CSE221

Lecture 20:
Directed Graphs and Graph Algorithms

Hyungon Moon

Acknowledgment: The content of this file is based on the slides of the textbook as well as the slides provided by Prof. Won-Ki Jeong.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Outline

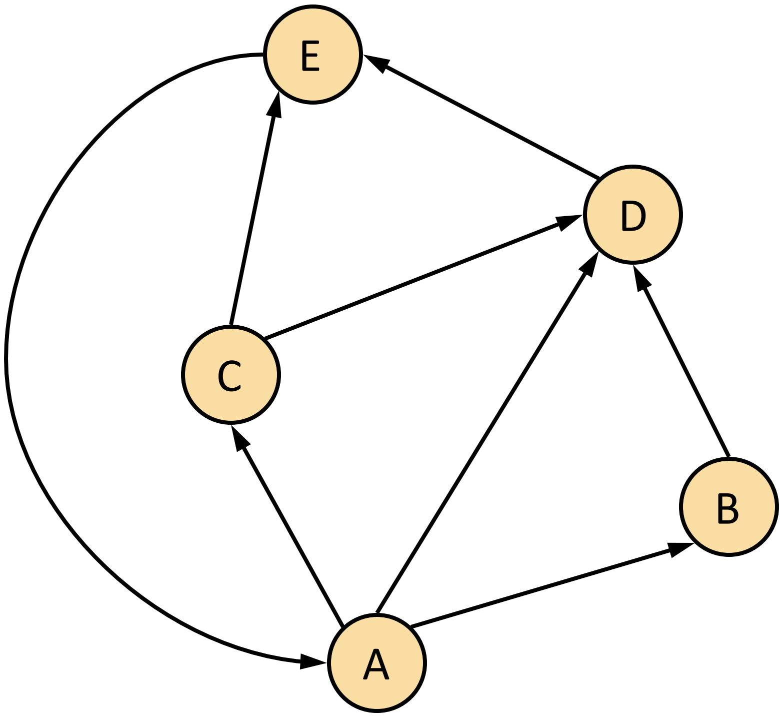
- Directed graphs
- Shortest path algorithms

Outline

- Directed graphs
  - Digraph properties
  - Reachability
  - Topological sorting
- Shortest path algorithms

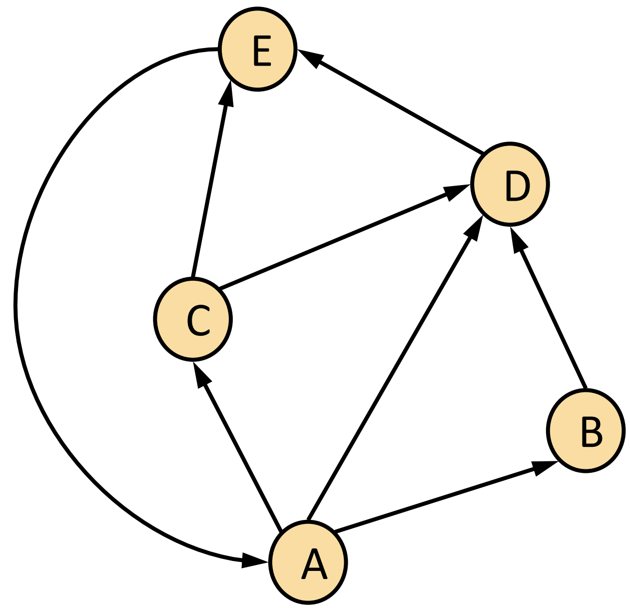
Digraphs

- A digraph is a graph whose edges are all directed
  - Short for “directed graph”
- Applications
  - one-way streets
  - flights
  - task scheduling



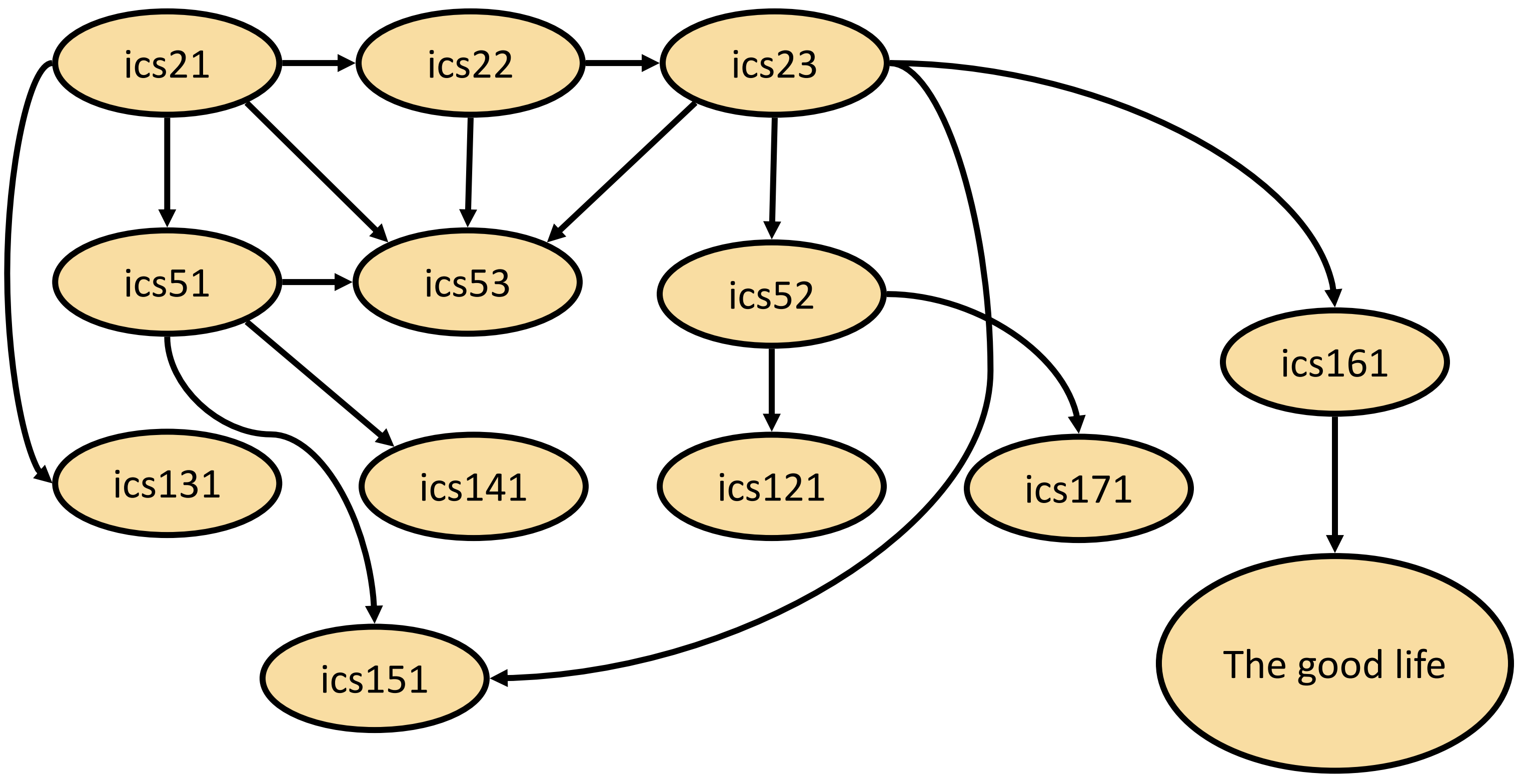
Digraph Properties

- A graph G=(V,E) such that
  - Each edge goes in one direction:
  - Edge (a,b) goes from a to b, but not b to a
- If G is simple, $m \leq n\,(n-1)$
- If we keep in-edges and out-edges in separate adjacency lists, we can perform listing of incoming edges and outgoing edges in time proportional to their size



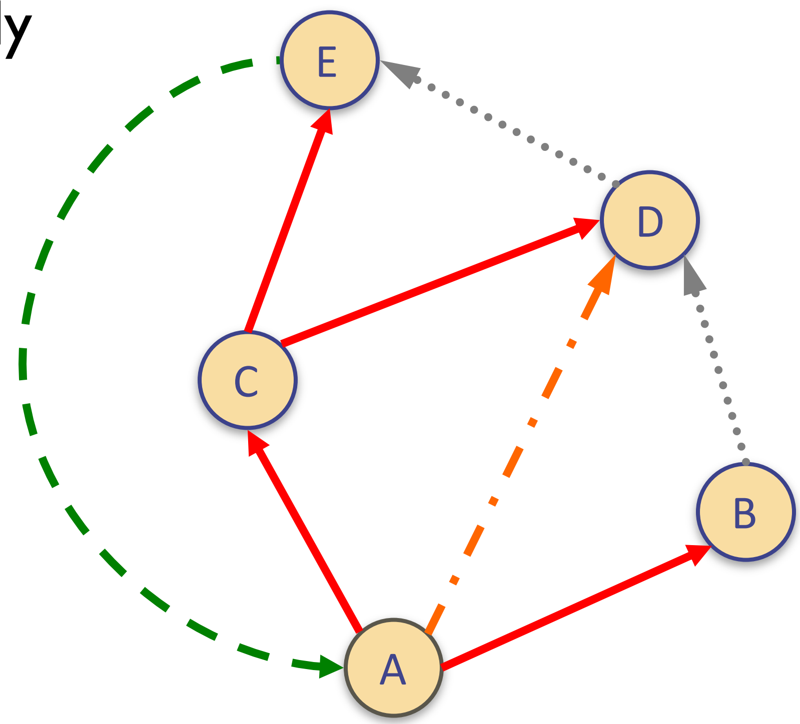
Digraph Application

- Scheduling: edge (a,b) means task a must be completed before b can be started



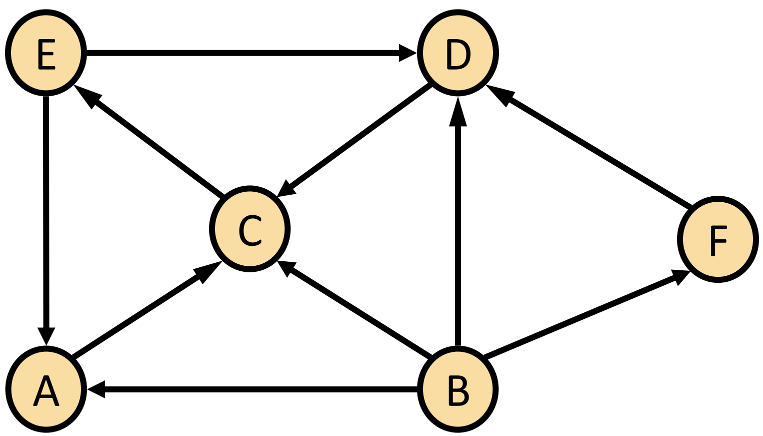
Directed DFS

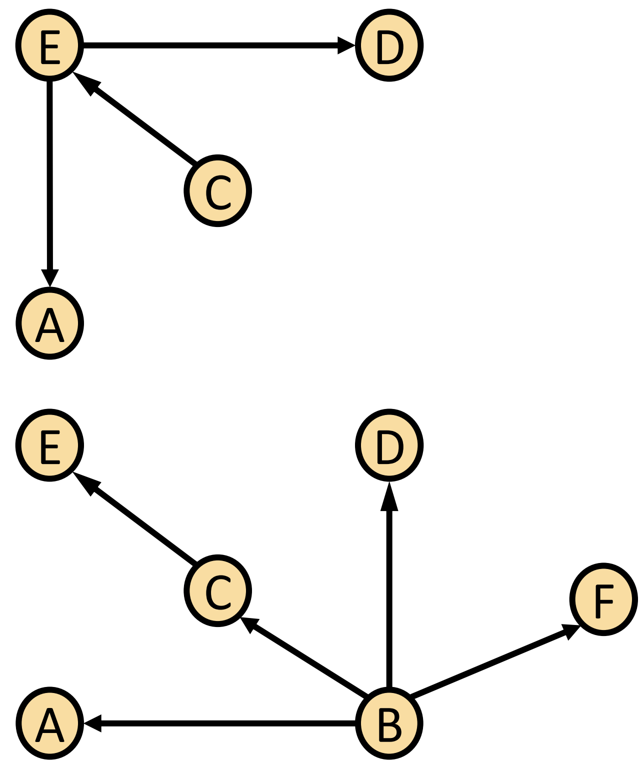
- We can specialize the traversal algorithms (DFS and BFS) to digraphs by traversing edges only along their direction
- In the directed DFS algorithm, we have four types of edges
  - discovery edges
  - back edges
  - forward edges
  - cross edges
- A directed DFS starting at a vertex s determines the vertices reachable from s



Reachability

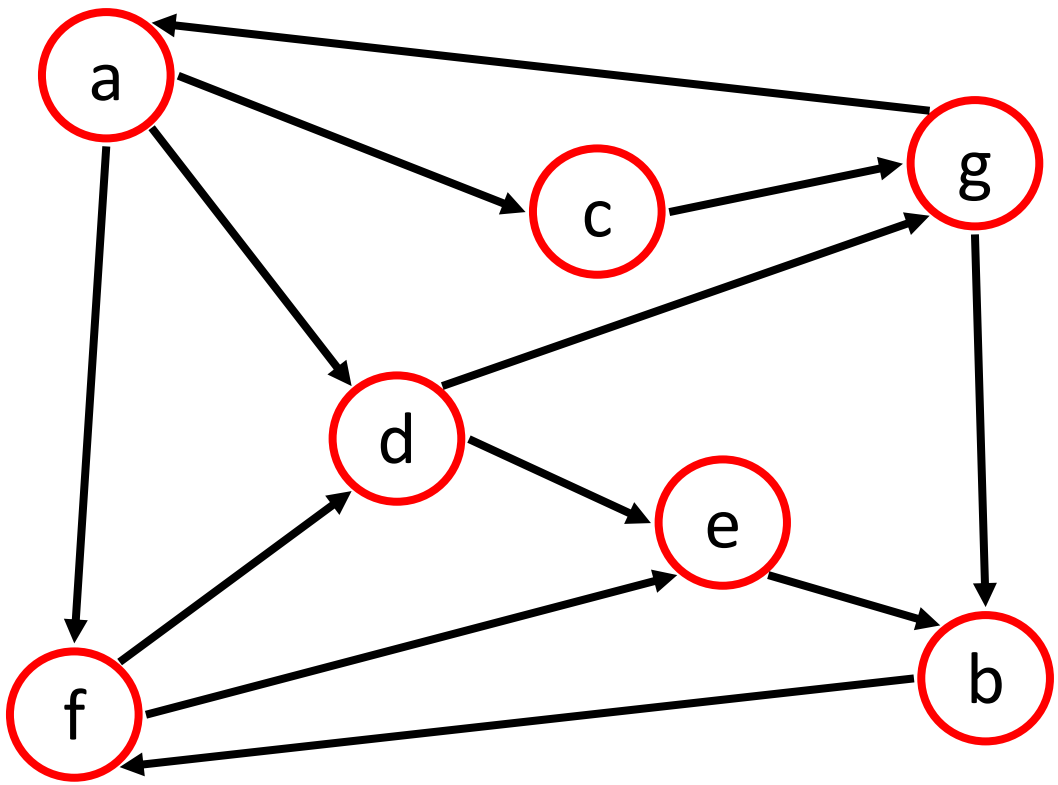
- DFS tree rooted at v: vertices reachable from v via directed paths





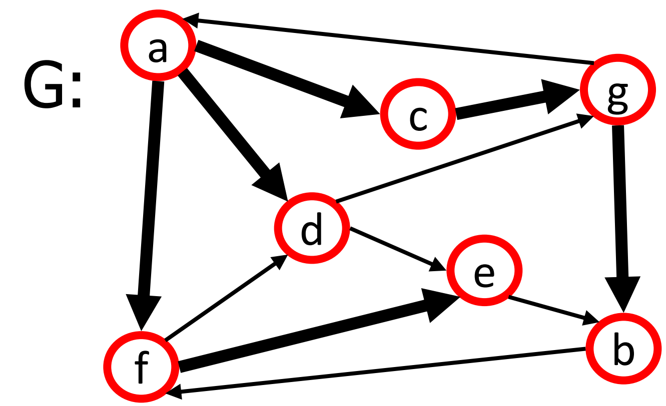
Strong Connectivity

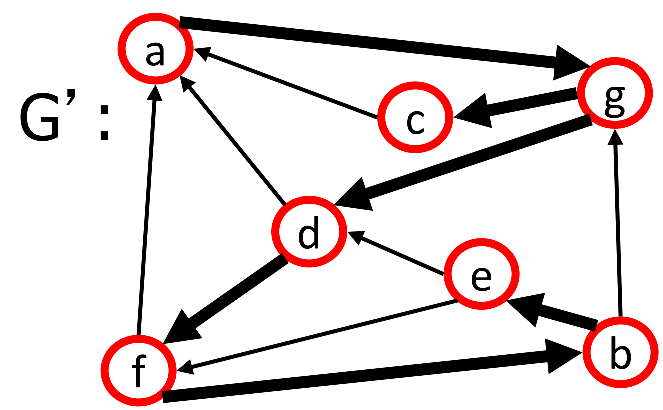
- Each vertex can reach all other vertices
  - Run directedDFS for every vertex : O(n(n+m))



Strong Connectivity Algorithm

- Pick any vertex v in G
- Perform a DFS from v in G
  - If there's a w not visited, print "no"
- Let G' be G with edges reversed
- Perform a DFS from v in G'
  - If there's a w not visited, print "no"
  - Else, print "yes"
- Running time: O(n+m)
  - Requires only two directedDFS





DAGs and Topological Ordering

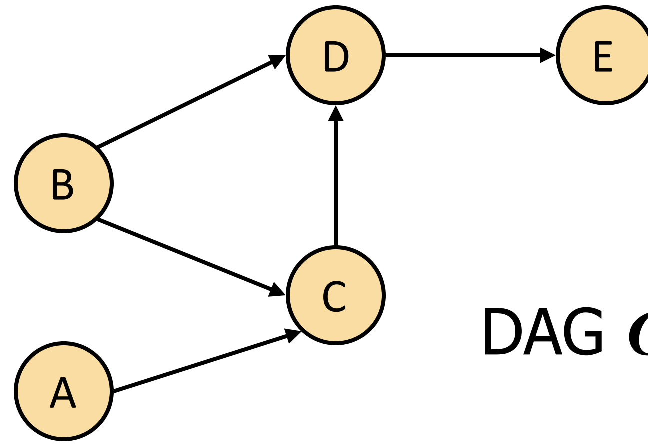
- A directed acyclic graph (DAG) is a digraph that has no directed cycles
- A topological ordering of a digraph is a numbering

$$v_1, \ldots, v_n$$

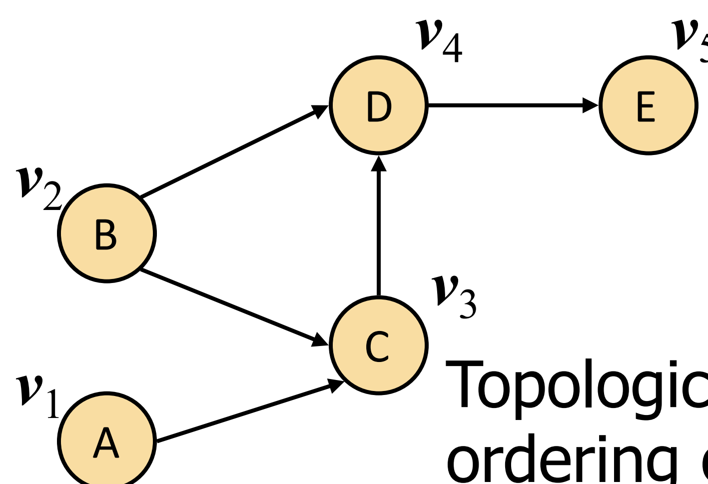
  of the vertices such that for every edge (v_i, v_j), we have $i < j$
- Example: in a task scheduling digraph, a topological ordering a task sequence that satisfies the precedence constraints

Theorem

A digraph admits a topological ordering if and only if it is a DAG



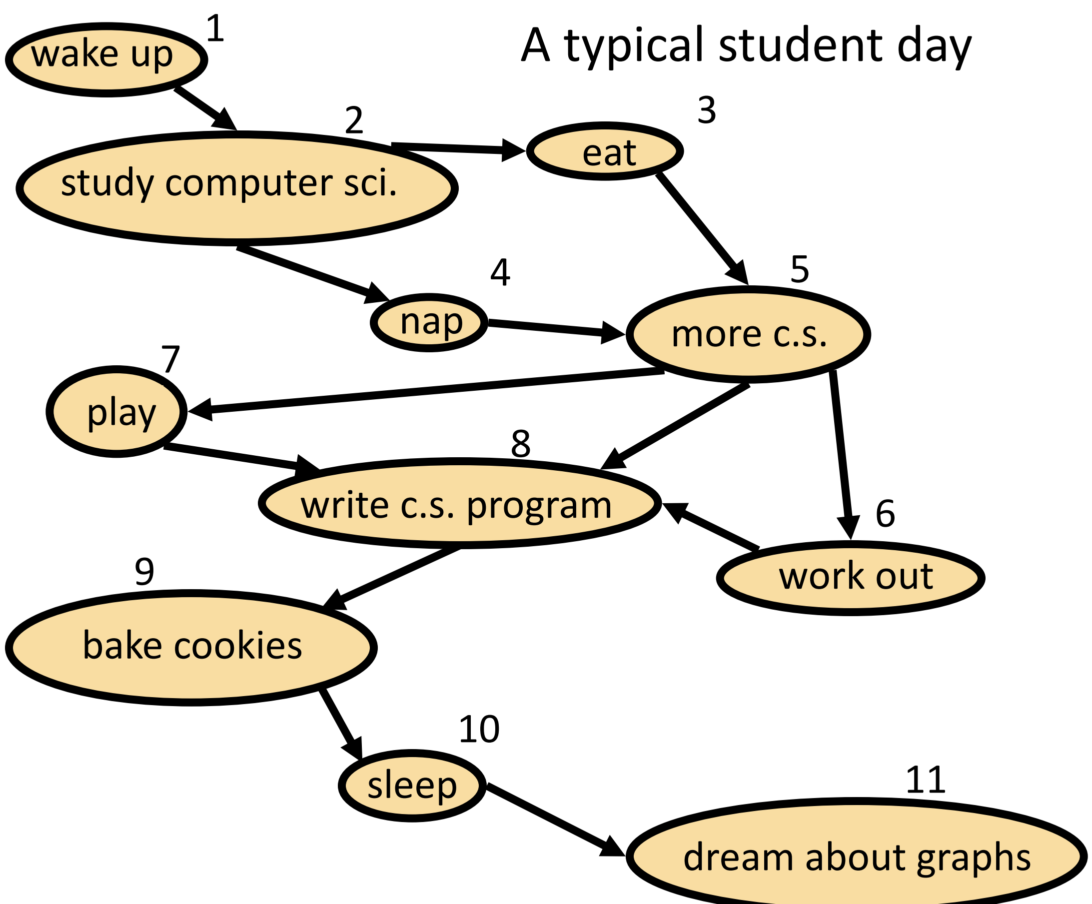
DAG G



Topological ordering of G

Topological Sorting

- Number vertices, so that (u,v) in E implies u < v



Q: is topological sorting unique?

Algorithm for Topological Sorting

- Running time: O(n + m)

```
Algorithm TopologicalSort(G)
    H ← G  // Temporary copy of G
    n ← G.numVertices()
    while H is not empty do
        Let v be a vertex with no outgoing edges
        Label v ← n
        n ← n – 1
        Remove v from H
```

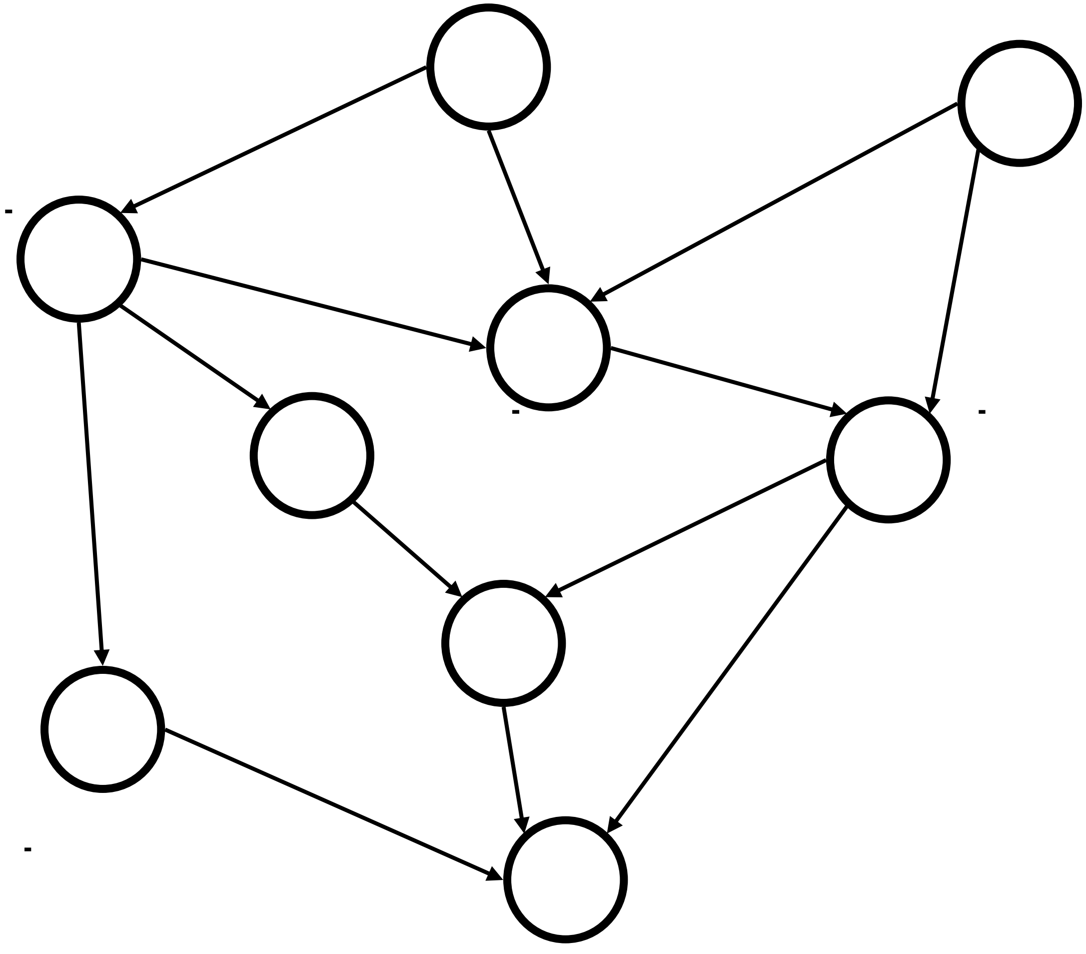
Implementation with DFS

- Simulate the algorithm by using depth-first search
- O(n+m) time.

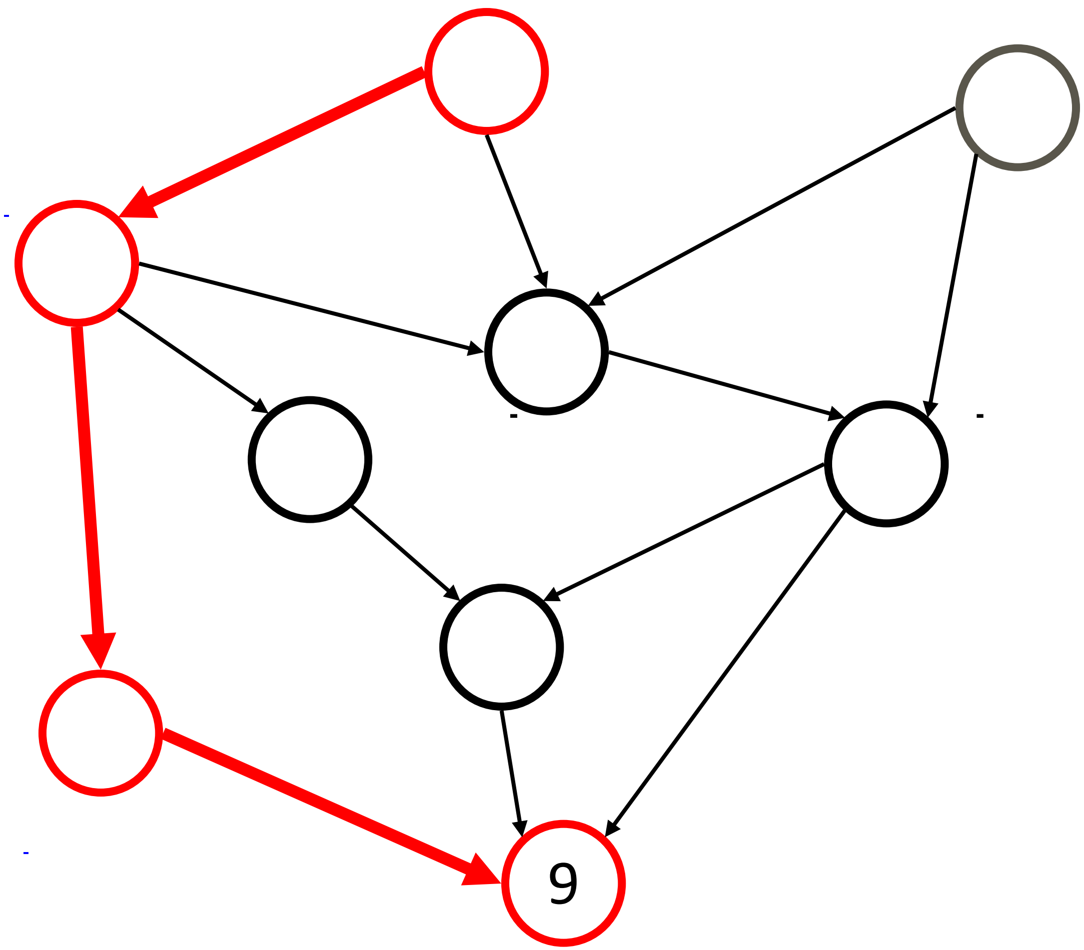
```
Algorithm topologicalDFS(G)
    Input dag G
    Output topological ordering of G
  n ← G.numVertices()
  for all u ιν G.vertices()
     u.setLabel(UNEXPLORED)
  for all v ιν G.vertices()
     if v.getLabel() = UNEXPLORED
        topologicalDFS(G, v)
```

```
Algorithm topologicalDFS(G, v)
    Input graph G and a start vertex v of G
    Output labeling of the vertices of G
       in the connected component of v
    v.setLabel(VISITED)
    for all e ιν v.outEdges()
       { outgoing edges }
       w ← e.opposite(v)
       if w.getLabel() = UNEXPLORED
          { e is a discovery edge }
          topologicalDFS(G, w)
       else
          { e is a forward or cross edge }
    Label v with topological number n
     n ← n - 1
```

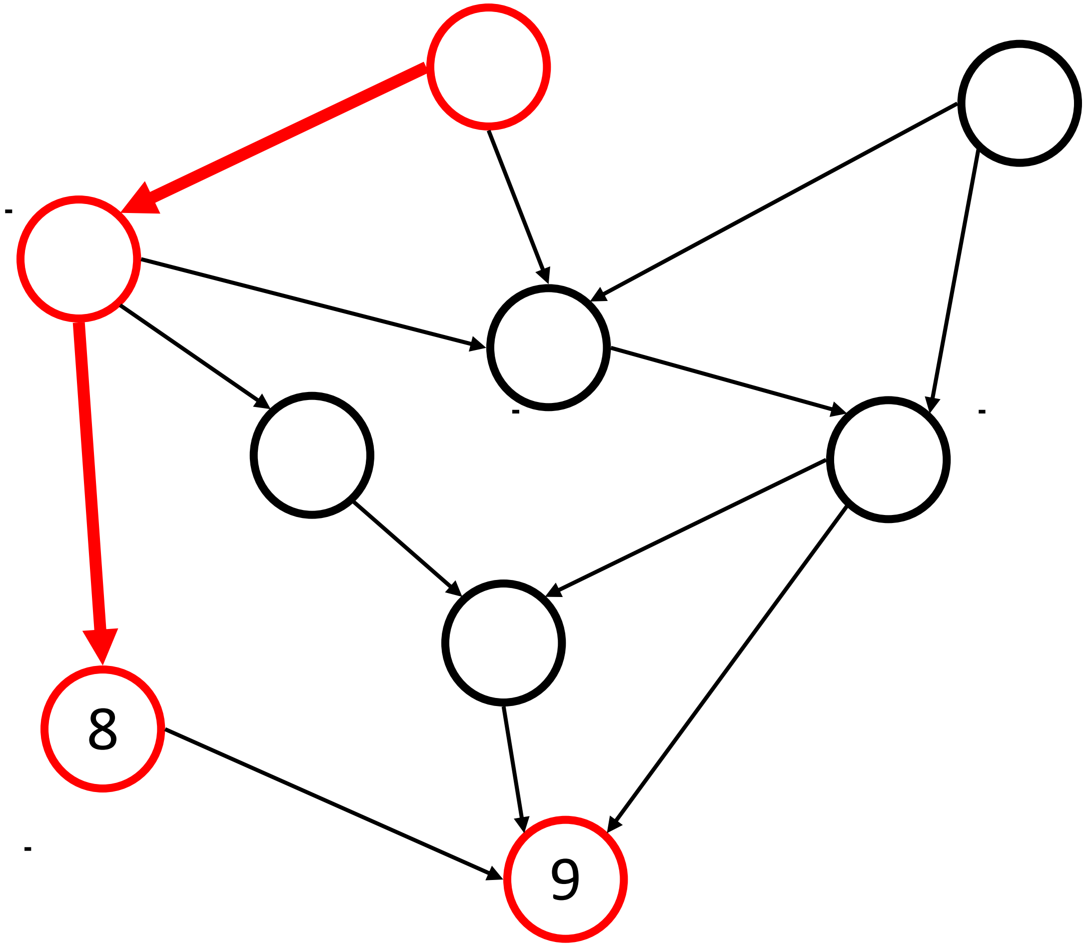
Topological Sorting Example



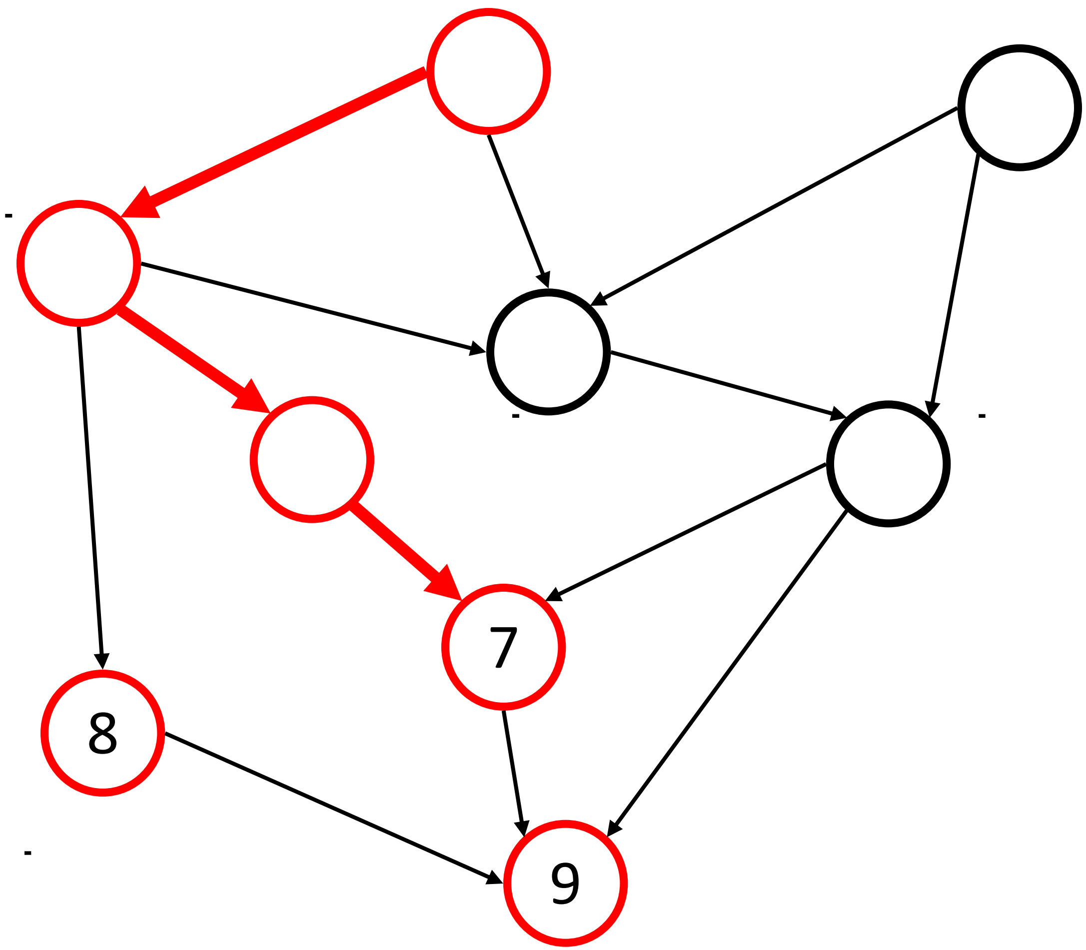
Topological Sorting Example



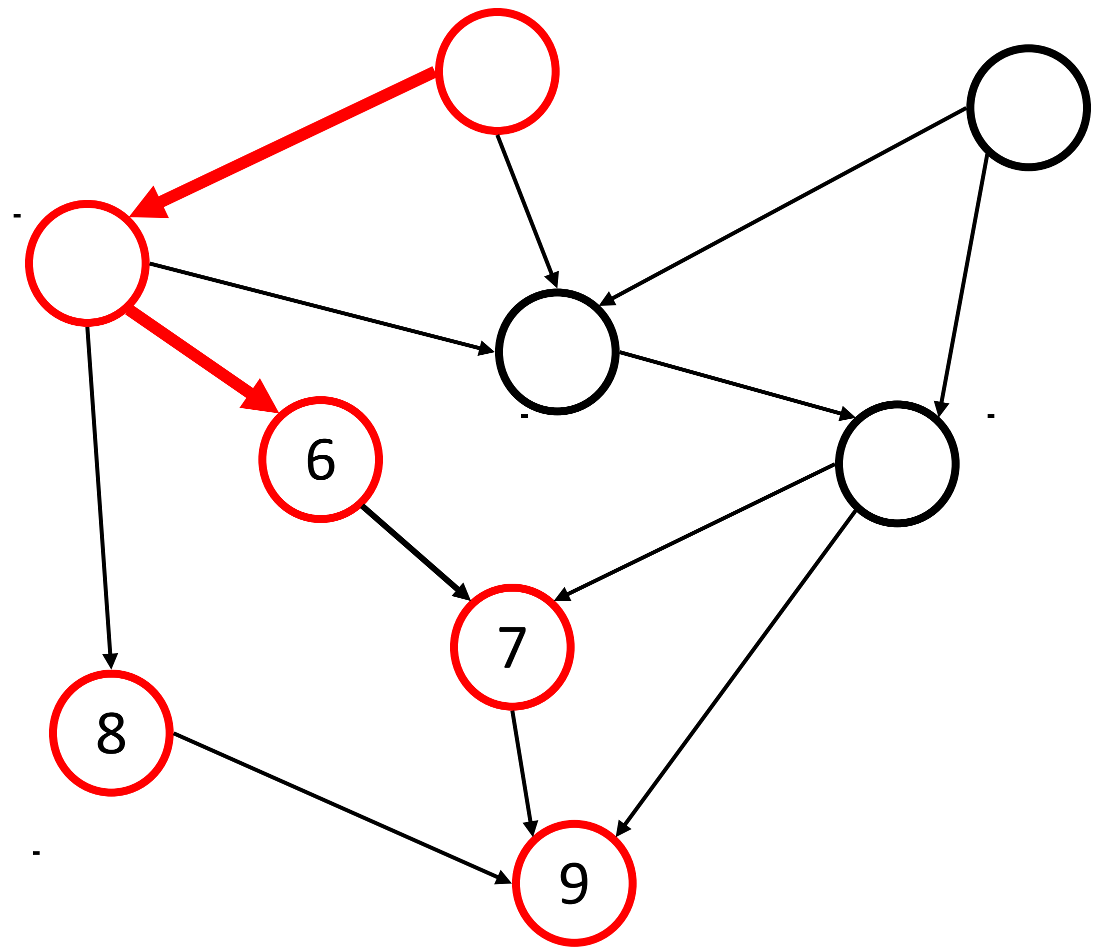
Topological Sorting Example



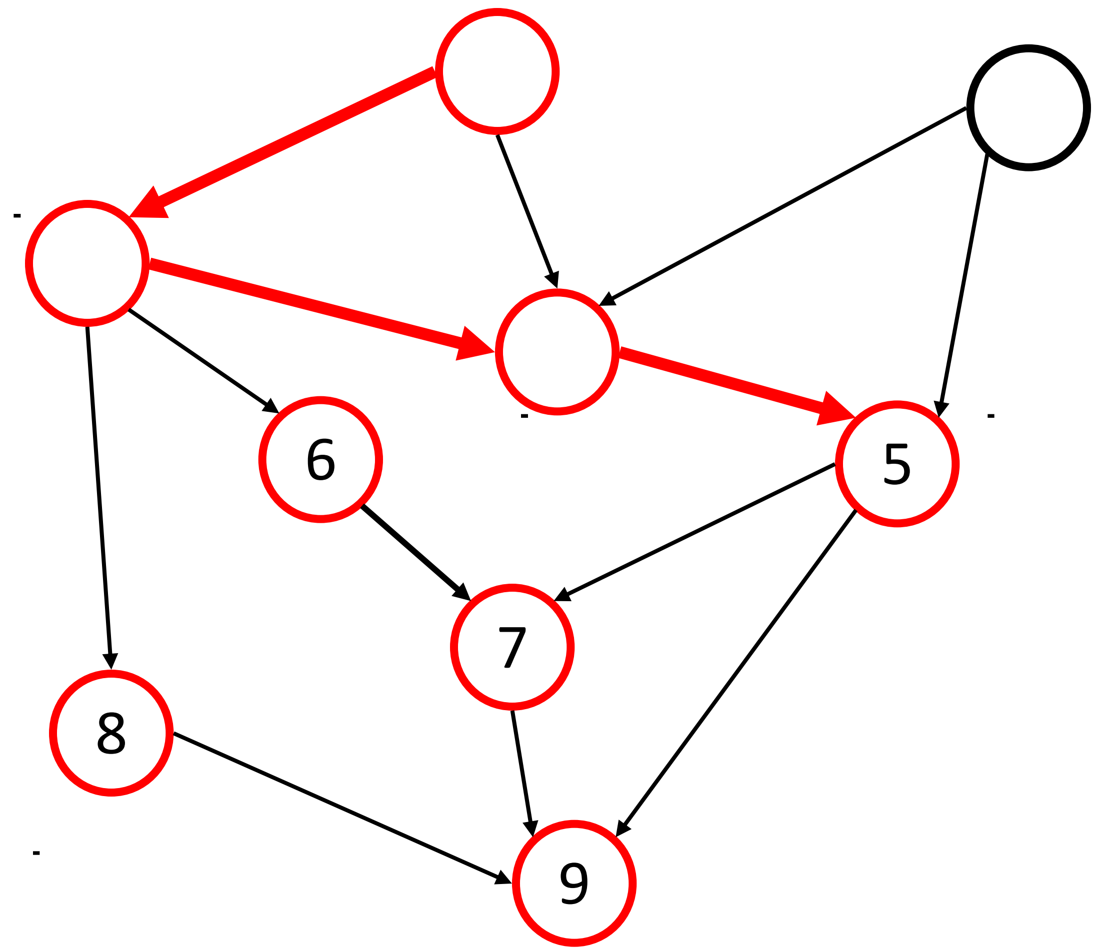
Topological Sorting Example



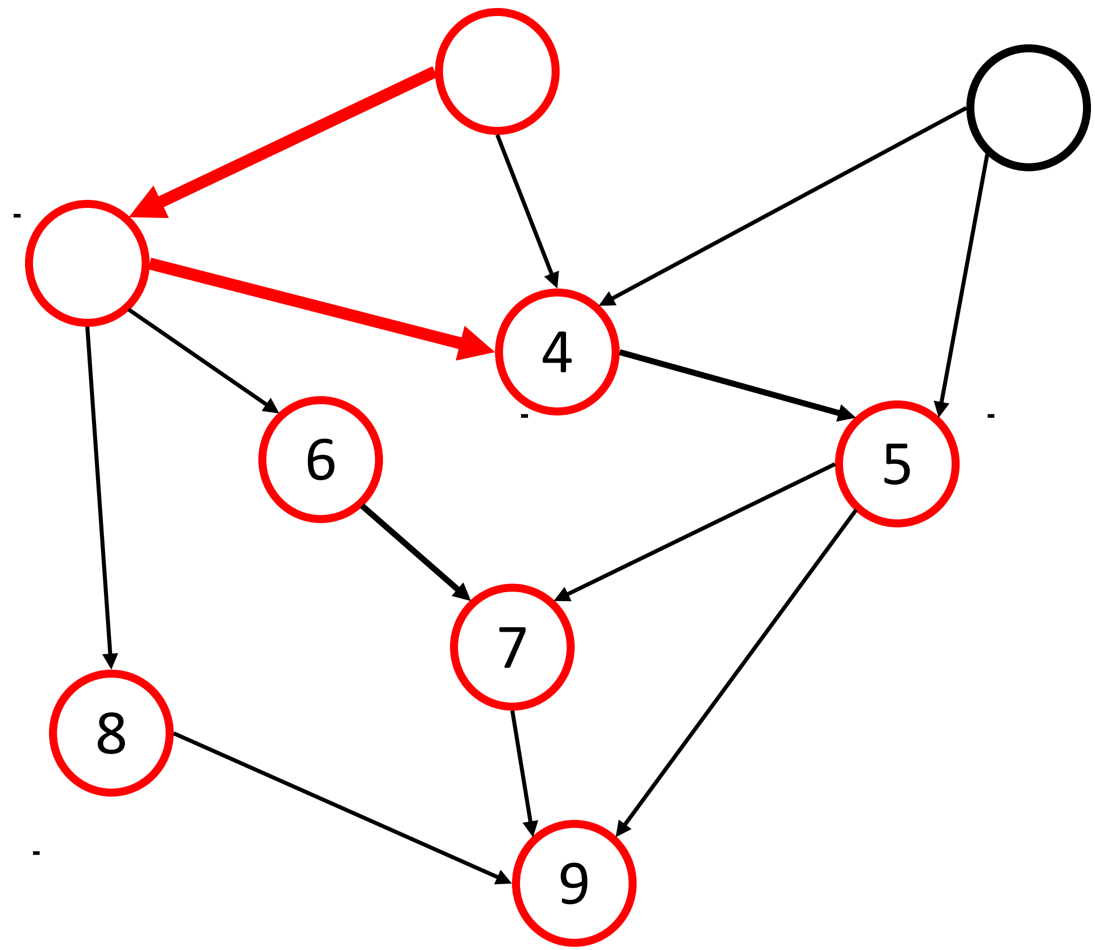
Topological Sorting Example



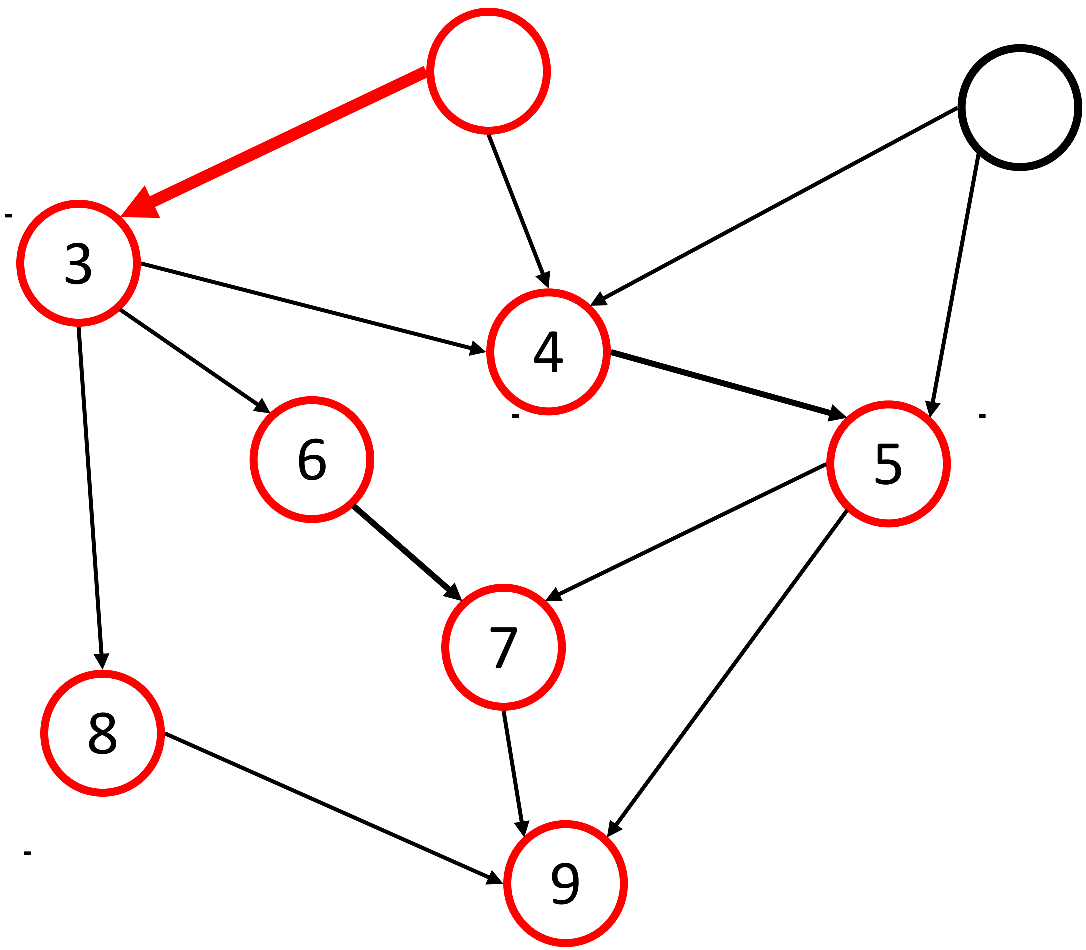
Topological Sorting Example



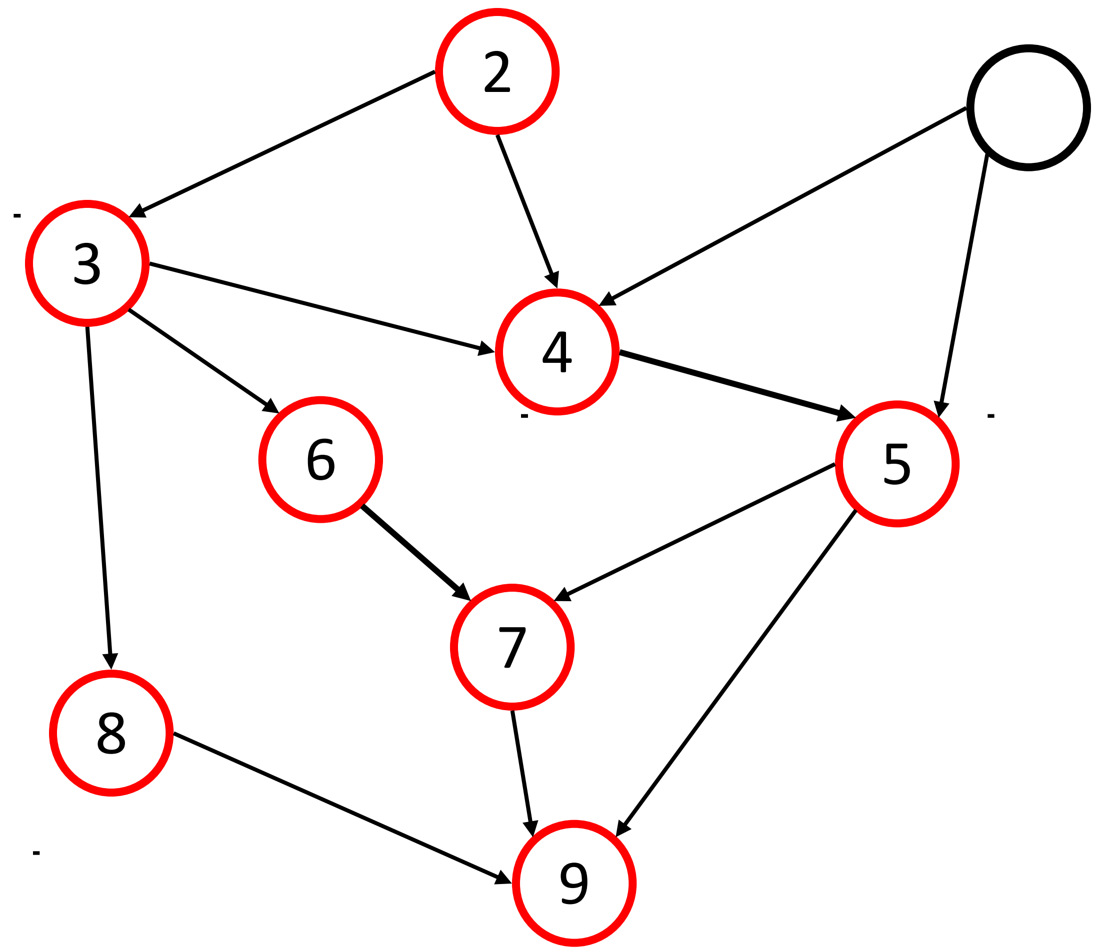
Topological Sorting Example



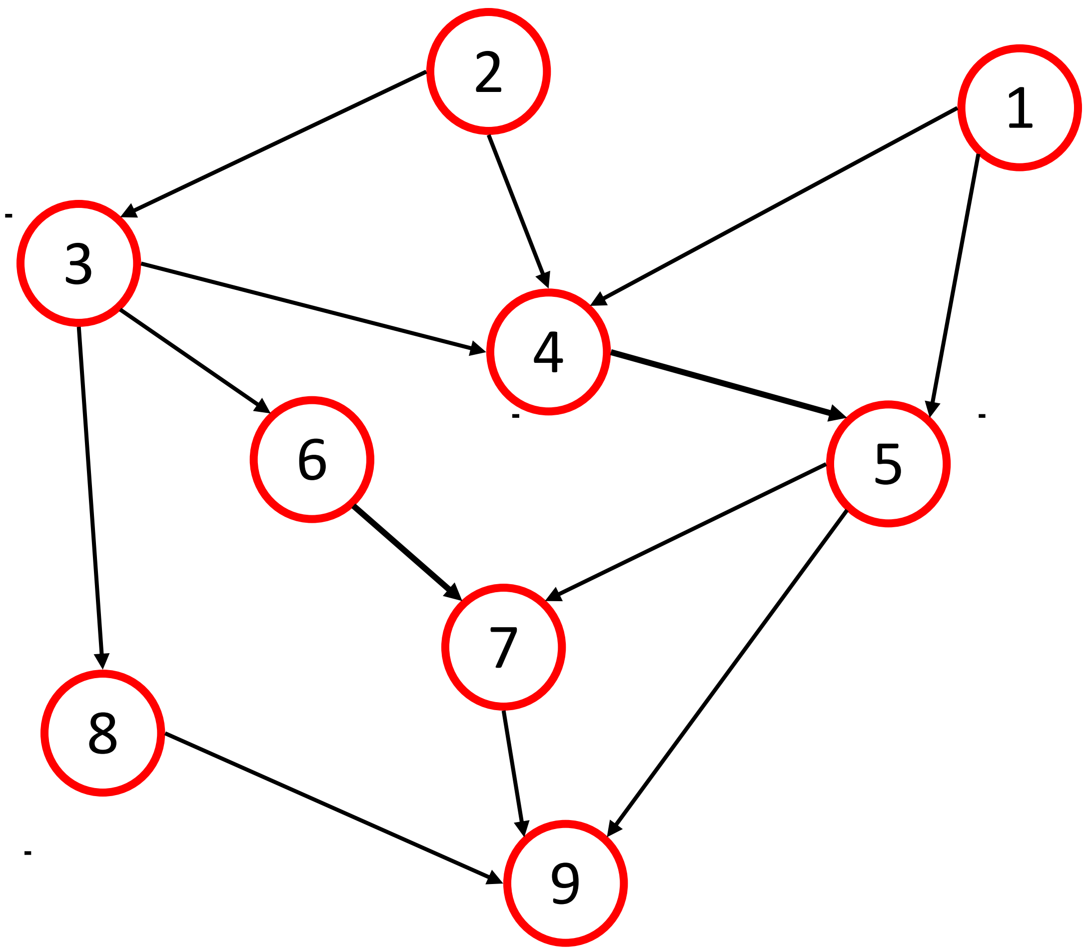
Topological Sorting Example



Topological Sorting Example



Topological Sorting Example



Outline

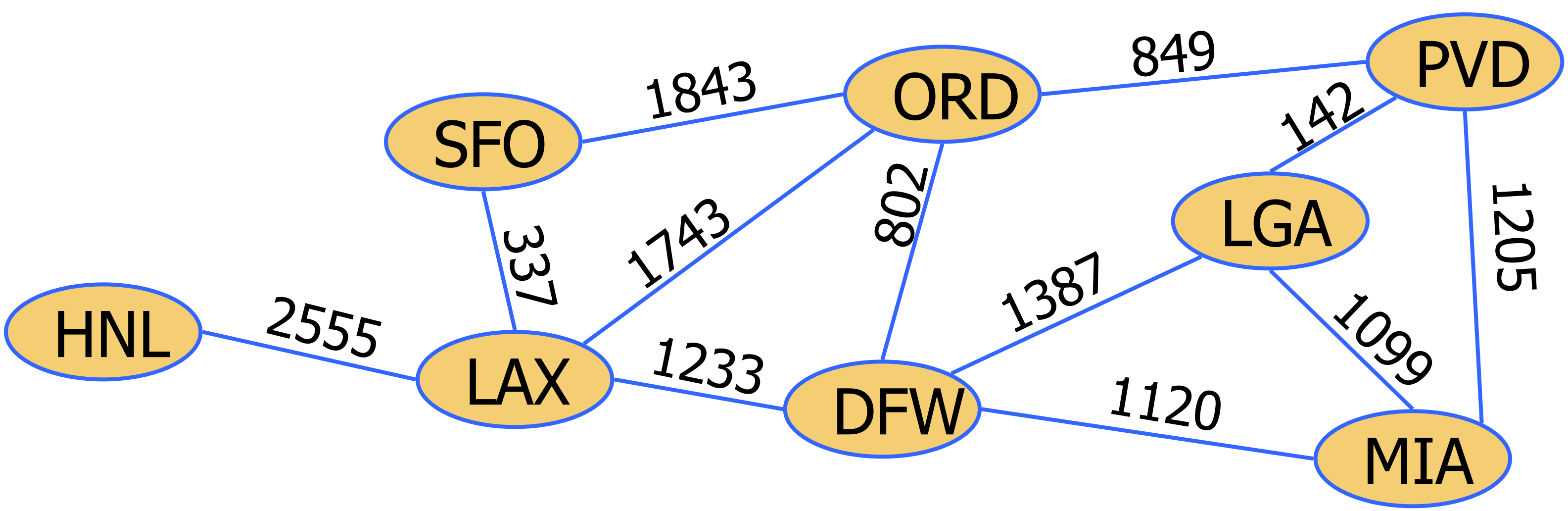
- Directed graphs
- Shortest path algorithms
  - Dijkstra
  - Bellman-ford

Shortest Paths

- Cities : vertices
- Roads : edges
- Cost : edge length
- Source : start vertex
- Destination : end vertex
- Graph : directed
  - Allow one-way road

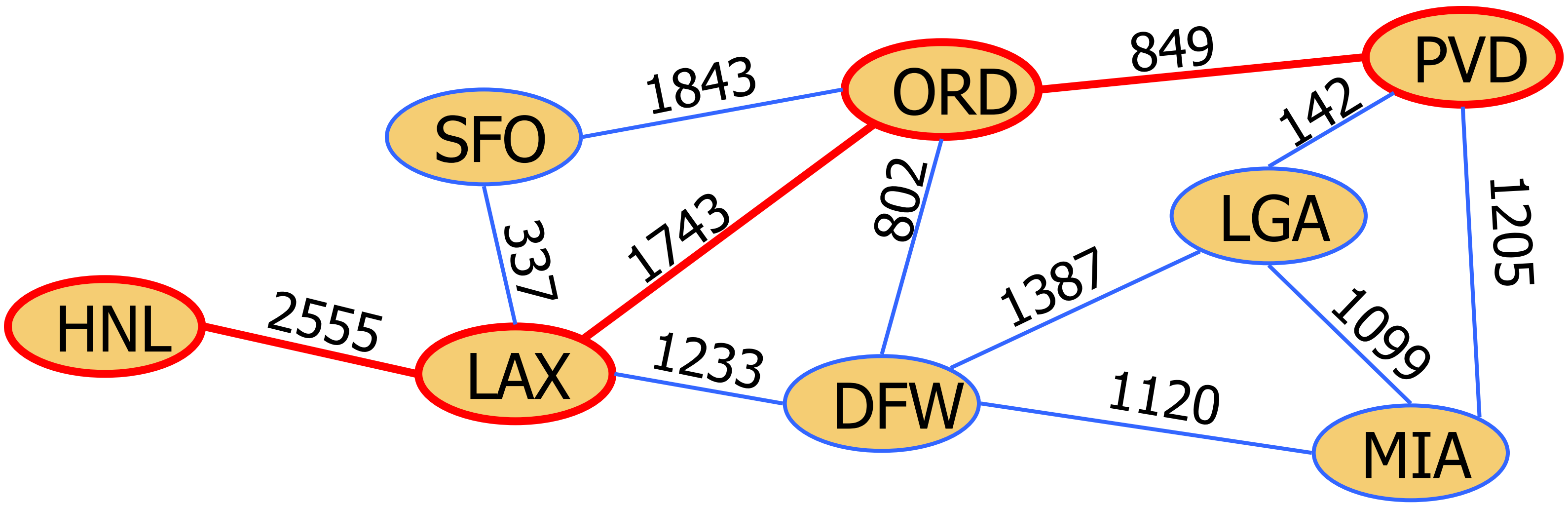
Weighted Graphs

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- Edge weights may represent, distances, costs, etc.
- Example:
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



Shortest Paths

- Given a weighted graph and two vertices u and v, we want to find a path of minimum total weight between u and v.
  - Length of a path is the sum of the weights of its edges.
- Example:
  - Shortest path between Providence and Honolulu



Shortest Path Problems

- Single-source shortest path problems
  - Find shortest paths from a **source vertex** s (which is given) to all other vertices in a graph.
  - If we just need to find a shortest path to one particular vertex, you can stop the algorithm earlier.
  - Solution: a shortest-path tree rooted at s
    - which is also a spanning tree of G.
  - Algorithms: Dijkstra's algorithm and Bellman-Ford algorithm
- All pairs shortest path problems
  - Find shortest paths between **every pair** of vertices in a graph
  - Solution: a data structure from which shortest paths can be quickly constructed by some path reconstruction algorithms.
  - Algorithms: Floyd-Warshall algorithm and Johnson's algorithm

Dijkstra Shortest Path Algorithm

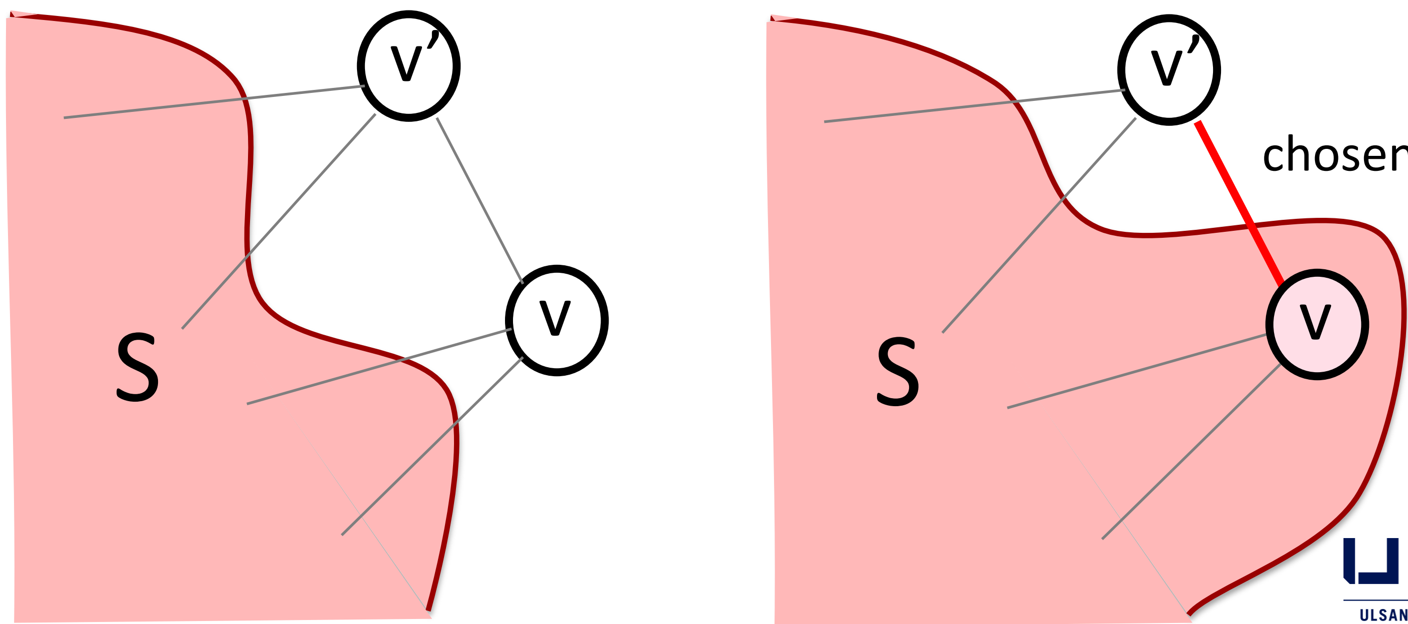
- For graphs with non-negative weights
- Algorithm
  - Let D[v] be the length of a *currently best, known path* from s to v
    - Initially, D[s] = 0 and D[v] = ∞ for all other vertices.
  - Let S be a set of visited vertices.
    - Initially, S is empty.
  - Repeat the following steps until all vertices are added to S.
    - Among all vertices not in S, choose the vertex v with the smallest D[v] and add v to S
    - **Edge Relaxation**:
      For every vertex v' adjacent to v and v' is not in S, update D[v']:

$$D[v'] = \min(D[v'], D[v] + \text{weight}(v, v'))$$

- The algorithm will eventually terminate since S is monotonically expanding.

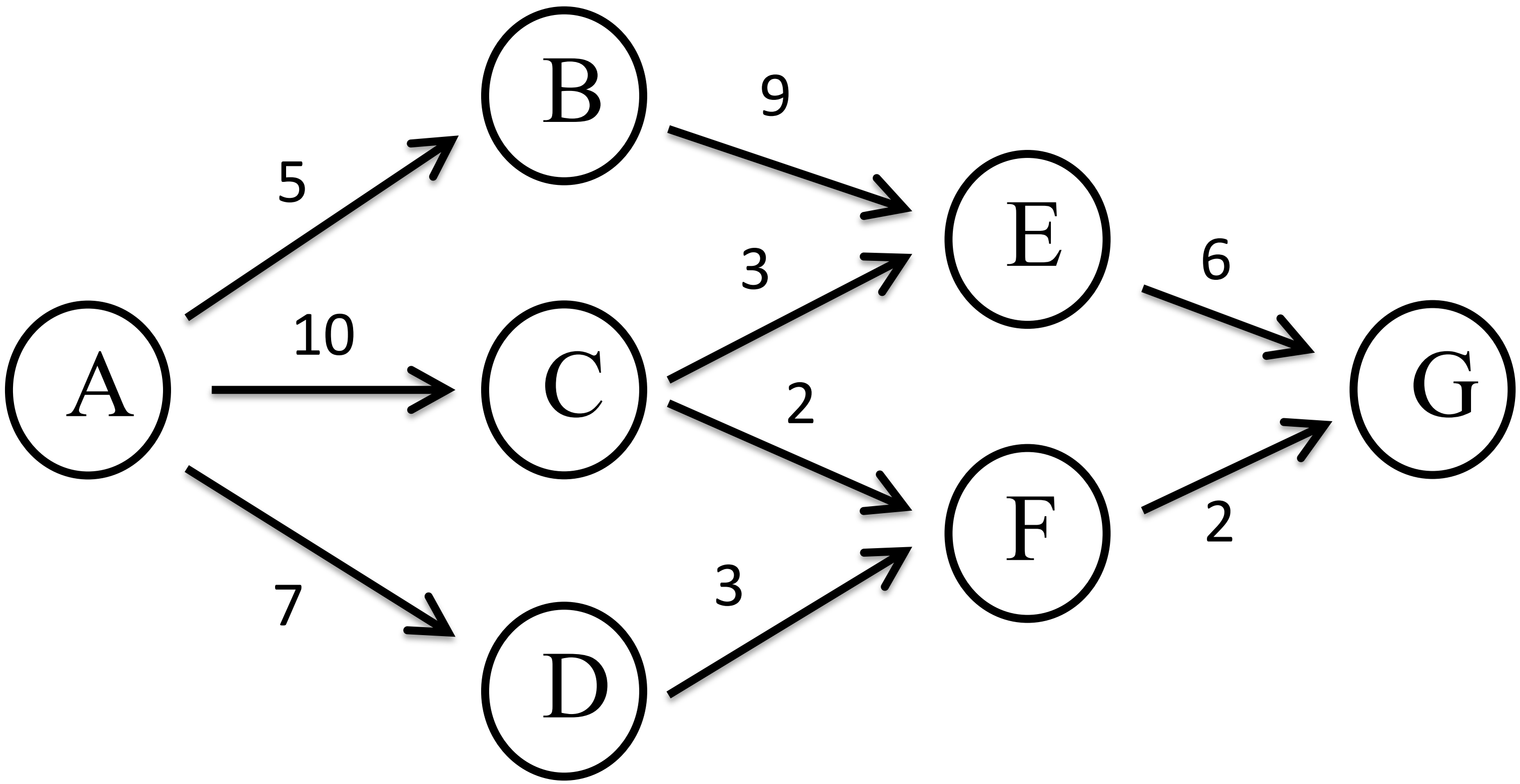
Edge Relaxation

- After a vertex v is added to S, check its adjacent neighbor v' if v reduces D[v'] (i.e., the path that goes through v is shorter than the previous best path that yields D[v'])
  - D[v'] = min (D[v'], D[v] + weight(v, v'))
- If the value of D[v'] is updated, mark the edge (v, v') as "chosen"
  - If v' has another incoming edge that has been marked as chosen previously, unmark it.
  - When v' is added to S, the chosen incoming edge of v' will be part of the shortest-path tree.



Dijkstra's Algorithm

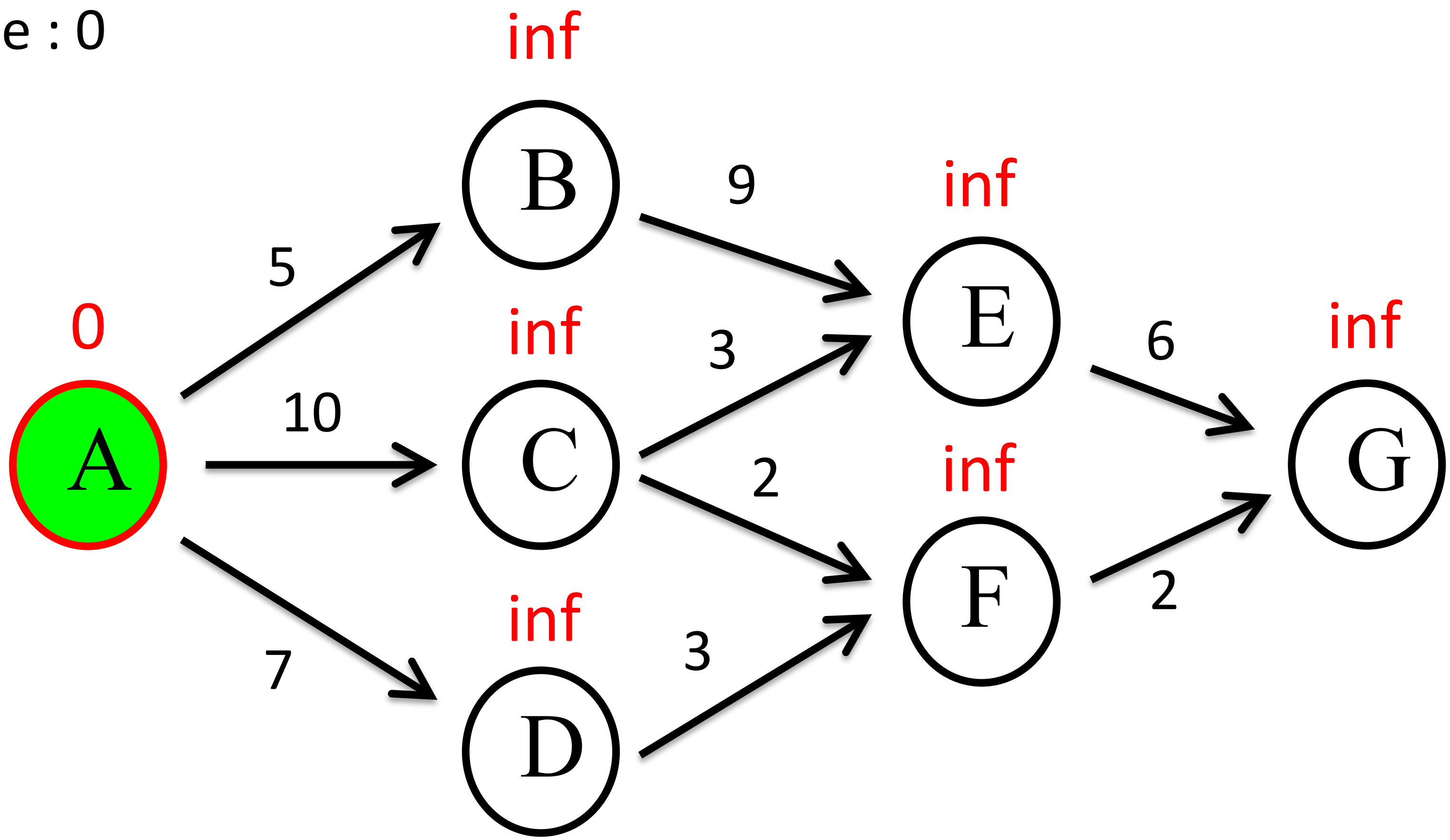
Source : A



S : {}

Dijkstra's Algorithm

Source : 0

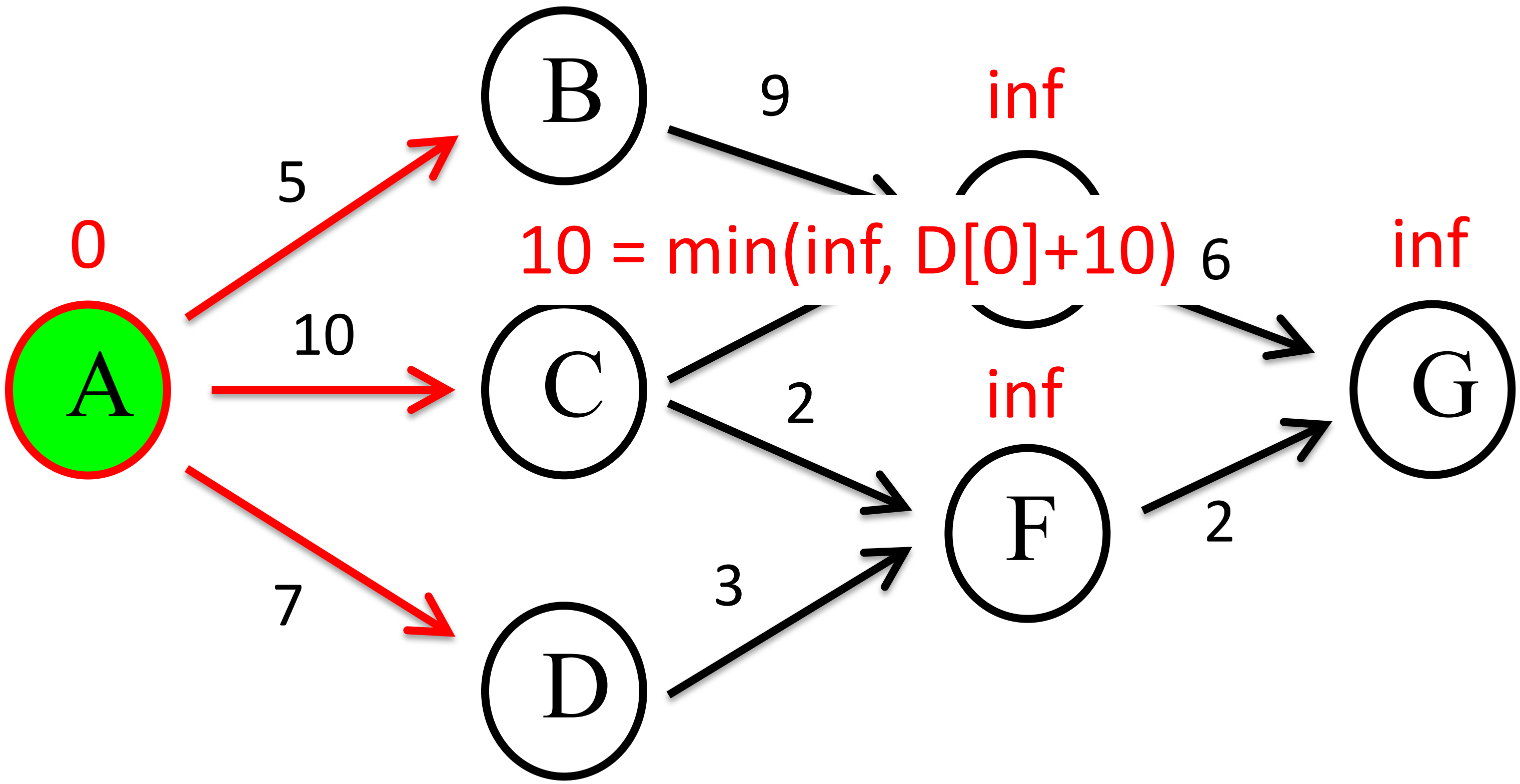


S : { A }

Dijkstra's Algorithm

Source : 0

5 = min(inf, D[0]+5)

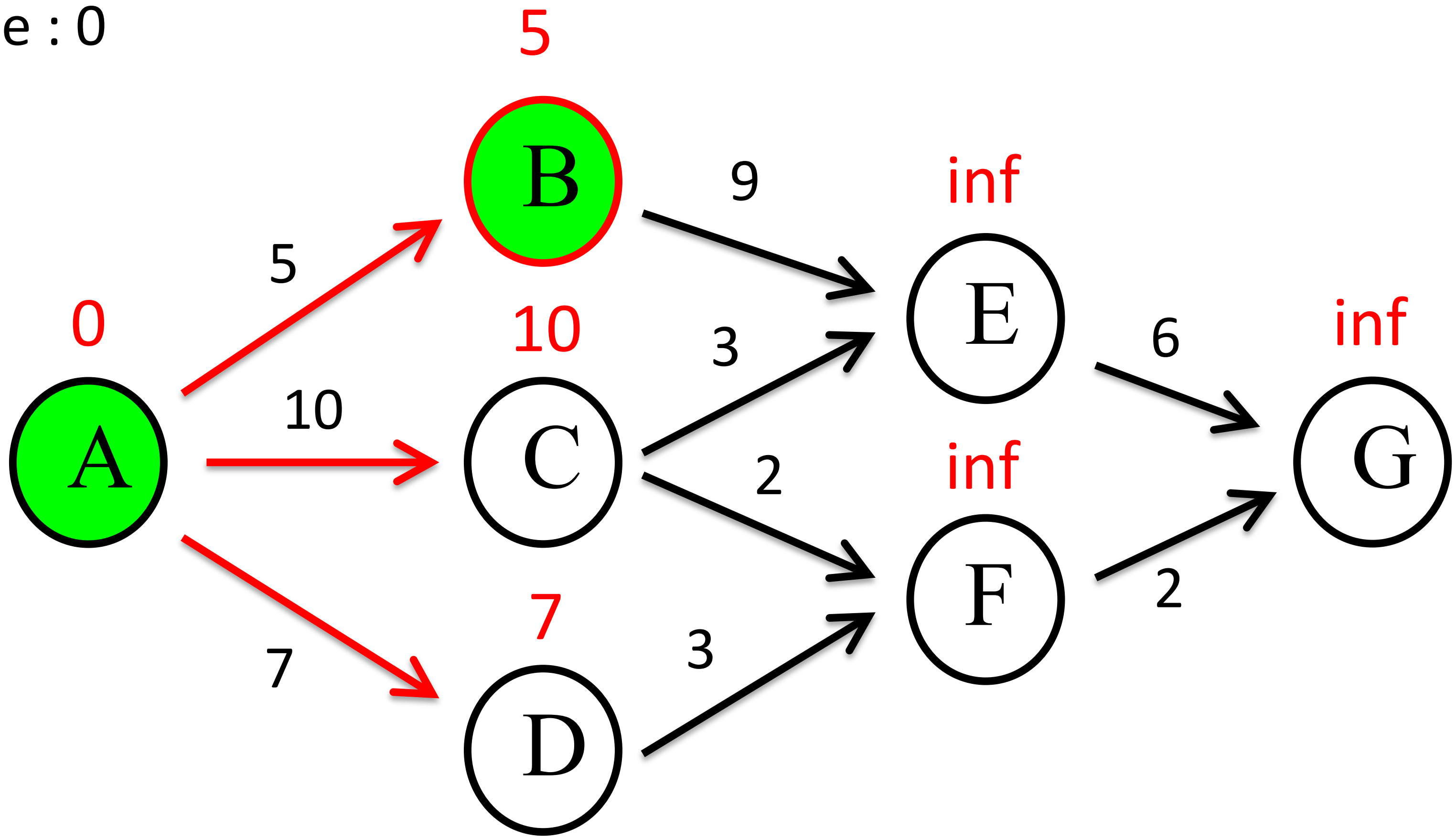


7 = min(inf, D[0]+7)

S : { A }

Dijkstra's Algorithm

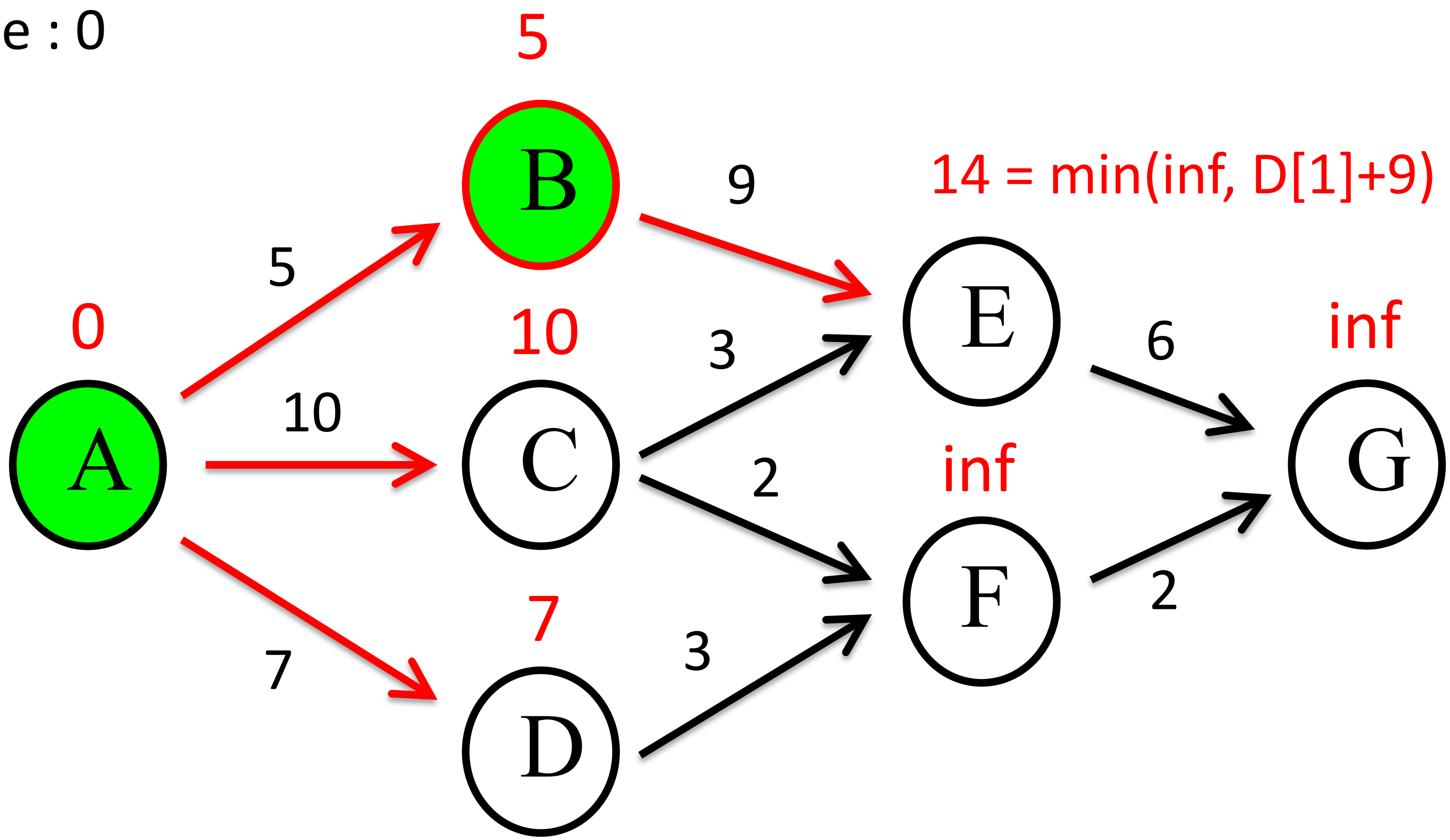
Source : 0



S : { A, B }

Dijkstra's Algorithm

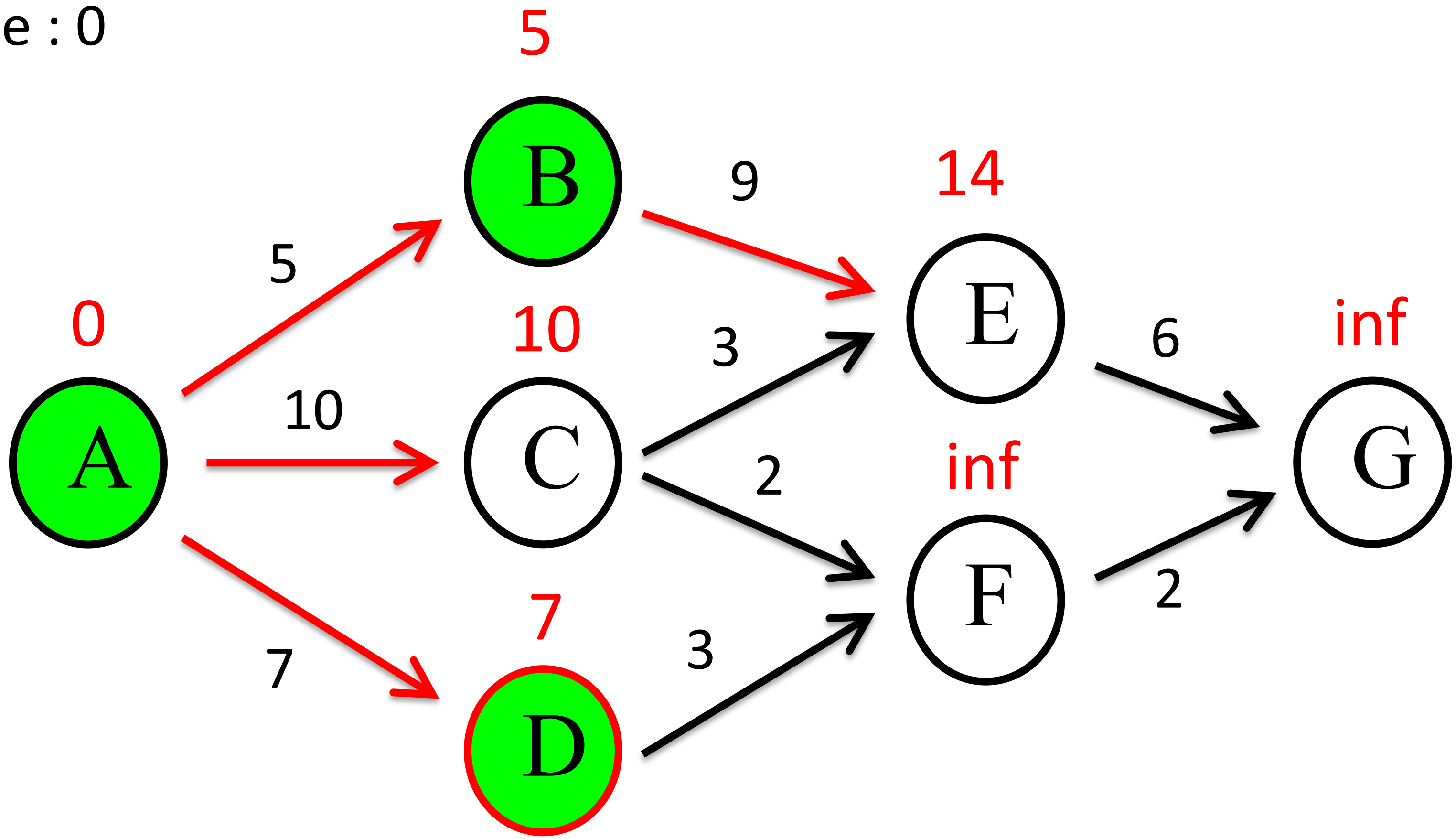
Source : 0



S : { A, B }

Dijkstra's Algorithm

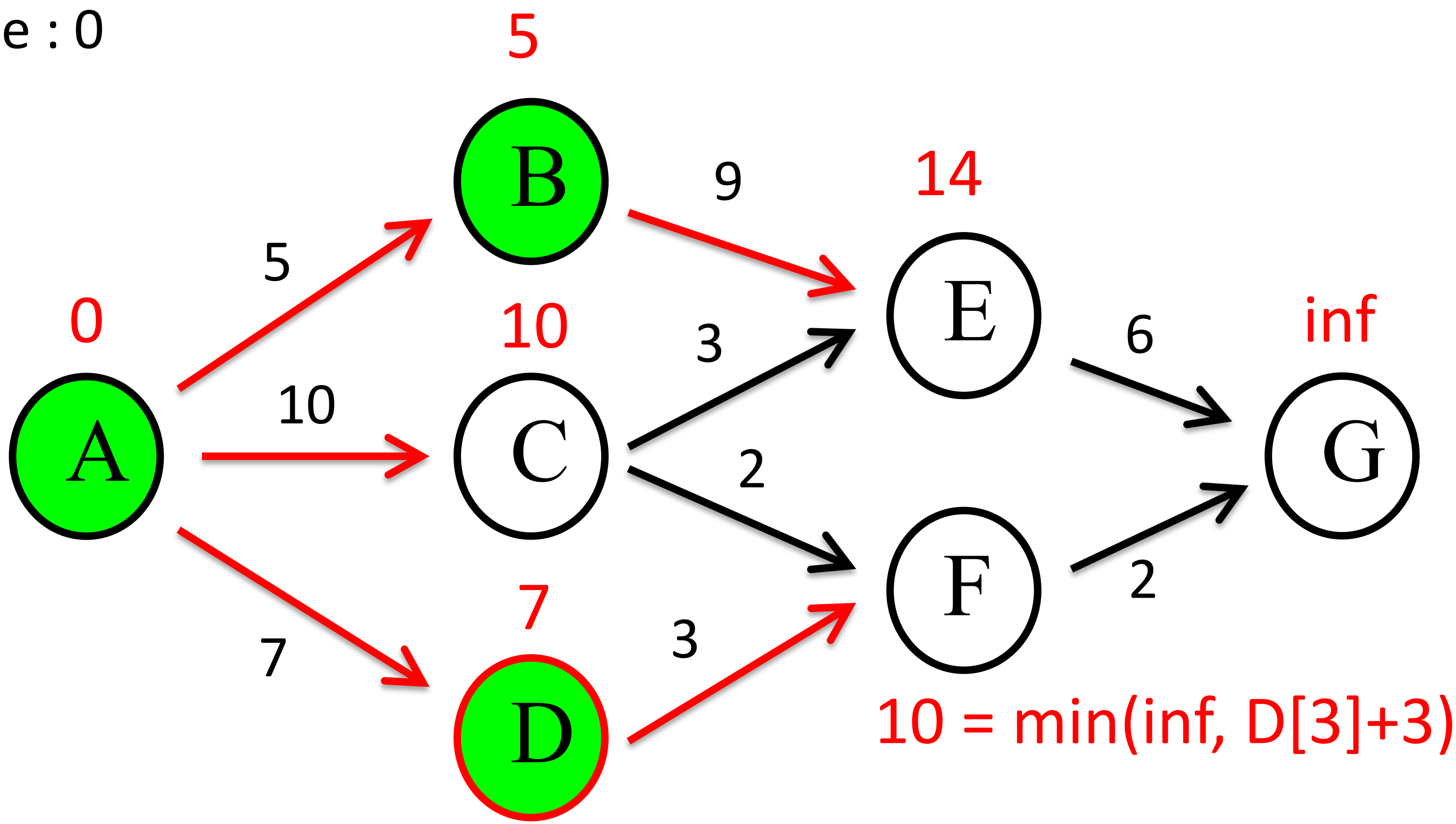
Source : 0



S : {A, B, D}

Dijkstra's Algorithm

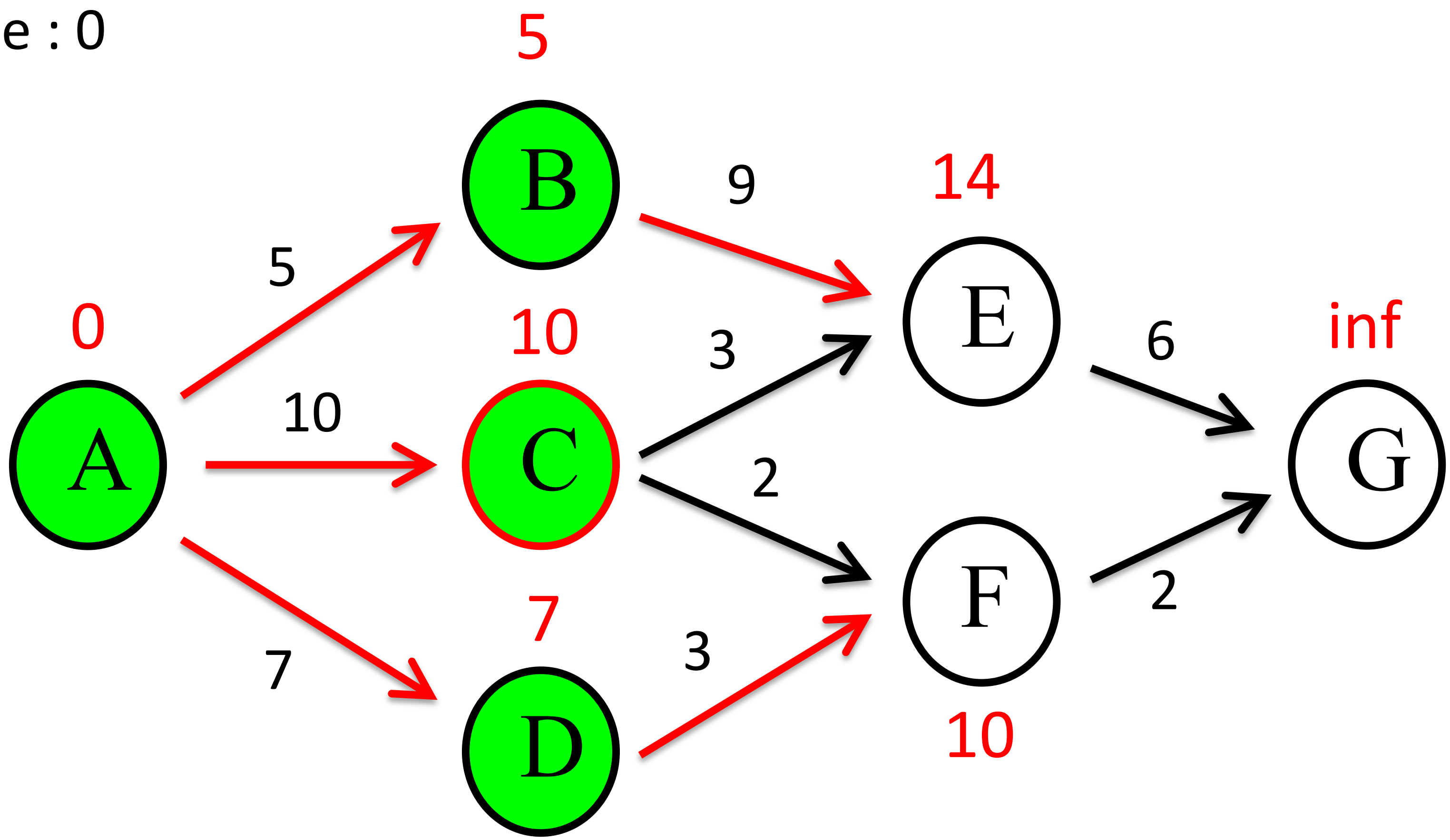
Source : 0



S : { A, B, D }

Dijkstra's Algorithm

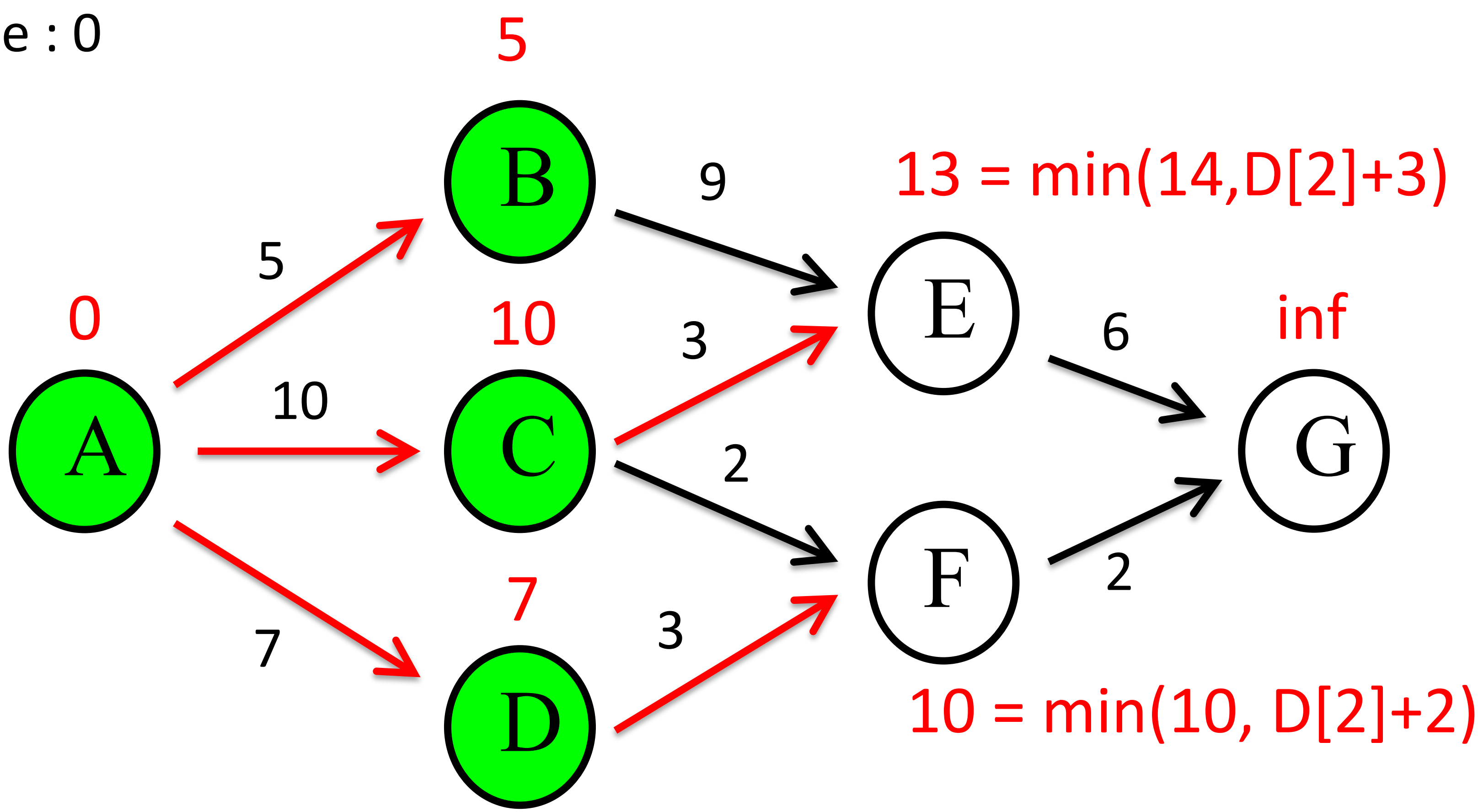
Source : 0



S : { A, B, C, D }

Dijkstra's Algorithm

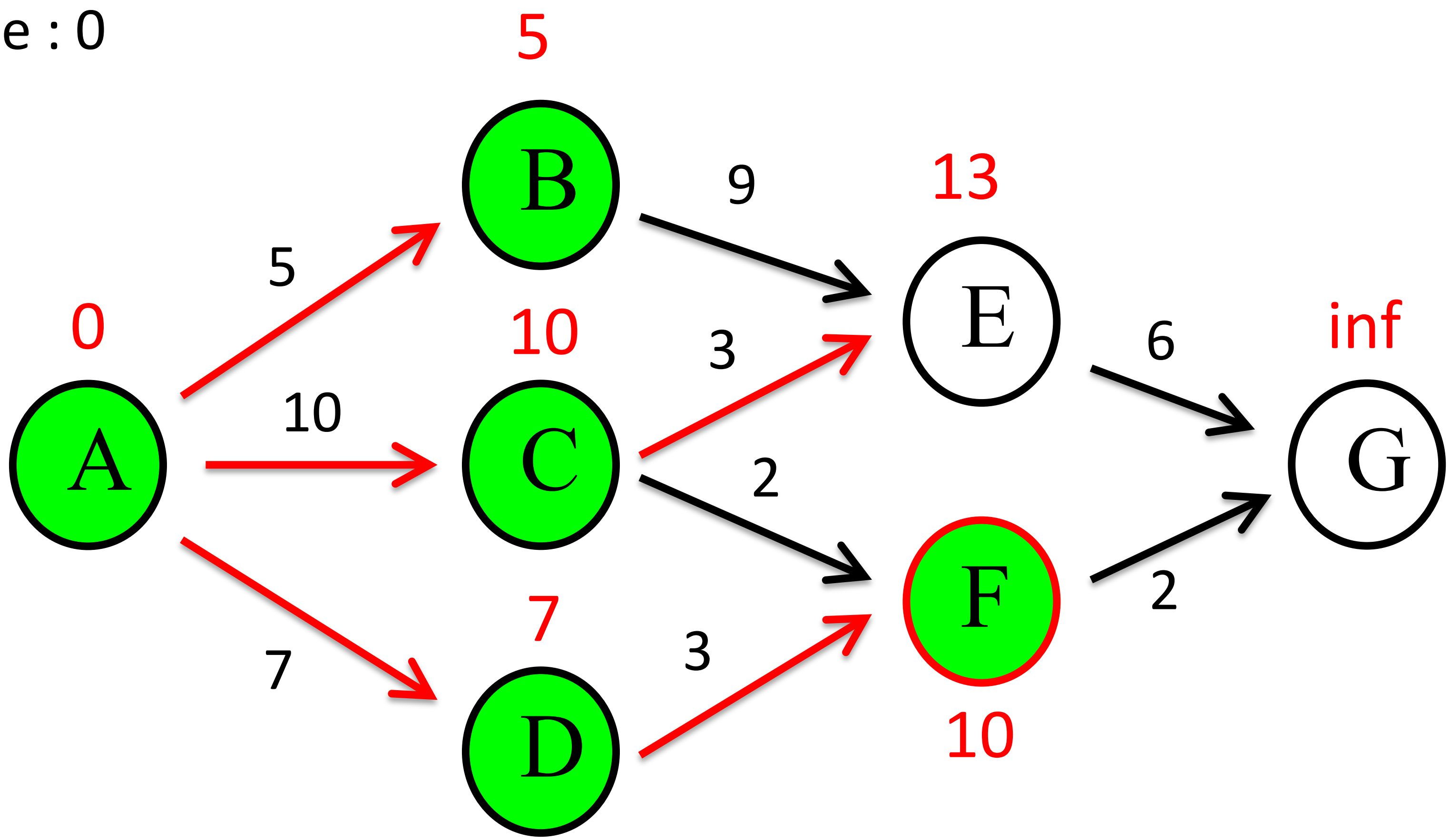
Source : 0



S : { A, B, C, D }

Dijkstra's Algorithm

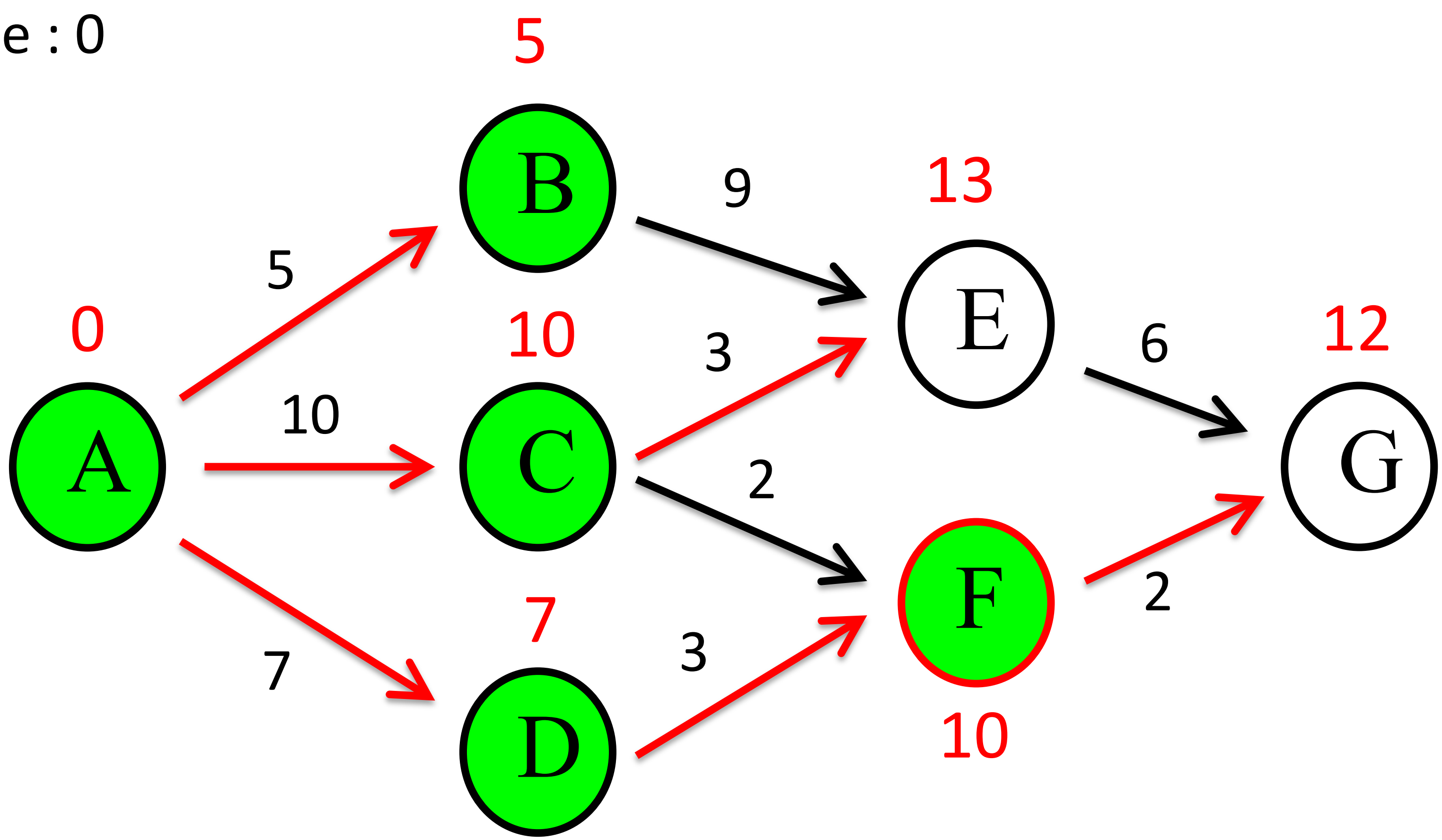
Source : 0



S : { A, B, C, D, F }

Dijkstra's Algorithm

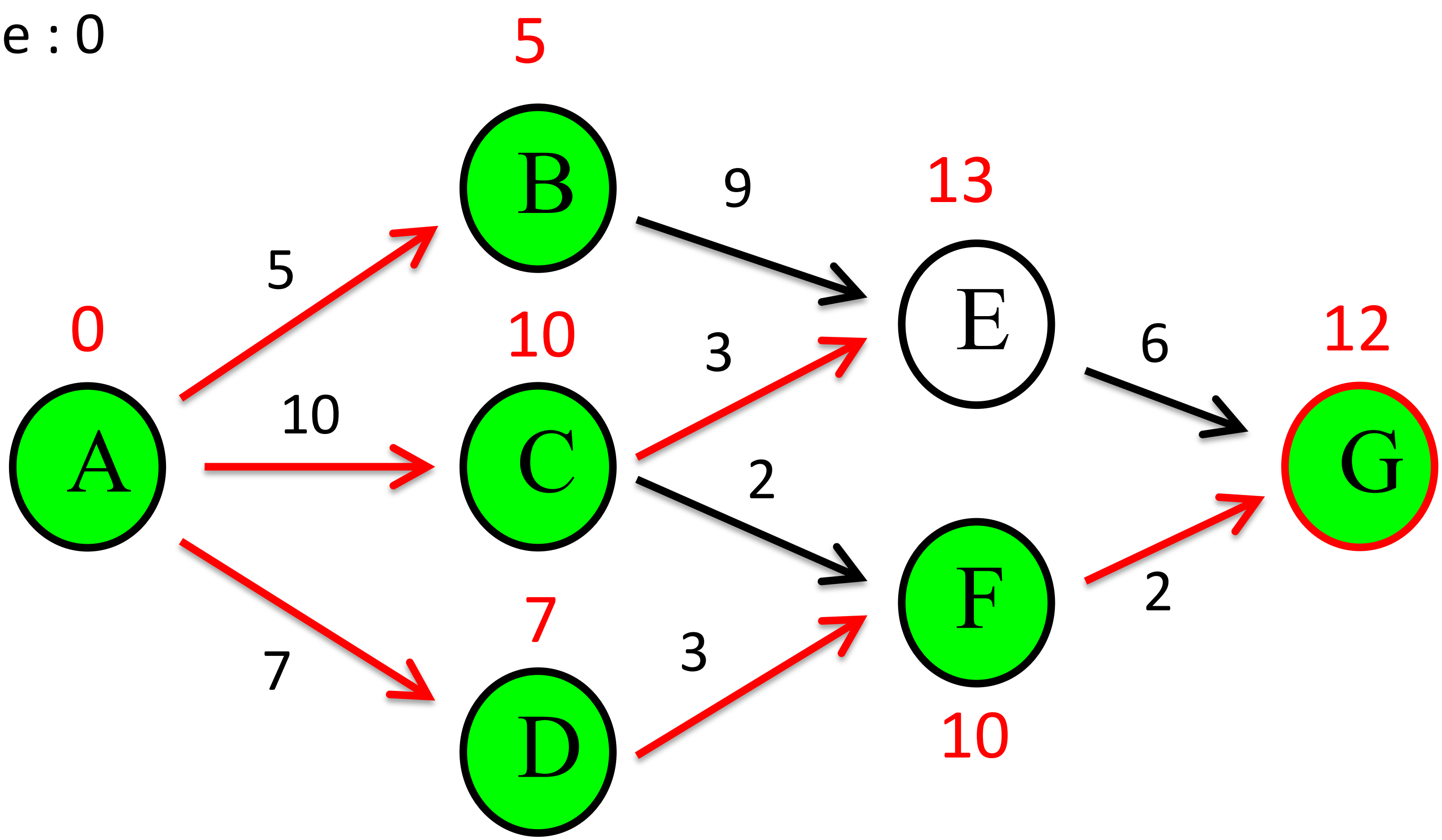
Source : 0



S : { A, B, C, D, F }

Dijkstra's Algorithm

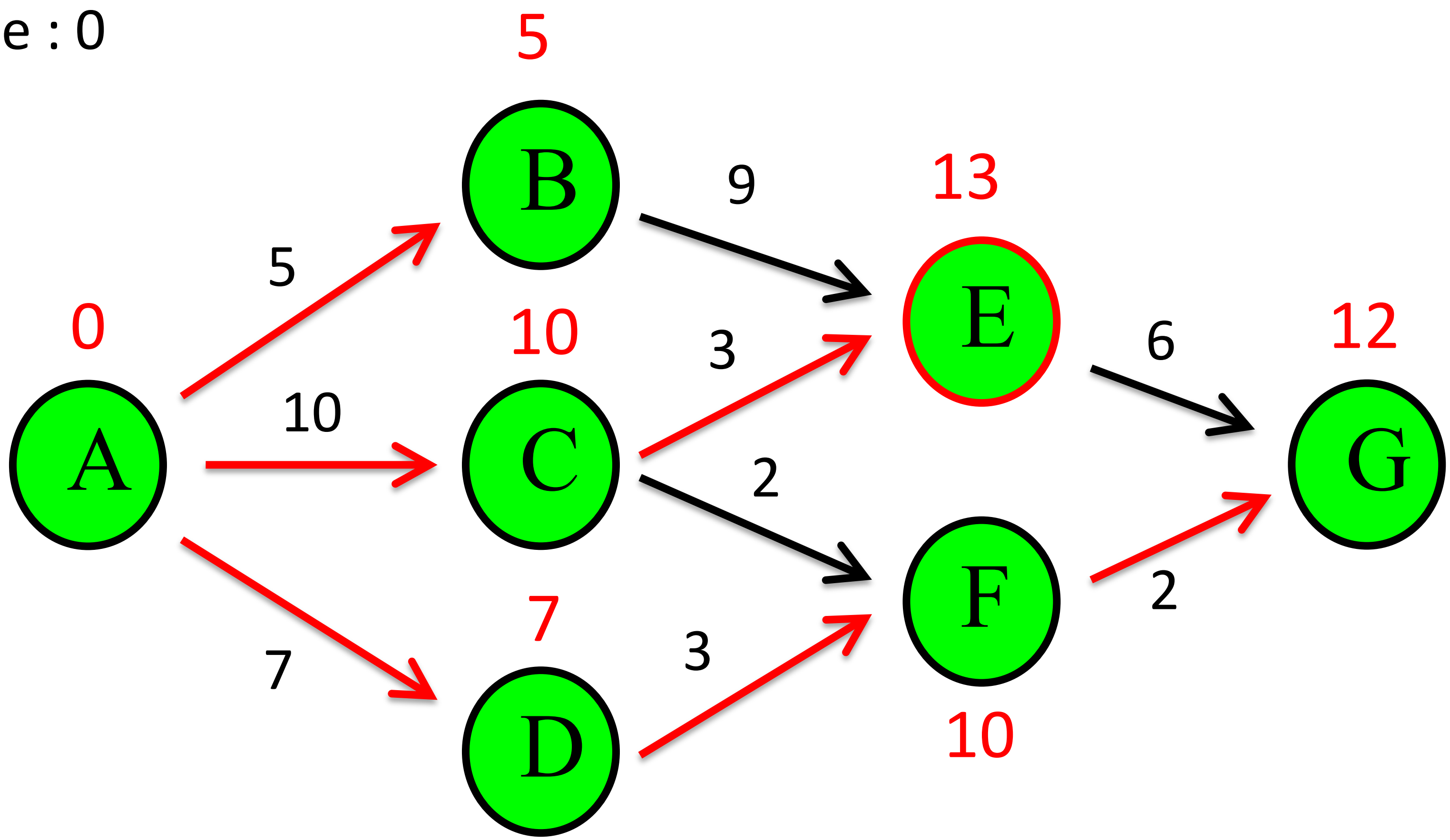
Source : 0



S : { A, B, C, D, F, G }

Dijkstra's Algorithm

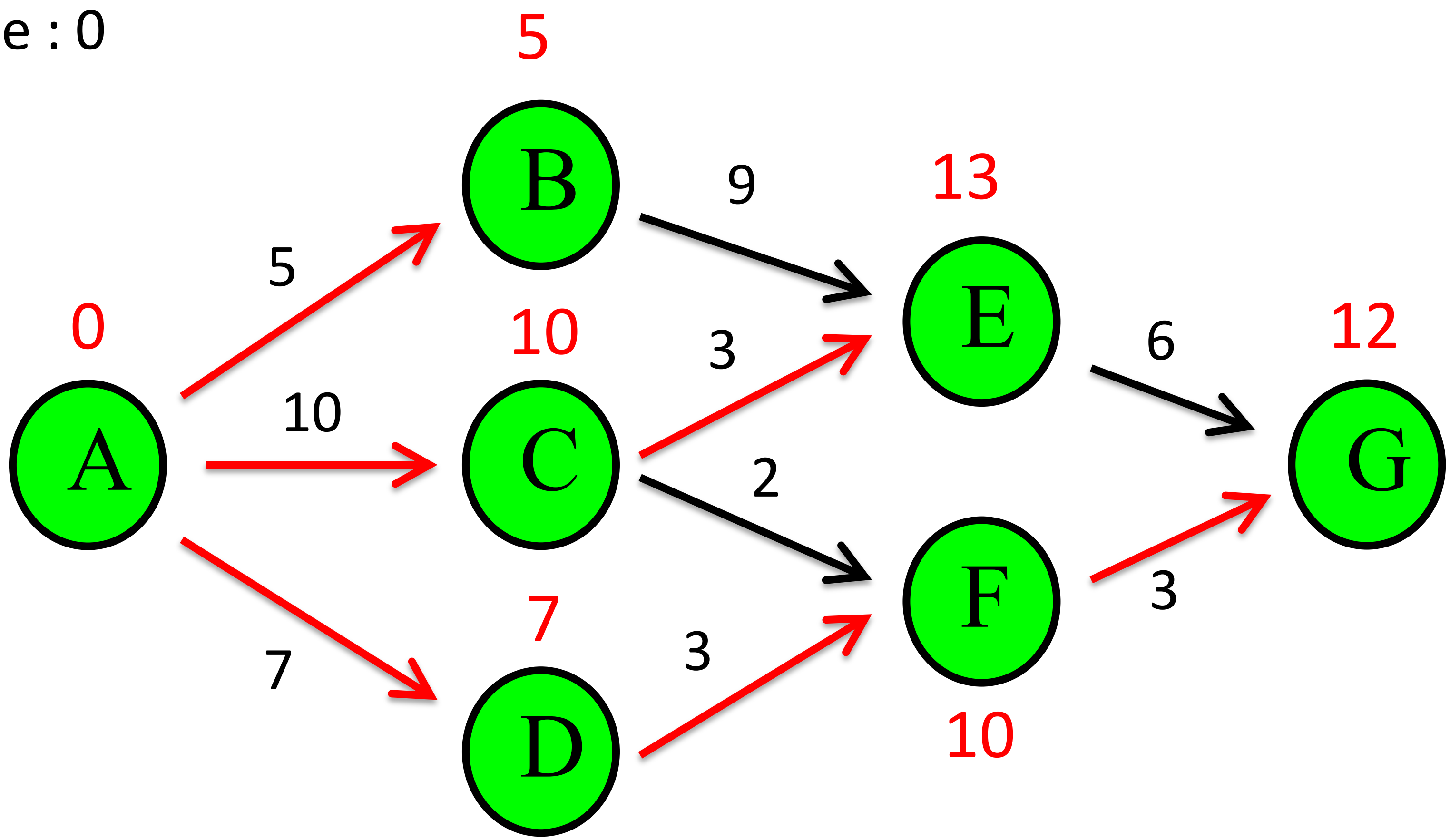
Source : 0



S : { A, B, C, D, E, F, G }

Dijkstra's Algorithm

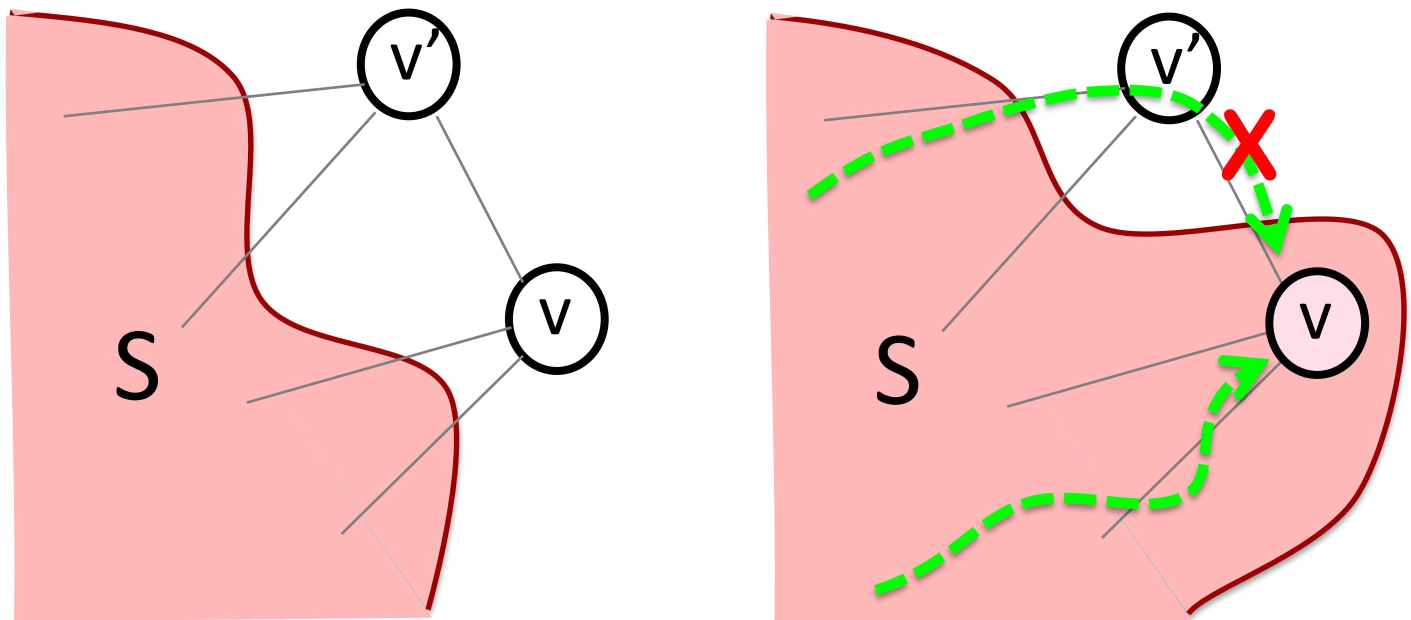
Source : 0



S : { A, B, C, D, E, F, G }

Optimality

- Once a vertex v is added to S, it is finalized—the shortest path from s to v has been found.
  - i.e., there is no v' $\notin$ S such that the path s $\rightarrow$ v' $\rightarrow$ v is shorter than the path s $\rightarrow$ v using the vertices in S only.
  - Proof by contradiction: If such v' exists, D[v] > D[v'], which violates the fact that D[v] $\leq$ D[v'] when v is added to S



Green arrow: shortest path to node

Algorithm : Linear Search

```
temp = {}, S = {}
for all vertices v
   d(v) = inf
d(source) = 0
Put all vertices to temp
while temp is not empty    : n
   v = d(v) is min in temp : n
   add v to S
   for all neighbor u of v   : # neighbor
      if d(u) > d(v) + length(v,u)
         d(u) = d(v) + length(v,u)
```

$O(n^2)$ for linear search

Algorithm : Min Heap

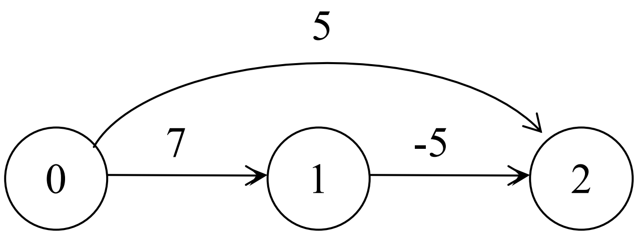
```
temp = {}, S = {}
for all vertices v
   d(v) = inf
d(source) = 0
Put all vertices to temp
while temp is not empty    : n
   v = d(v) is min in temp : log n
   add v to S
   for all neighbor u of v    : # neighbor
      if d(u) > d(v) + length(v,u)
         d(u) = d(v) + length(v,u) : log n
```

O((n+e)log(n)) for min heap

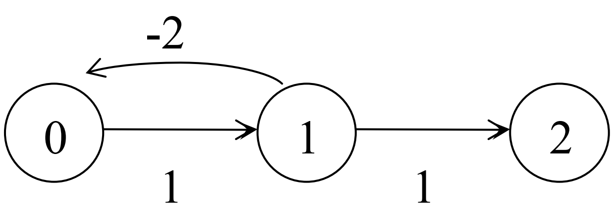
O(n log(n) +e) for Fibonacci heap

How to Handle General Weights

- Dijkstra does not work for negative weights



- No shortest path exists for a graph with cycles of negative length
  - We do not allow it



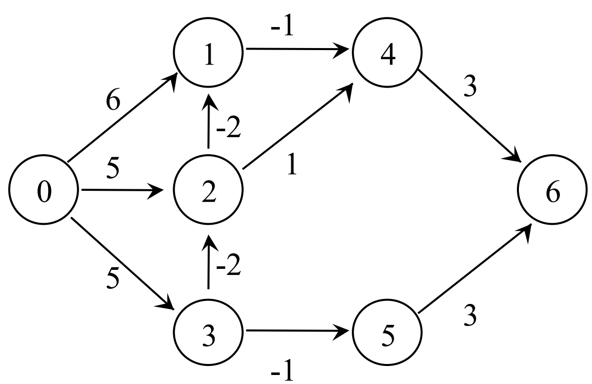
Bellman-Ford Algorithm

- Shortest path between two vertices of an n-vertex graph
  - At most n-1 edges if there are no negative length cycles
- $dist^{n-1}[u]$
  - Shortest path from source to u having at most n-1 edges

Bellman-Ford Algorithm

- Algorithm
  - Find $dist^{n-1}[u]$ for *all* u in the graph
  - Update rule from k=1 to n-1
    - $dist^k[u] = \min\{dist^{k-1}[u], \min_i\{dist^{k-1}[i] + length[i][u]\}\}$

i : all adjacent incoming vertex of u



(a) directed graph

k	Dist^k[i] 0	1	2	3	4	5	6
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

(b) $dist^k$

Bellman-Ford Algorithm

```
void Graph::BellmanFord(const int n, const int v)
{
    // distance initialization (distance for k=1)
    for(int i=0; i<n; i++) dist[i] = length[v][i];

O(n) for(int k=2; k<=n-1; k++)
    {
O(n2) / O(e)  for(each u s.t u!=v and u has at least one incoming edge)
        {
            for(each <i,u> in the graph)
            {
                if(dist[u]>dist[i]+length[i][u])
                    dist[u] = dist[i] + length[i][u];
            }
        }
    }
}
```

O(?)

Questions?