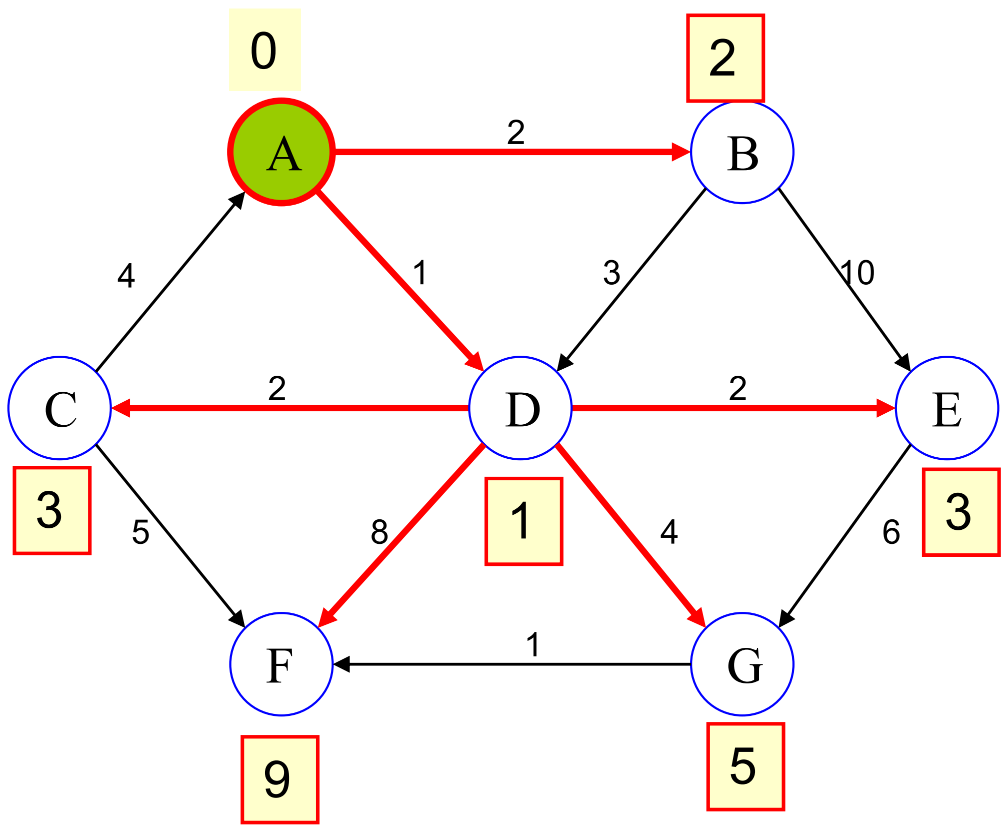
V	distTo[]	edgeTo
A	0	--
B	∞	
C	∞	
D	∞	
E	∞	
F	∞	

Update A’s neighbors



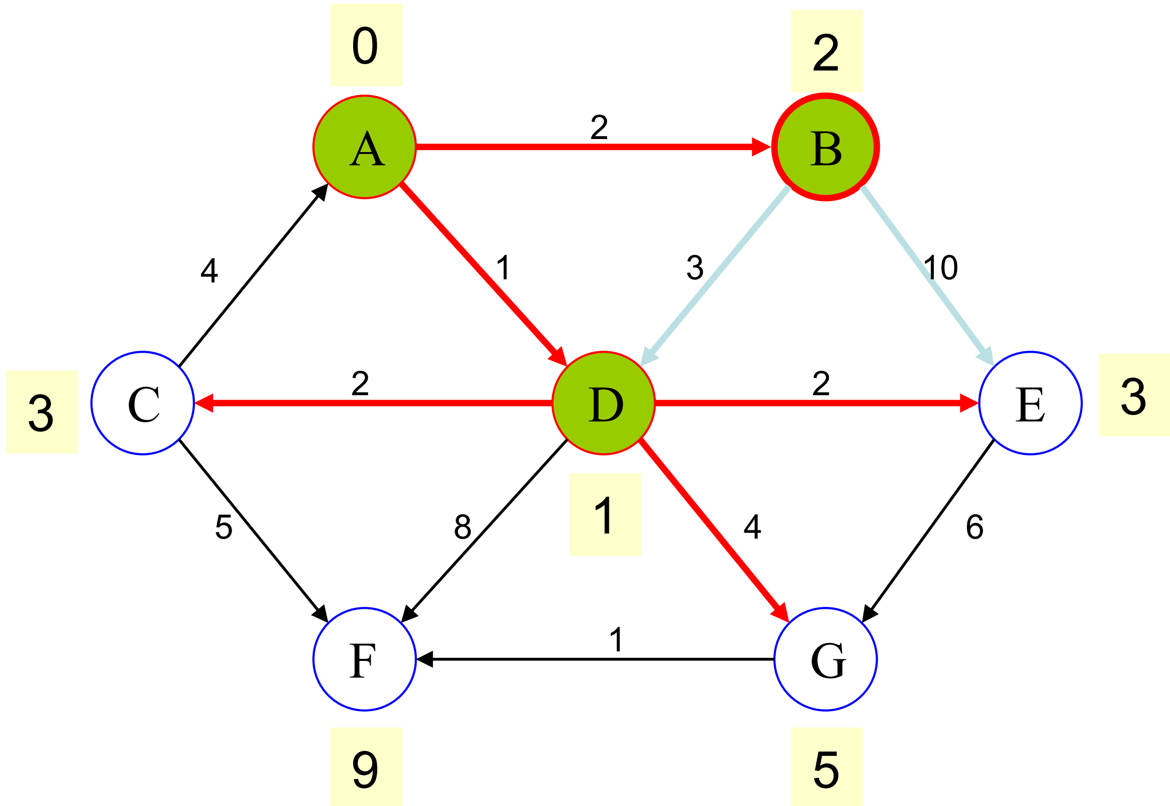
V	distTo[]	edgeTo
A	0	--
B	**2**	0
C	∞	
D	**1**	A
E	∞	
F	∞	

Update D’s neighbors



V	distTo[]	edgeTo
A	0	--
B	**2**	A
C	**3**	D
D	**1**	A
E	**3**	D
F	**9**	D
G	**5**	D

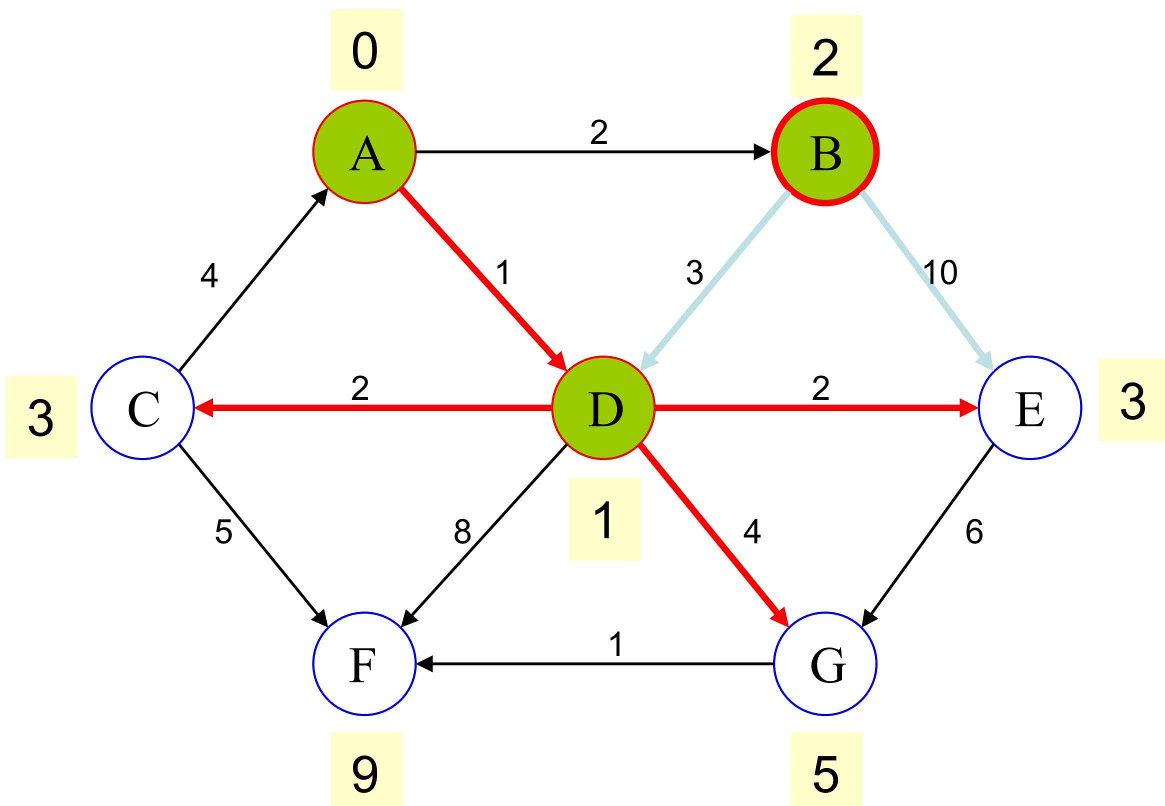
Update B's neighbors



V	distTo[]	edgeTo
A	0	--
B	**2**	A
C	**3**	D
D	**1**	A
E	**3**	D
F	**9**	D
G	**5**	D

No Update

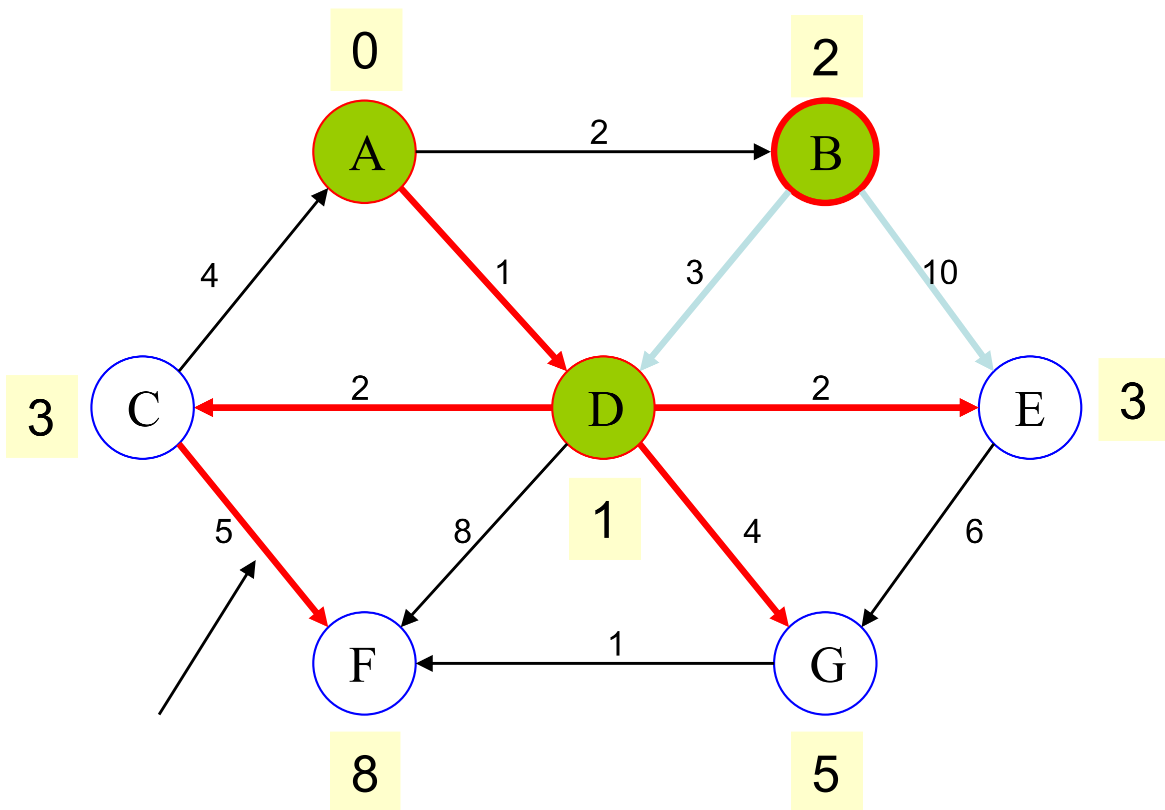
Update E’s neighbors



V	distTo[]	edgeTo
A	0	--
B	**2**	A
C	**3**	D
D	**1**	A
E	**3**	D
F	**9**	D
G	**5**	D

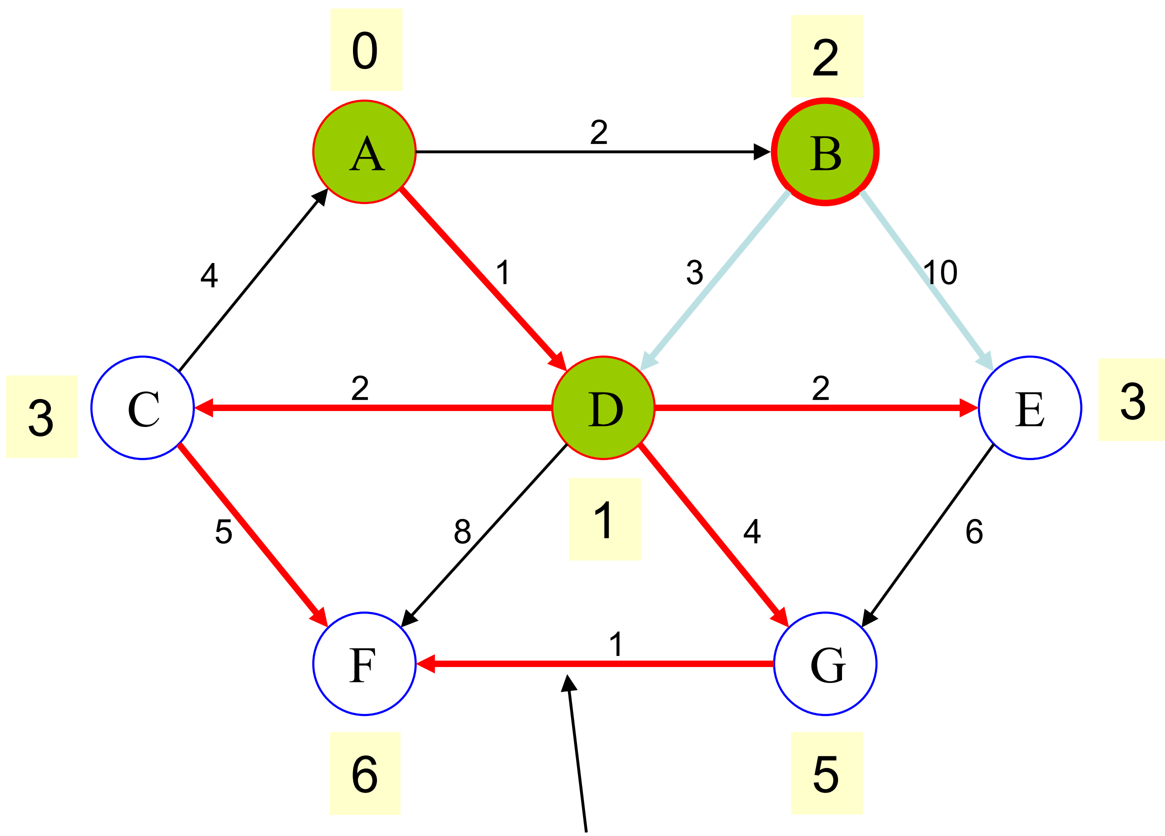
No Update

Update C's neighbors



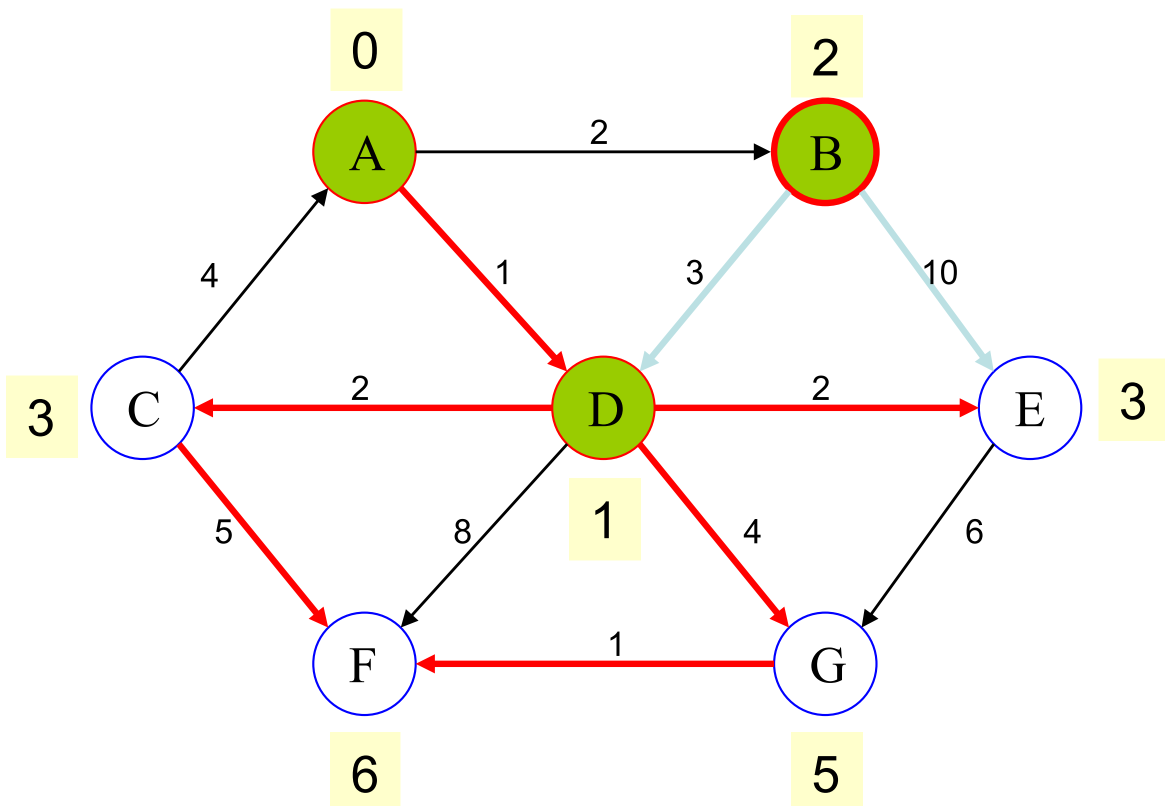
V	distTo[]	edgeTo
A	0	--
B	**2**	A
C	**3**	D
D	**1**	A
E	**3**	D
F	**8**	C
G	**5**	D

Update G's neighbors



V	distTo[]	edgeTo
A	0	--
B	**2**	A
C	**3**	D
D	**1**	A
E	**3**	D
F	**6**	G
G	**5**	D

Update F's neighbors

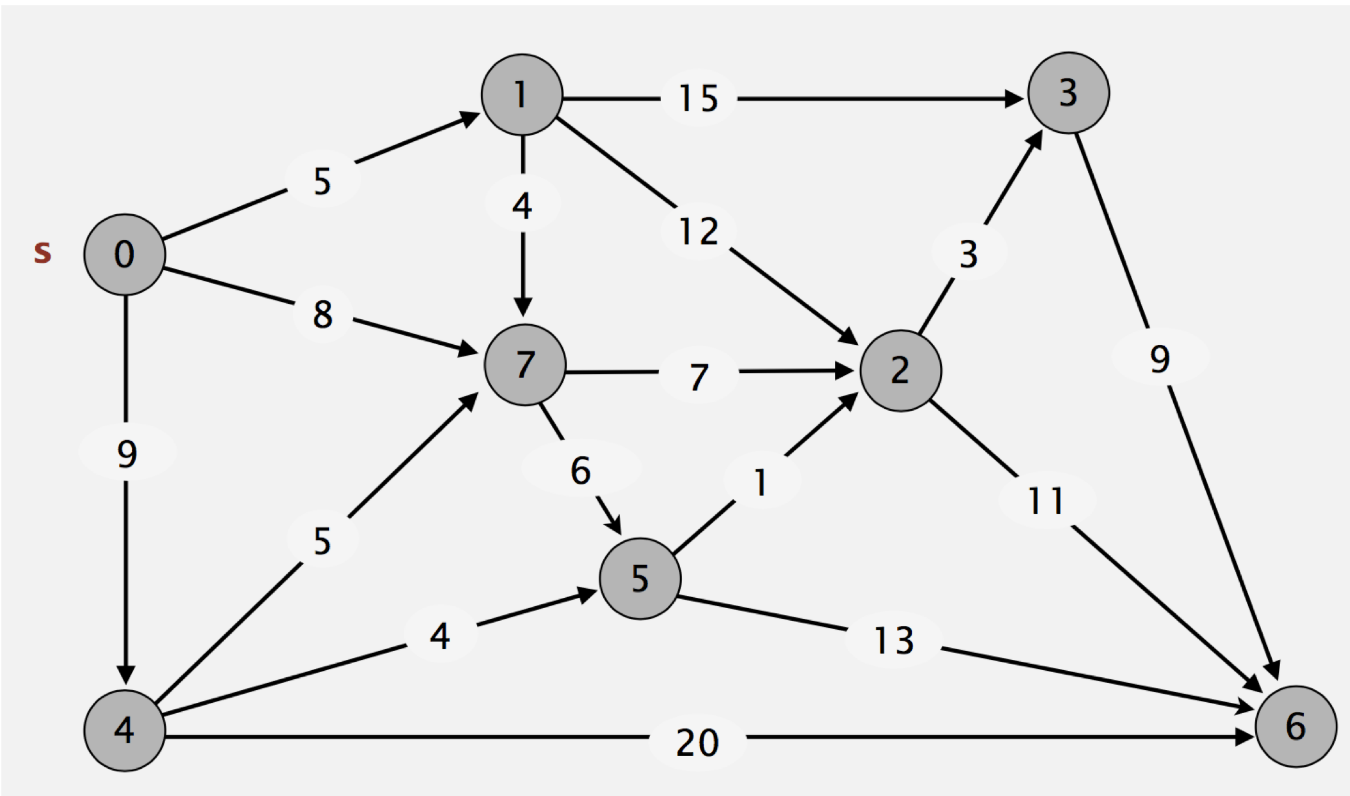


V	distTo[]	edgeTo
A	0	--
B	**2**	A
C	**3**	D
D	**1**	A
E	**3**	D
F	**6**	G
G	**5**	D

No Update

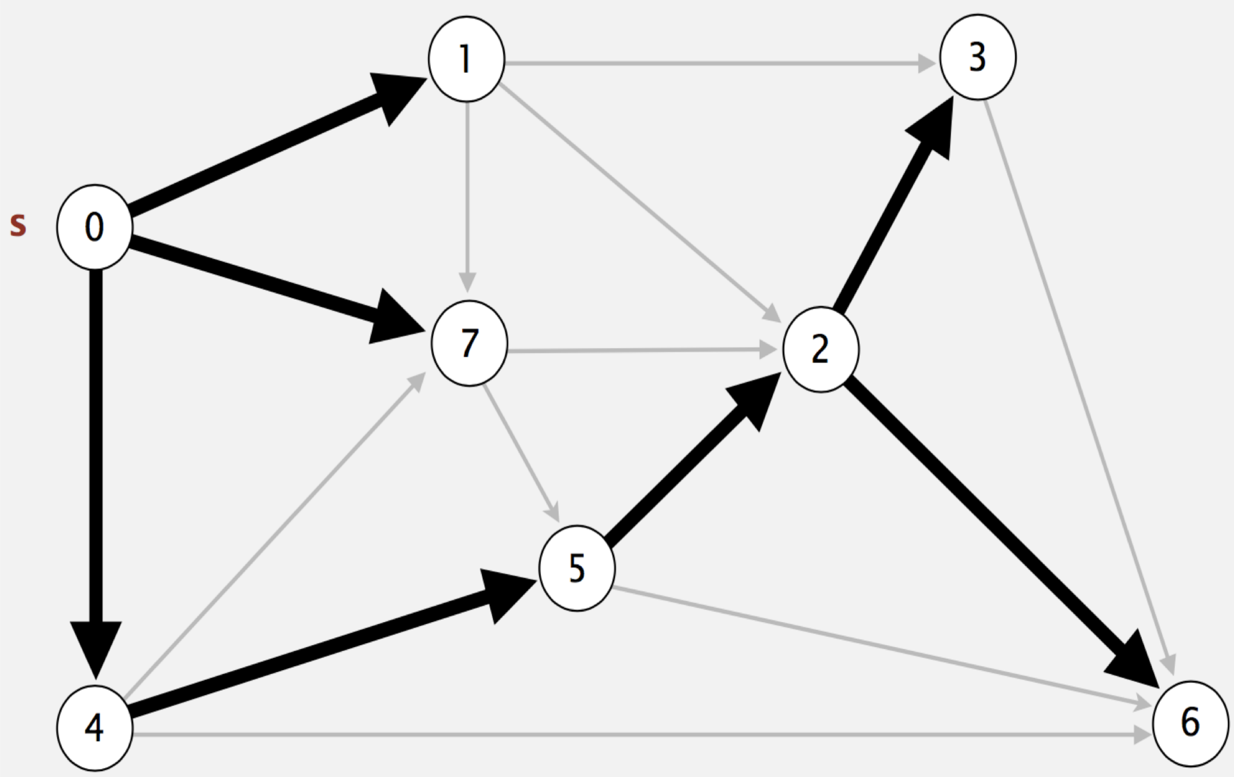
Dijkstra's algorithm Demo

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest distTo[] value).
- Add vertex to tree and relax all edges pointing from that vertex.



Dijkstra's algorithm

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest distTo[] value).
- Add vertex to tree and relax all edges pointing from that vertex.

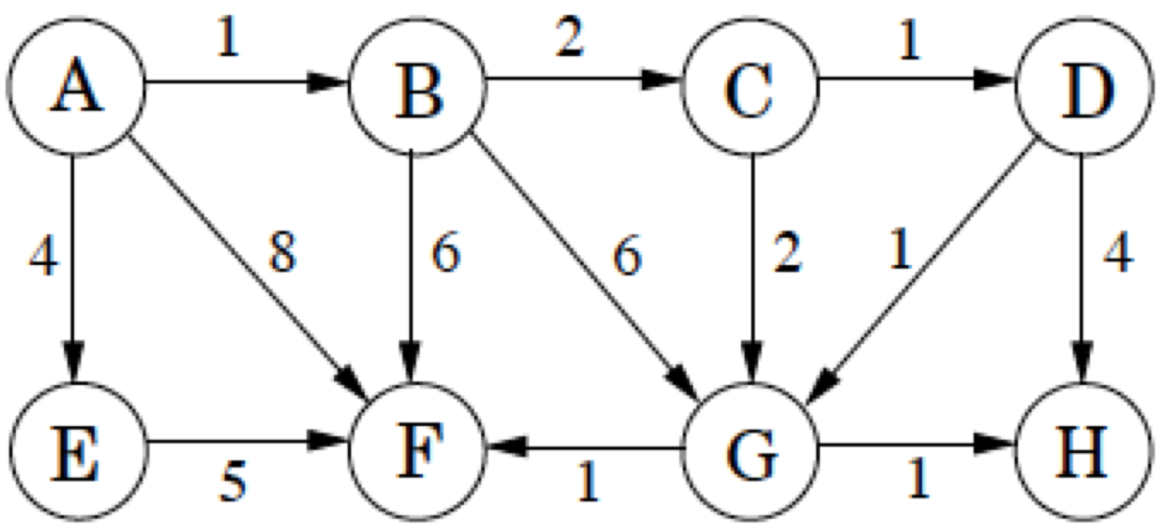


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

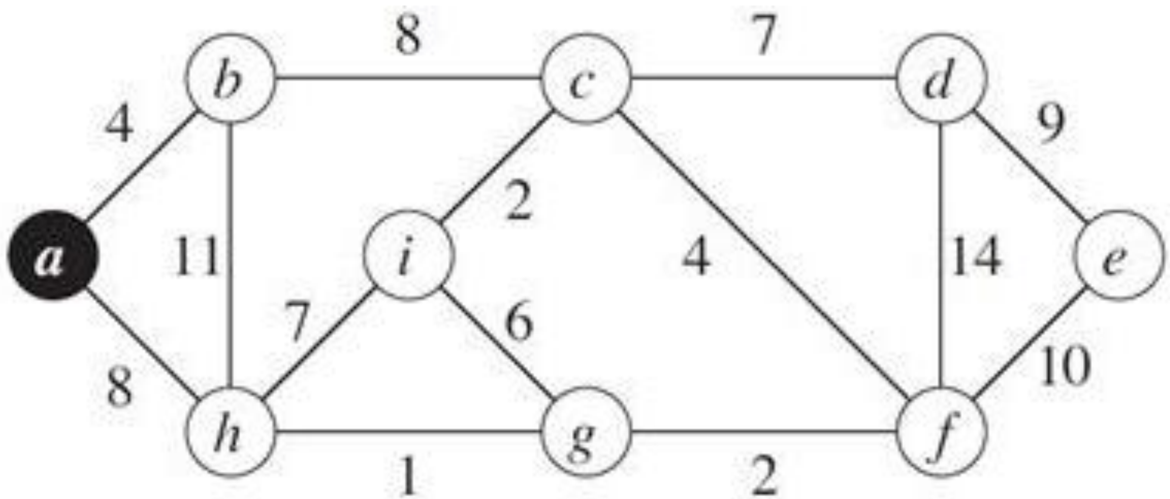
Dijkstra's algorithm Implementation

```
public class DijkstraSP{
   private DirectedEdge[] edgeTo;
   private double[] distTo;
   private IndexMinPQ<Double> pq;

  public DijkstraSP(EdgeWeightedDigraph G, int s) {
      edgeTo = new DirectedEdge[G.V()];
      distTo = new double[G.V()];
      pq = new IndexMinPQ<Double>(G.V());
      for (int v = 0; v < G.V(); v++)
         distTo[v] = Double.POSITIVE_INFINITY;
      distTo[s] = 0.0;
      pq.insert(s, 0.0);
      while (!pq.isEmpty()){
         int v = pq.delMin();
         for (DirectedEdge e : G.adj(v))
             relax(e);
      }
  }
}
```

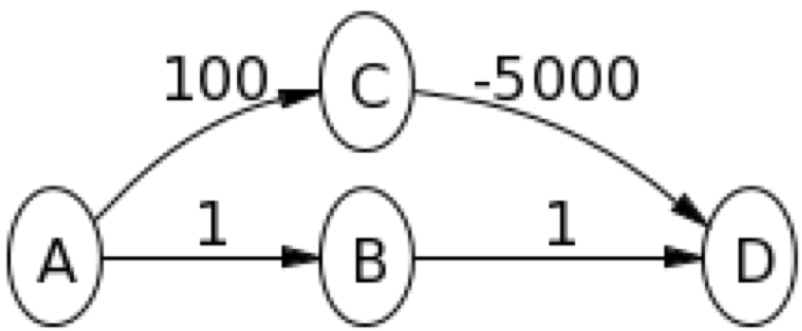
Shortest Path Demo



Shortest Path Demo



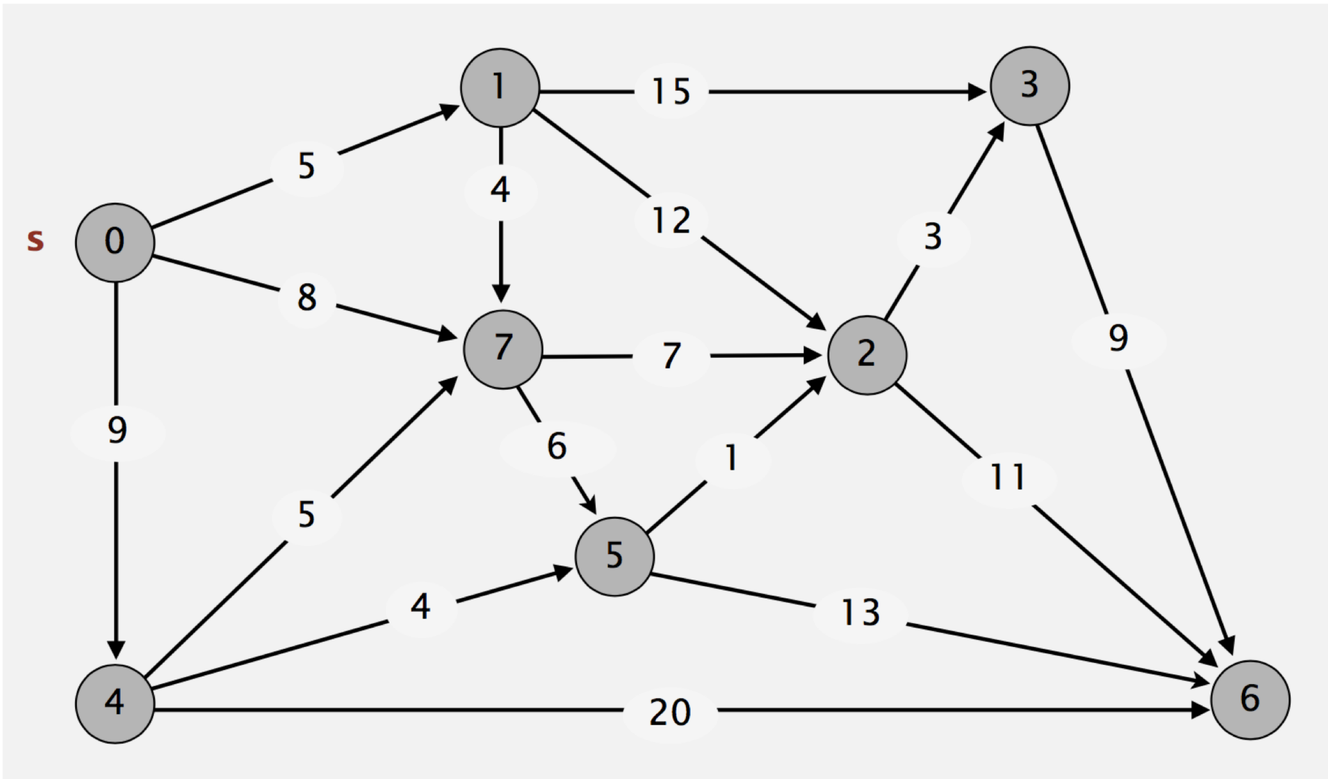
Shortest Path Demo



Acyclic shortest paths

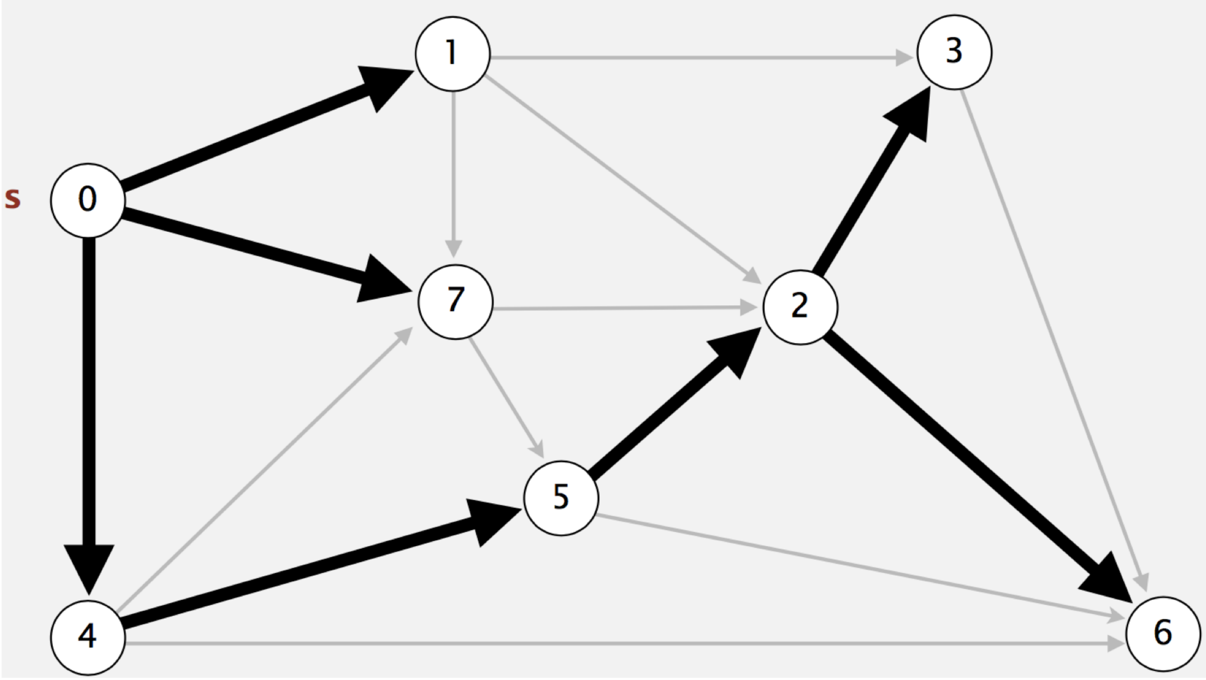
- Consider vertices in topological order. Relax all edges pointing from that vertex.

0 1 4 7 5 2 3 6



Acyclic shortest paths

- Consider vertices in topological order.
- Relax all edges pointing from that vertex.



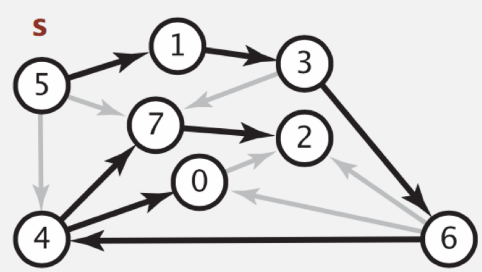
0 1 4 7 5 2 3 6

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Longest paths in edge-weighted DAGs

- Formulate as a shortest paths problem in edge-weighted DAGs.
  - Negate all weights.
  - Find shortest paths.
  - Negate weights in result
- Key point. Topological sort algorithm works even with negative weights.

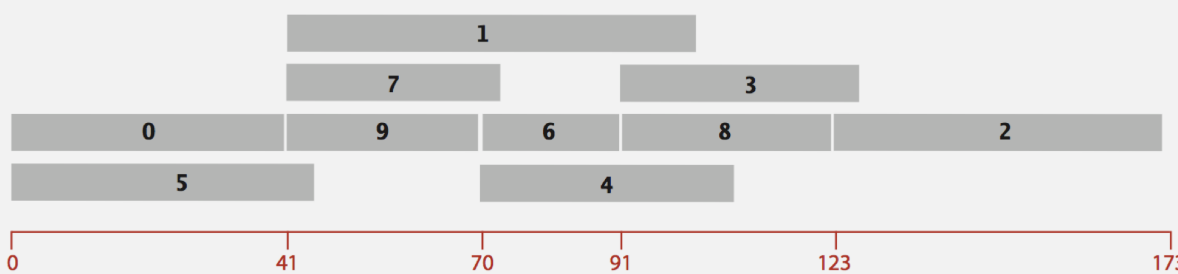
longest paths input		**shortest paths input**	
5->4	0.35	5->4	-0.35
4->7	0.37	4->7	-0.37
5->7	0.28	5->7	-0.28
5->1	0.32	5->1	-0.32
4->0	0.38	4->0	-0.38
0->2	0.26	0->2	-0.26
3->7	0.39	3->7	-0.39
1->3	0.29	1->3	-0.29
7->2	0.34	7->2	-0.34
6->2	0.40	6->2	-0.40
3->6	0.52	3->6	-0.52
6->0	0.58	6->0	-0.58
6->4	0.93	6->4	-0.93



Longest paths in edge-weighted DAGs

- Parallel job scheduling.
  - Given a set of jobs with durations and precedence constraints, schedule the jobs (by finding a start time for each) so as to achieve the minimum completion time, while respecting the constraints.

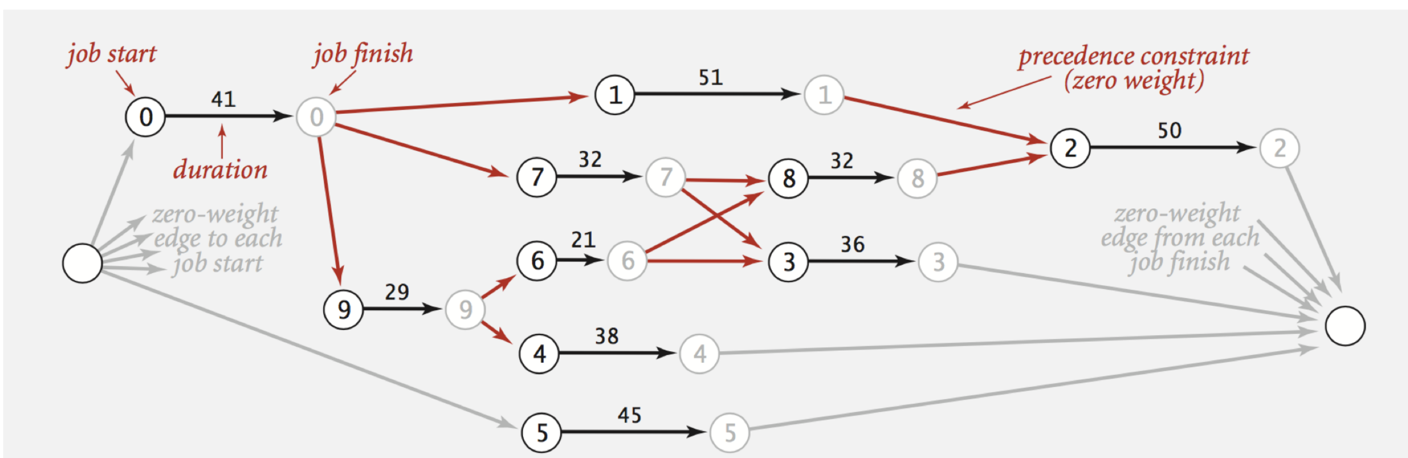
job	*duration*	*must complete before*
0	41.0	1 7 9
1	51.0	2
2	50.0	
3	36.0	
4	38.0	
5	45.0	
6	21.0	3 8
7	32.0	3 8
8	32.0	2
9	29.0	4 6



Parallel job scheduling solution

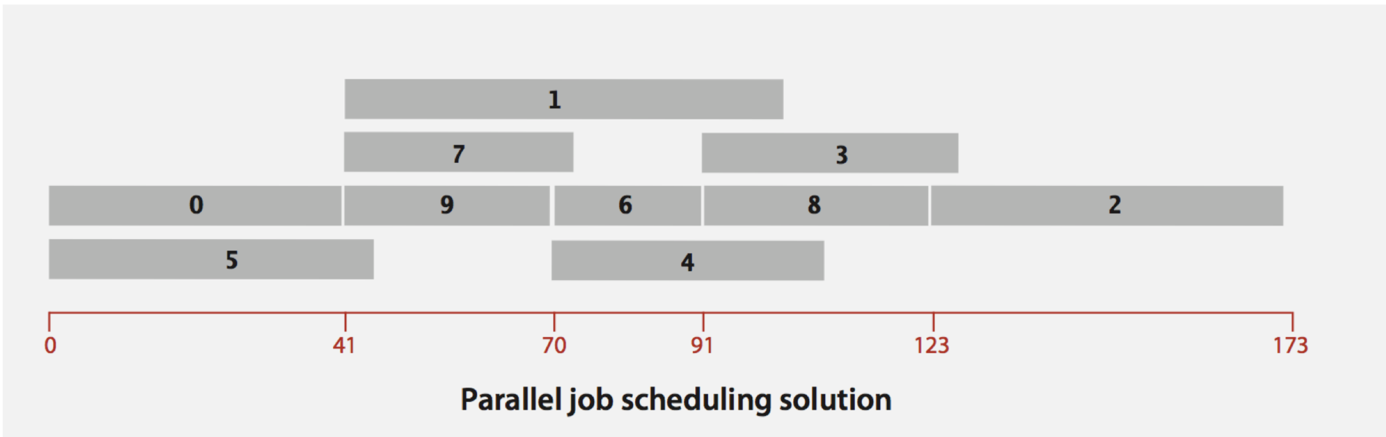
Critical path method

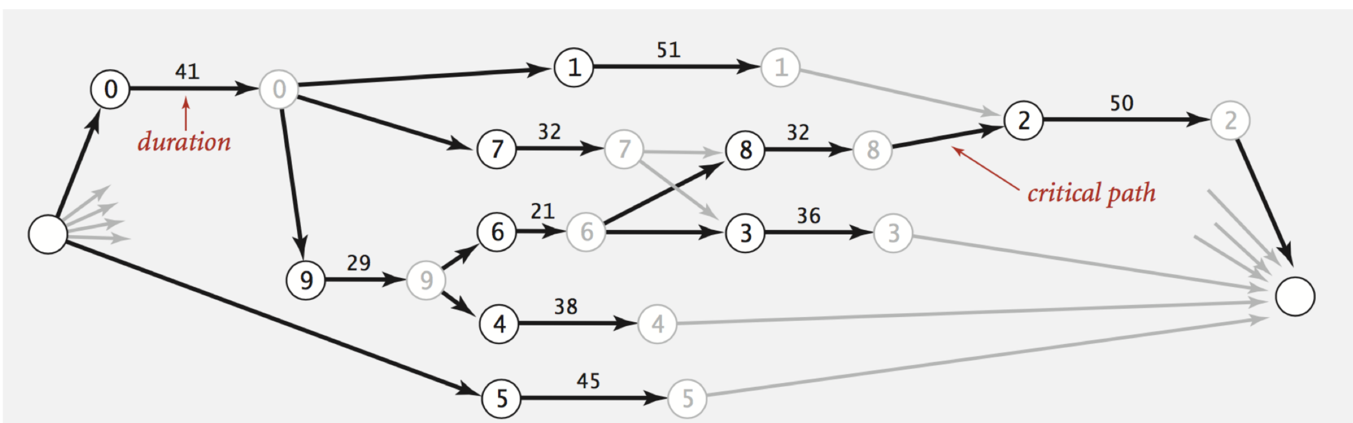
- To solve a parallel job-scheduling problem, create edge-weighted DAG:
  - Source and sink vertices.
  - Two vertices (begin and end) for each job.
  - Three edges for each job.
    - Begin to end (weighted by duration)
    - Source to begin(0 weight)
    - End to sink(0 weight)
- One edge for each precedence constraint (0 weight).



Critical path method

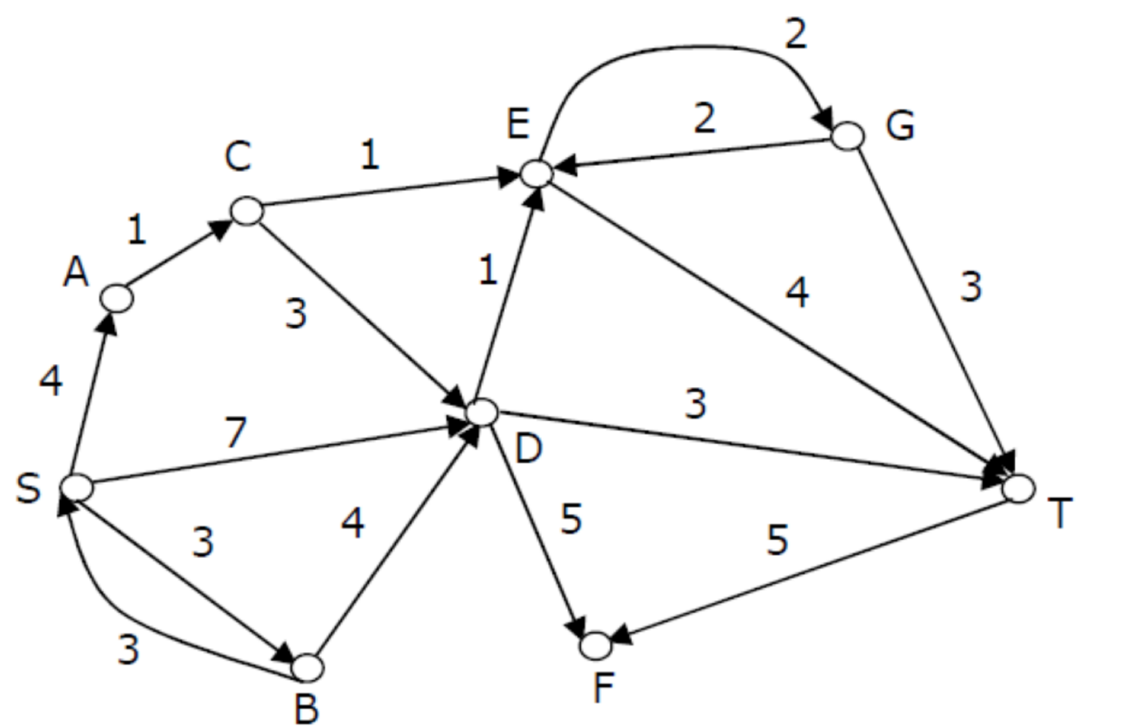
Use longest path from the source to schedule each job.





Quiz 1

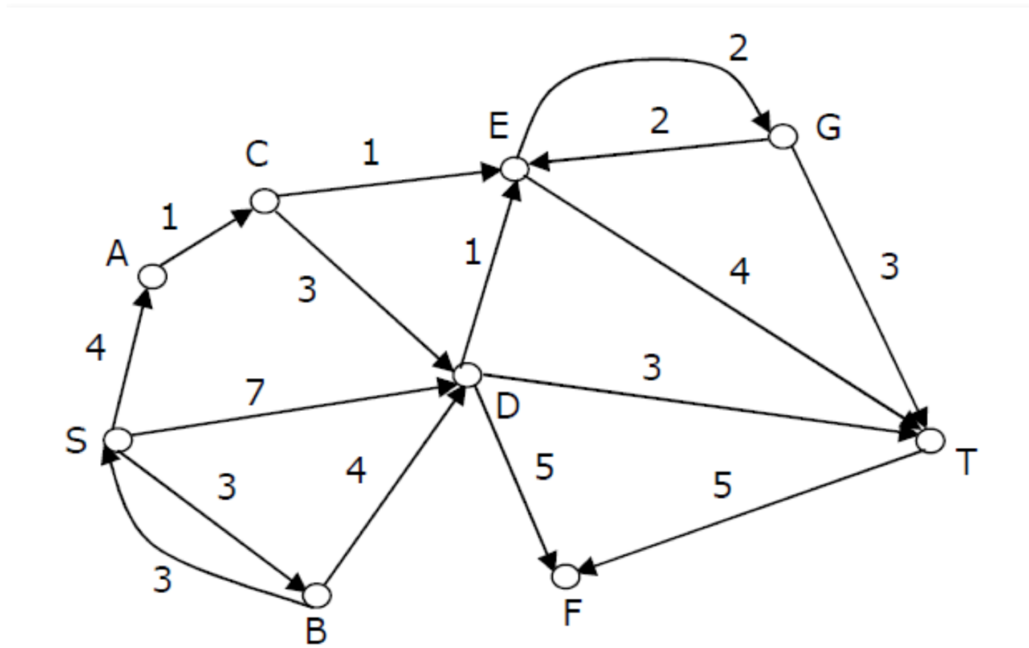
There are multiple shortest paths between vertices S and T. Which one will be reported by Dijstra's shortest path algorithm?



A. SDT
B. SBDT
C. SACDT
D. SACET

Quiz 1

There are multiple shortest paths between vertices S and T. Which one will be reported by Dijstra's shortest path algorithm?



A. SDT
B. SBDT
C. SACDT
D. SACET

Quiz 2

In an unweighted, undirected connected graph, the shortest path from a node S to every other node is computed most efficiently, in terms of time complexity by

A. Dijkstra’s algorithm starting from S.
B. Performing a DFS starting from S.
C. Performing a BFS starting from S.
D. None of the above

Quiz 2

In an unweighted, undirected connected graph, the shortest path from a node S to every other node is computed most efficiently, in terms of time complexity by

A. Dijkstra’s algorithm starting from S.
B. Performing a DFS starting from S.
C. Performing a BFS starting from S.
D. None of the above