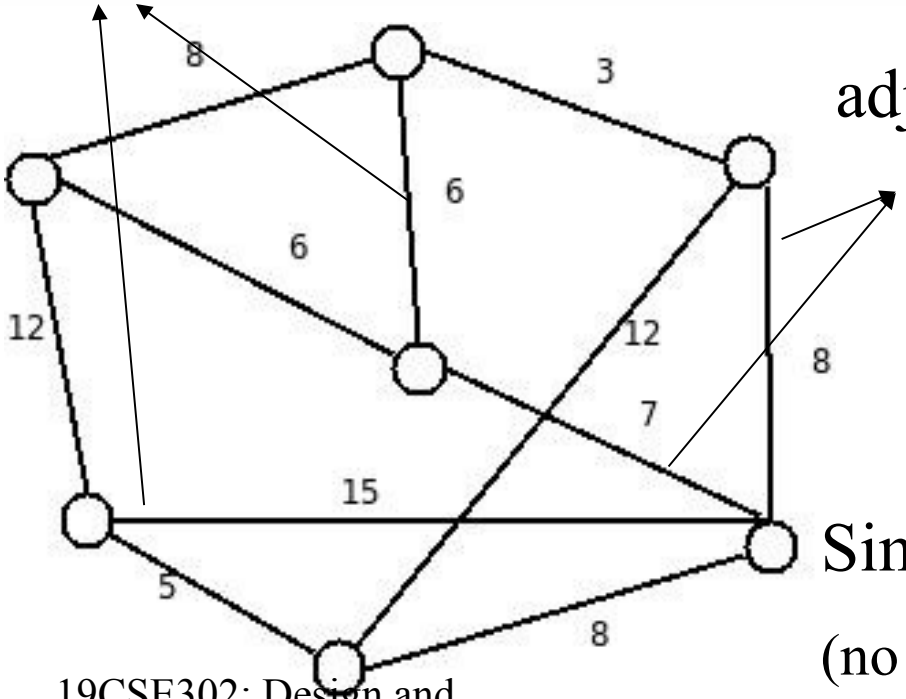
Graph Algorithms

Graph Definitions

- A graph G= (V,E) is a set of vertices V, and a collection of edges E which is a subset of VxV
  - Directed, undirected or mixed

independent

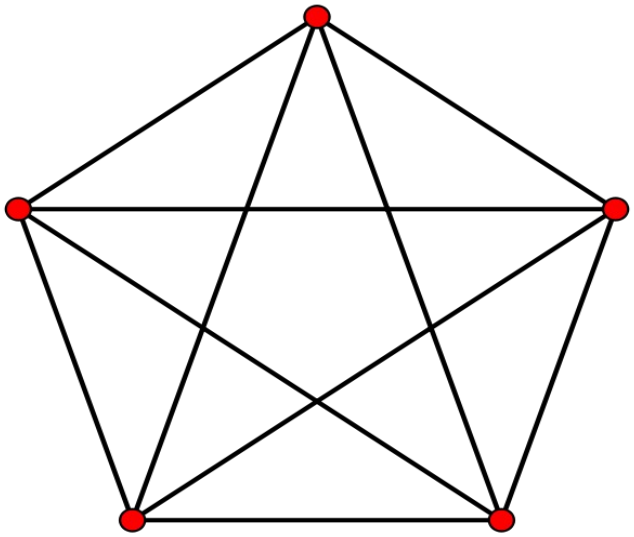


adjacent

Simple Graph

(no loops, parallel edges)

Complete Graph



Degree

- Number of edges incident at v (d(v))
  - In-degree – number of incoming edges to vertex v
  - Out-degree – number of outgoing edges
- An *isolated* vertex has degree 0
- Min degree of G --> min {d(v)| v \in V}
- Total degree of G is 2m (number of edges)
- To remember
  - Number of vertices of odd degree is always even in a graph

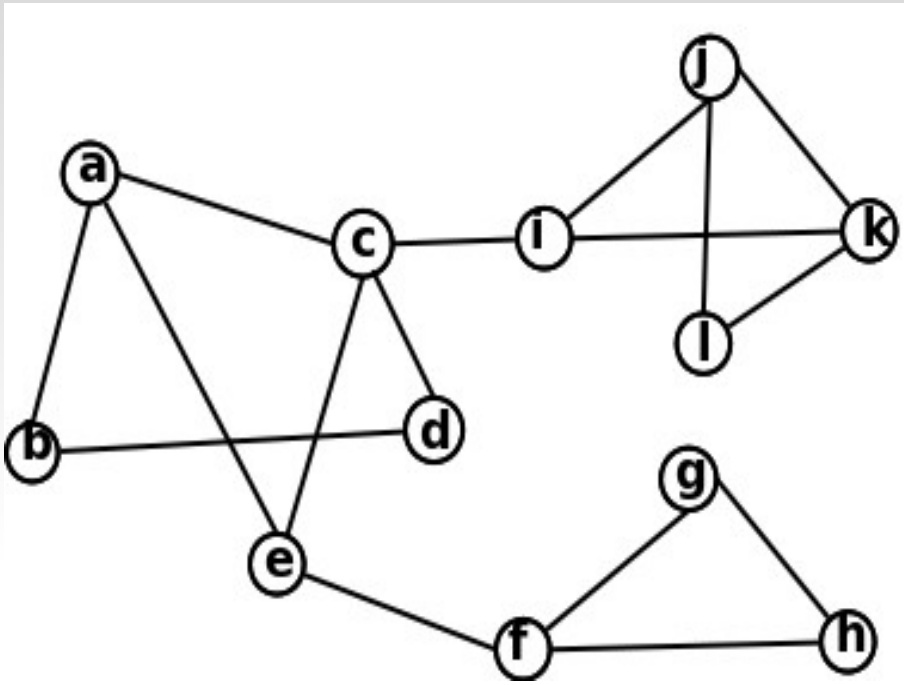
Graph Representations

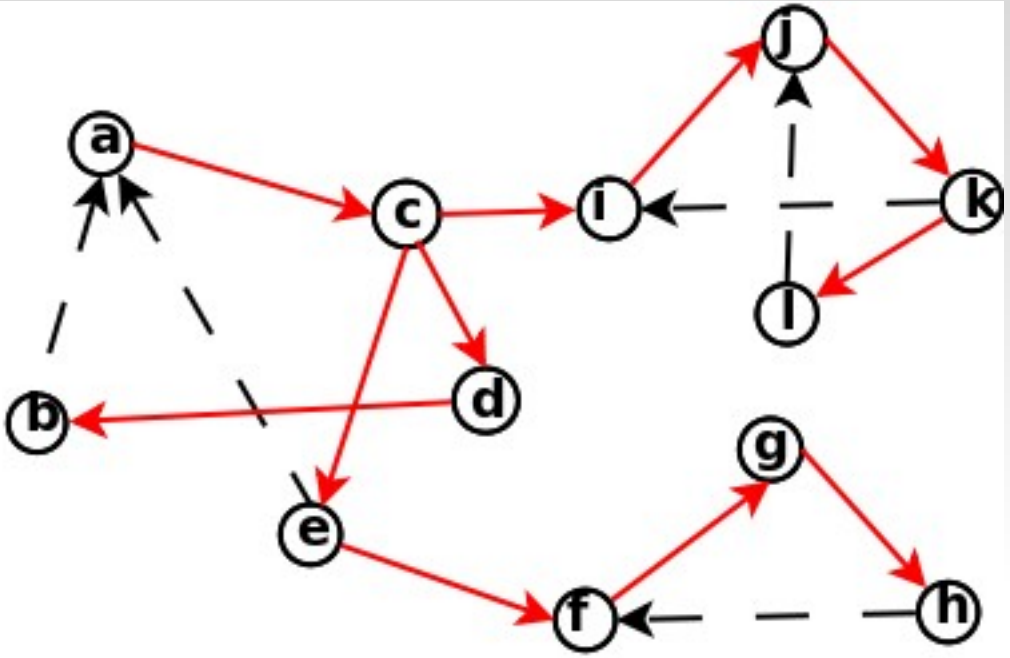
- Adjacency list
  - Each vertex has incidence container
    - List of vertices incident on v
  - Edge list
  - Useful in path algorithms or if graph is sparse
- Adjacency matrix
  - Edge list
  - Matrix A where A[i,j] represents edge
  - Edge retrieval quick
- Path – list of vertices connective u and v

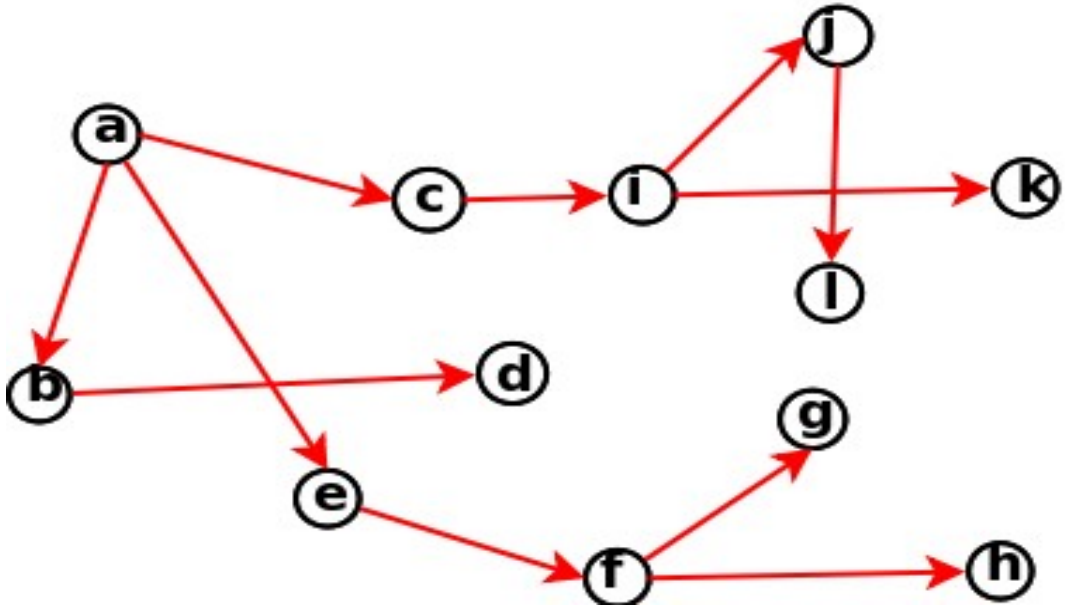
Graph Traversal

- Depth first search
  - Recursive – O(m+n) algorithm
  - Start with some node v
    - Of all neighbors of v, goto next w which is unexplored do DFS(w)
    - If w explored, then mark edge as back edge
- Breadth first search
  - Discovery in levels, marks new nodes in levels
  - O(m+n)
- Applications
  - Checking connectivity, cycle detection, finding biconnected components

Example







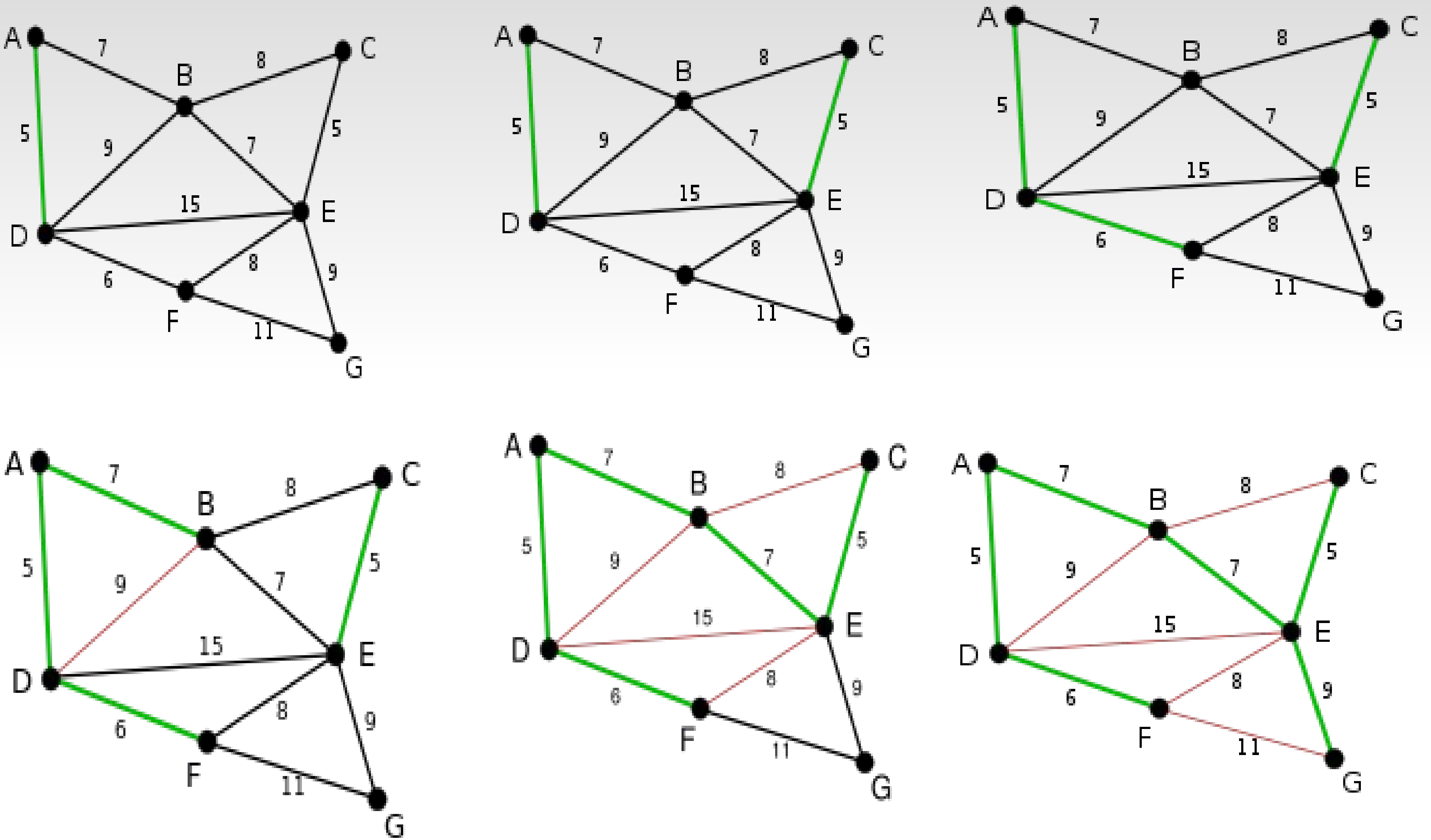
Minimum Spanning Tree

- Given a weighted undirected graph G, goal is to find a T such that
  - T contains all vertices in G
  - Sum of weights of edges in T is minimum
- Different algorithms
  - Prim's
  - Kruskal
  - Boruvka's
- All use some greedy strategy

Kruskal's Algorithm

- Let every node in G be a cluster C(v)
- Initialize a priority queue Q with all edges in G using weights as keys
- Take the minimum weight edge in Q and if C(u) <> C(v)
  - add the edge to MST T
  - Merge the clusters C(u) and C(v)
- Repeat till there are no more clusters to merge

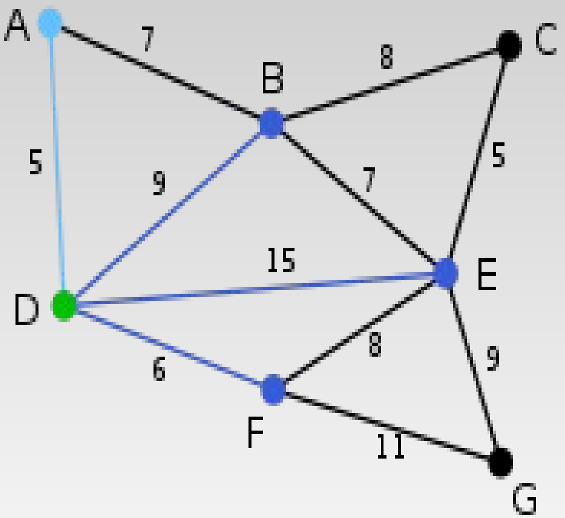
Kruskal's Algorithm: Example

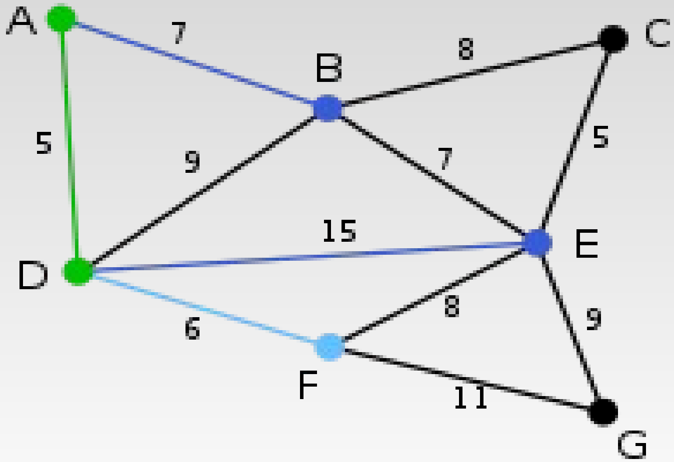


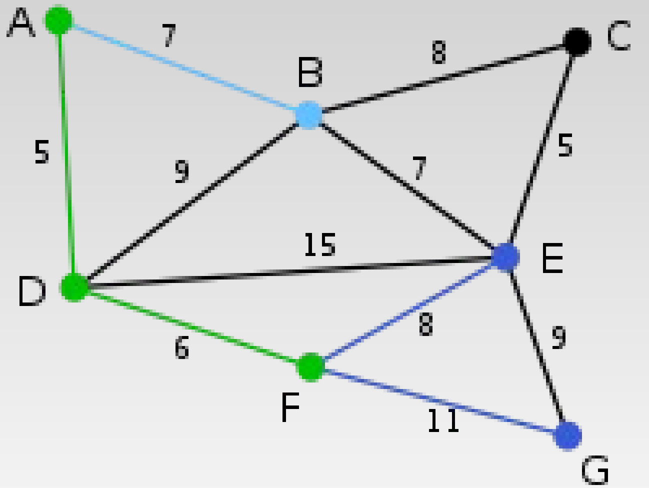
Prim's Algorithm

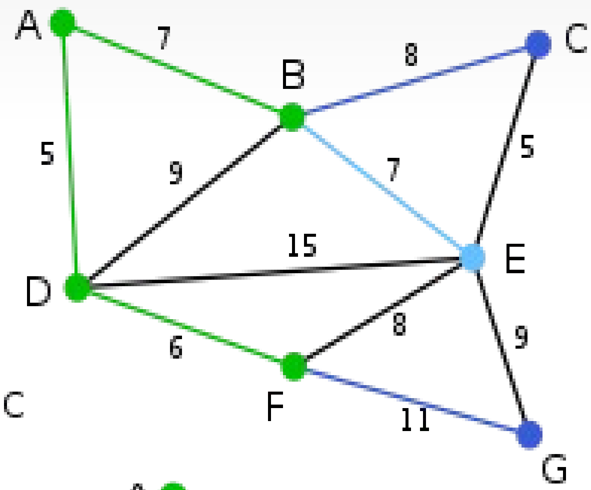
- Similar to Dijkstra
- Pick any vertex v of G
- Initialize for all u not v, d[u] to infinity and d[v]=0
- Remove from Q u with minimum d[u]
  - Add u and edge (u,v) to T
  - For all neighbors z of u, do relaxation by finding d[z] = w(u,z), and update Q

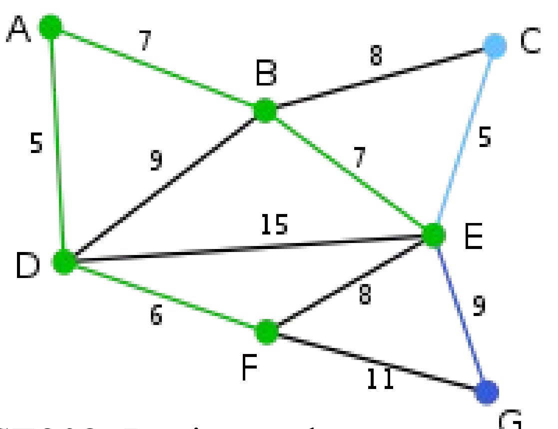
Example

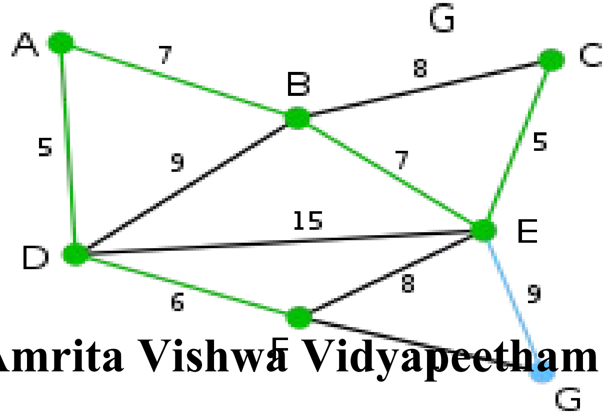

A B C D E F G
7 8 5 9 7 5 15 6 8 9 11

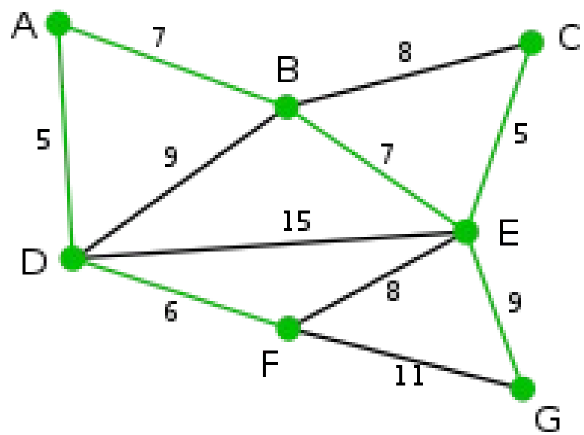

A B C D E F G
7 8 5 9 7 5 15 6 8 9 11


A B C D E F G
7 8 5 9 7 5 15 6 8 9 11


A B C D E F G
7 8 5 9 7 5 15 6 8 9 11


A B C D E F G
7 8 5 9 7 5 15 6 8 9 11

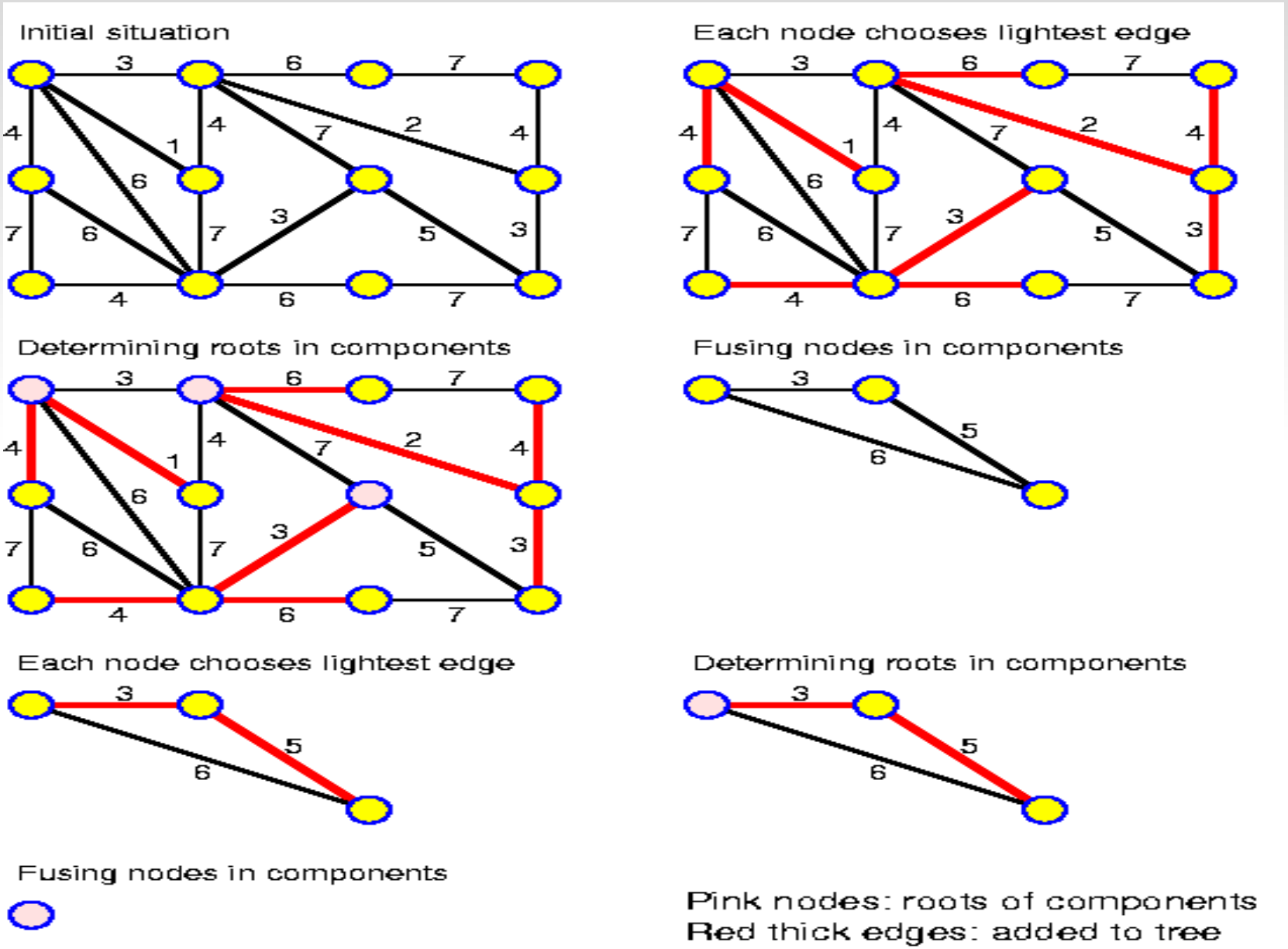

A B C D E F G
7 8 5 9 7 5 15 6 8 9


A B C D E F G
7 8 5 9 7 5 15 6 8 9 11

Baruvka's Algorithm

- Here greedy strategy explicitly optimizes certain priorities of vertices
- Initially every vertex is a component C_i
- Find the smallest weight edge (v,u) in E and add to C_i such that
  - v is in C_i, u is not in C_i
  - Add e to T

Example

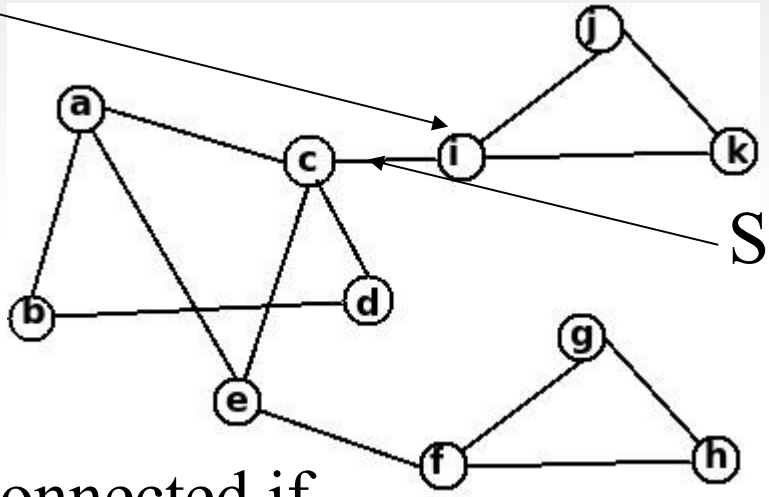


http://www8.cs.umu.se/~jopsi/dinf504/boruvka.gif

Vidhya Balasubramanian

Biconnected Components

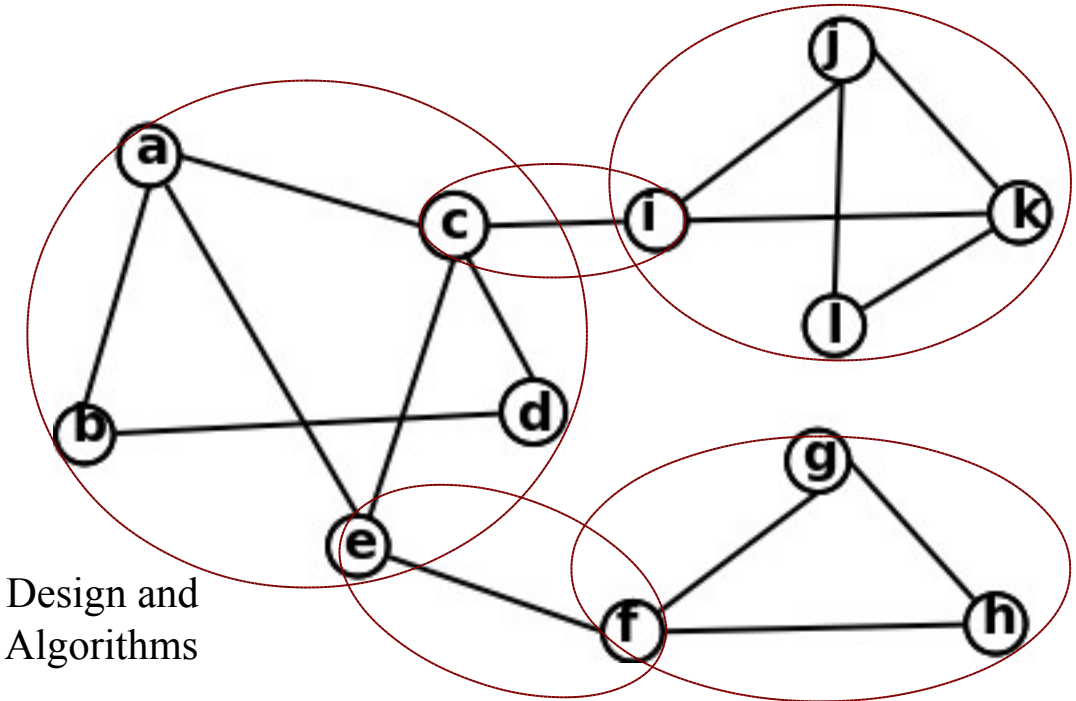
- Separation or cut edge - Edge whose removal disconnects a graph
- Separation vertex – vertex removal disconnects graph



- Graph/component biconnected if
  - There are no separation edge or vertices
  - For any two vertices u, v, there are 2 disjoint paths between them

Biconnected components

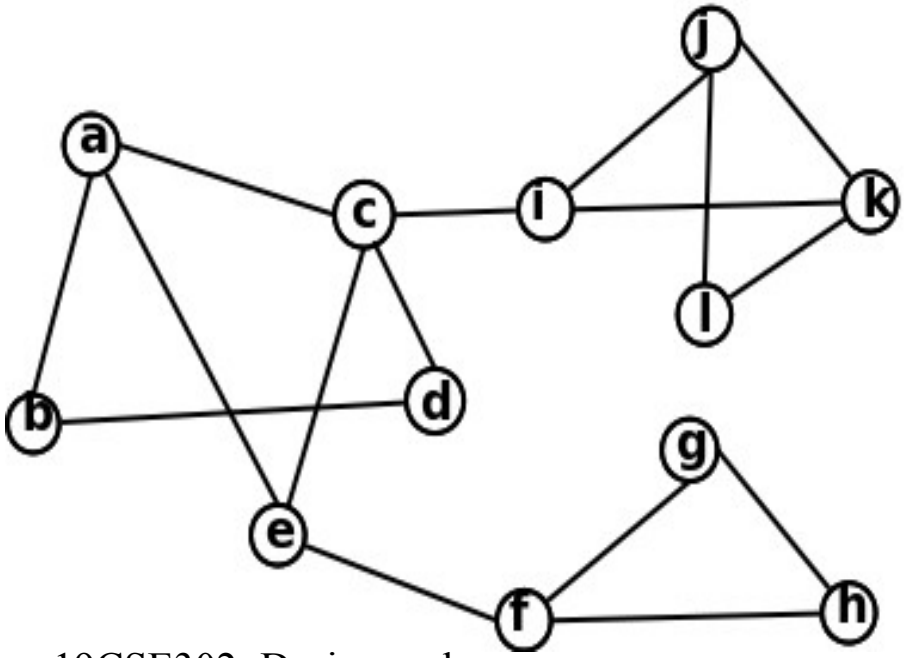
- Lemma : Let G be a biconnected graph. The following are equivalent
  - G is biconnected
  - For any two vertices of G, there is a simple cycle containing them
  - G does not have any separation vertices or separation edges

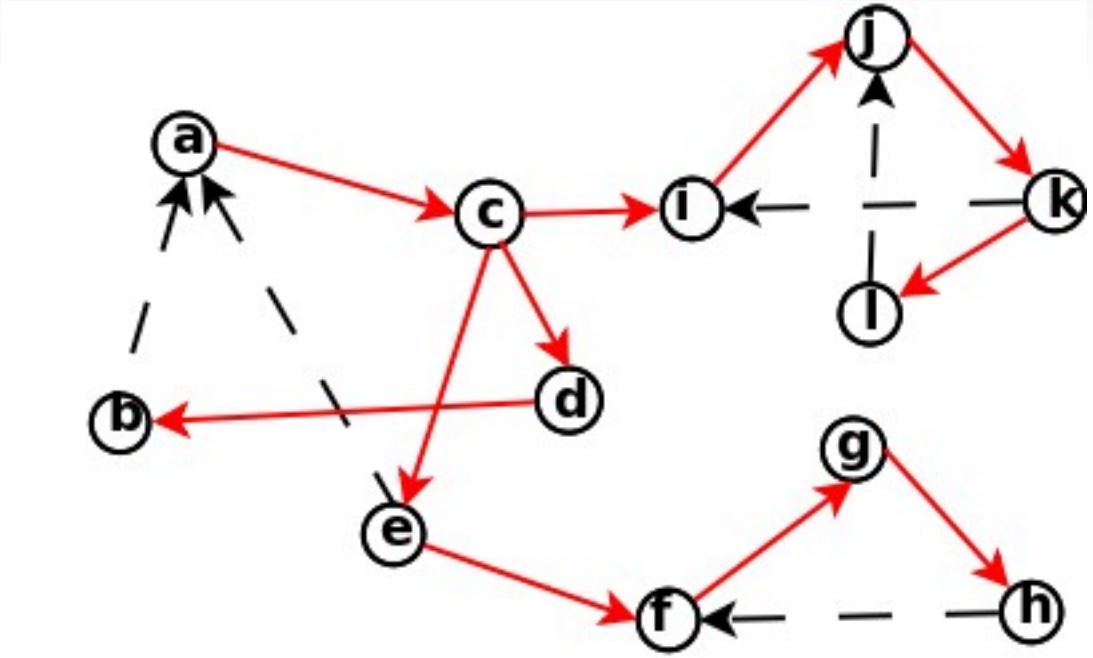


Compute Biconnected Components via DFS

- To construct biconnected components of G we need to compute the equivalence classes of the link relation among G's edges
- Steps

  1. Do a DFS traversal of G





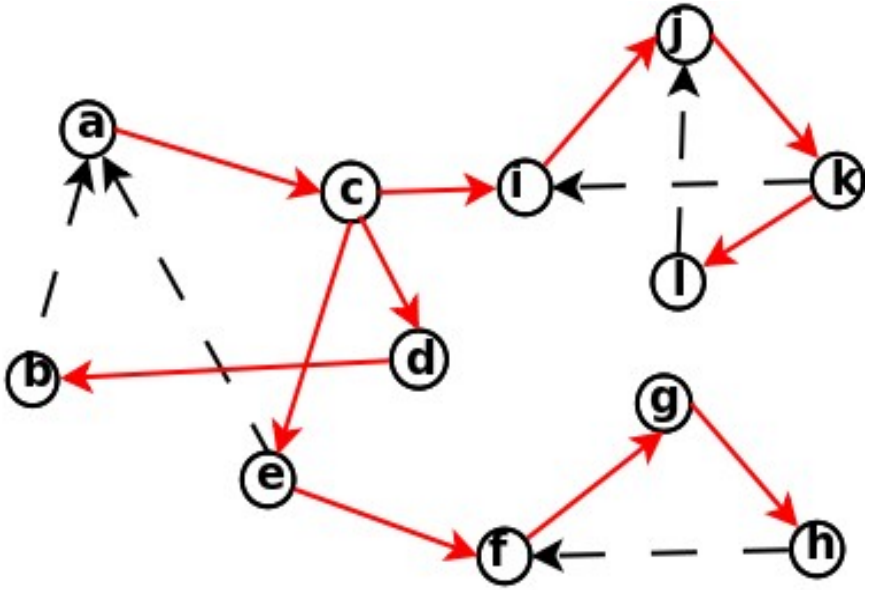
Auxiliary Graph

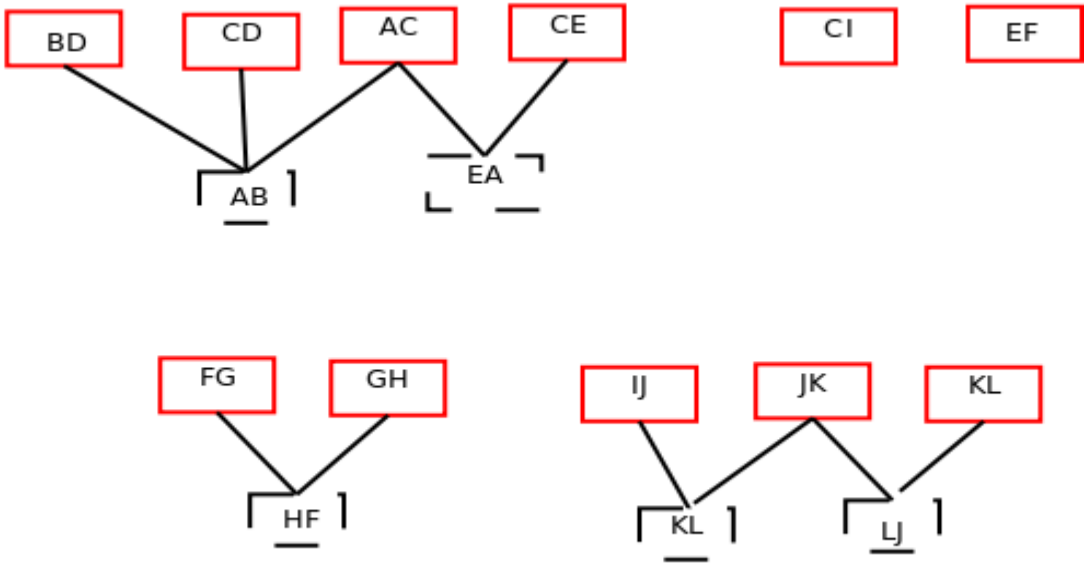
- Construct an auxiliary graph B as follows

  The vertices of B are edges of G

  For every back edge e of G, let f_1, ...f_k be the discovery edges of G that form a cycle with e

  Graph B contains edges (e,f_1), (e,f_2),...,(e,f_k)



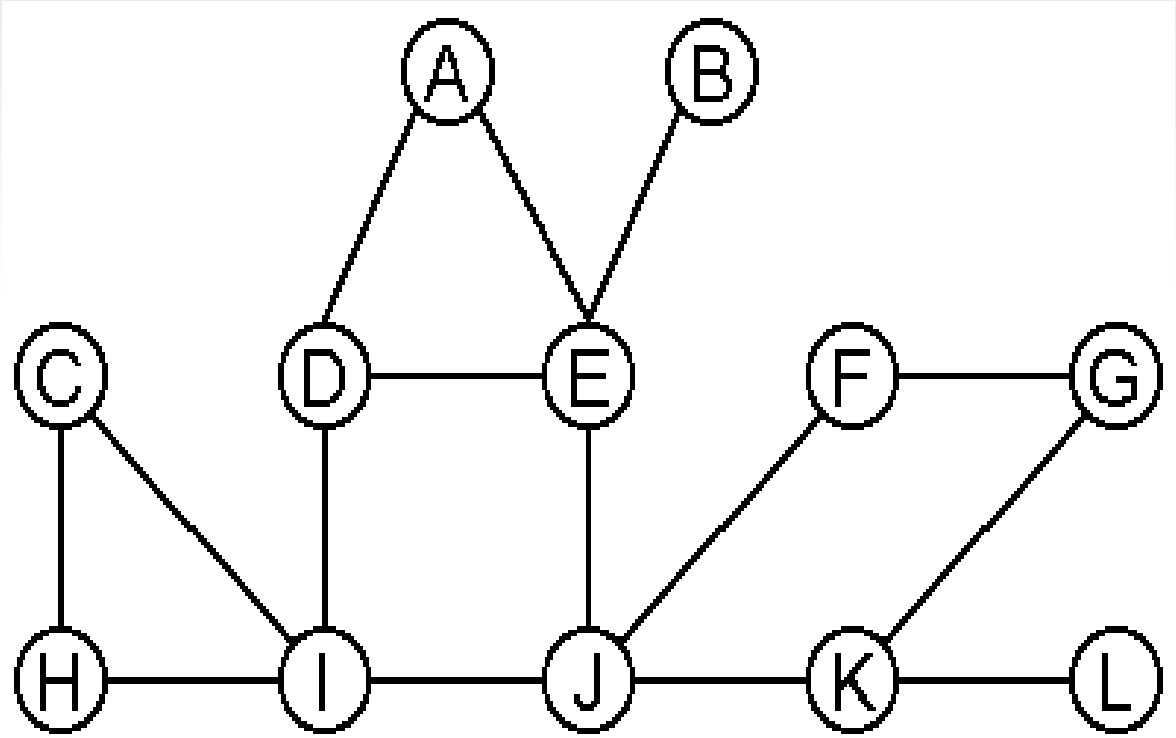


Finding the components

- The connected components of the auxiliary graph B correspond to link components of the graph G that induced B
  - Compute the connected components of B for example by performing a DFS traversal of the auxiliary graph B
  - For each connected component of B , output the vertices of B as a link component of G
    - Vertices of B are edges of G
- Complexity
  - Finding separation edges and vertices – O(m+n)
  - Finding auxiliary graph O(mn)
    - Reduce this by finding a spanning forest of B

Exercise

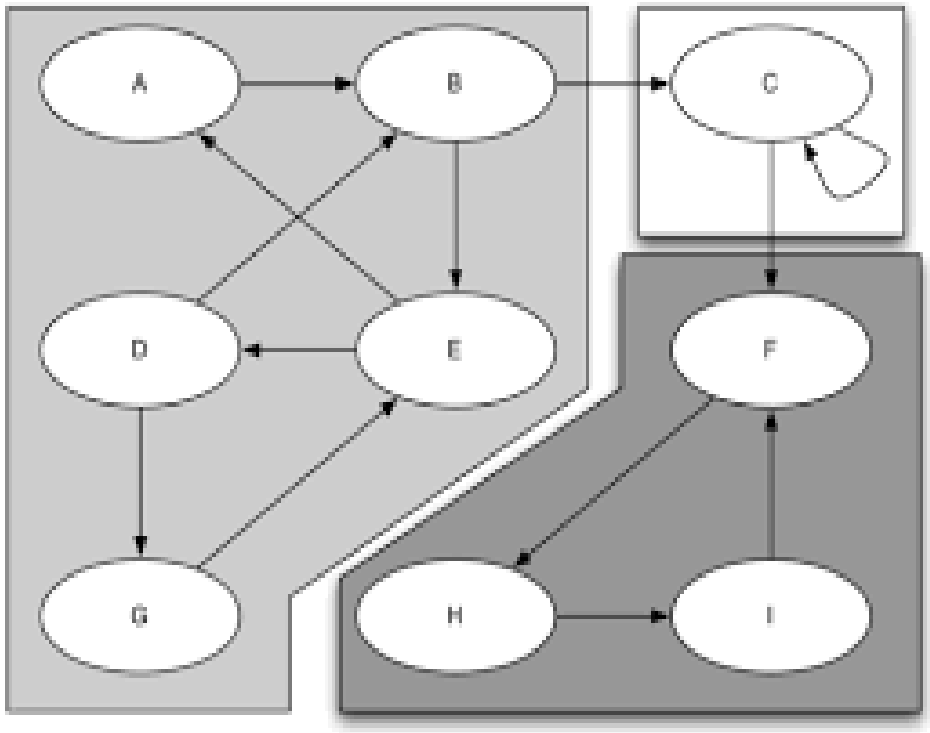
- Find all the biconnected components of this graph using DFS method



Strongly Connected Components

Two nodes a and b in a directed graph are connected if there are paths between a to b and vice versa

- A directed graph is strongly connected if there are directed paths between every pair of nodes
- Directed graph is DAG of its strongly connected components



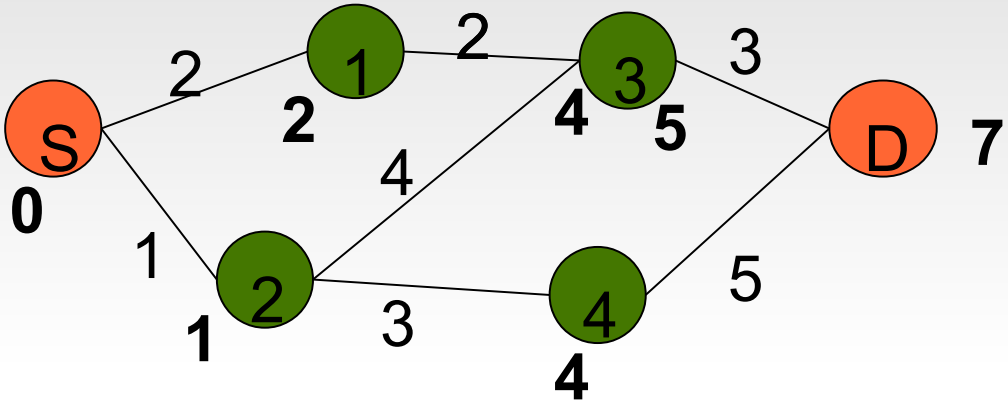
Use DFS to find the components

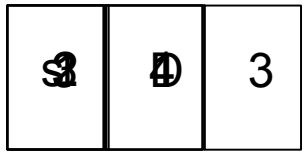
Apply DFS on graph and then on transpose graph G'

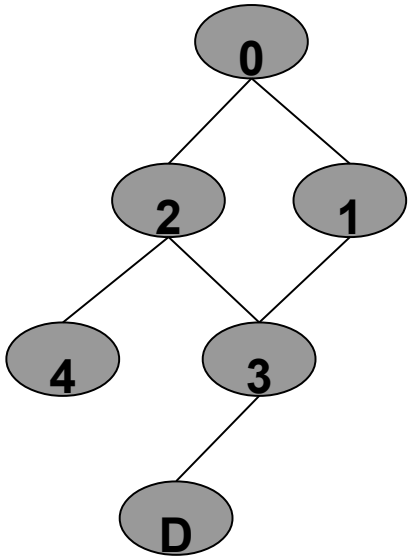
G' has same set of vertices with all of the edges reversed compared to the orientation of the corresponding edges in G

Dijkstra's Algorithm

- Dijkstra O((n+m) logn)
  - Uses greedy approach
- Edge relaxation
  - d(z) ← min{d(z),d(u) + weight(e)}
- Priority Queue
  - Closest to source inserted first





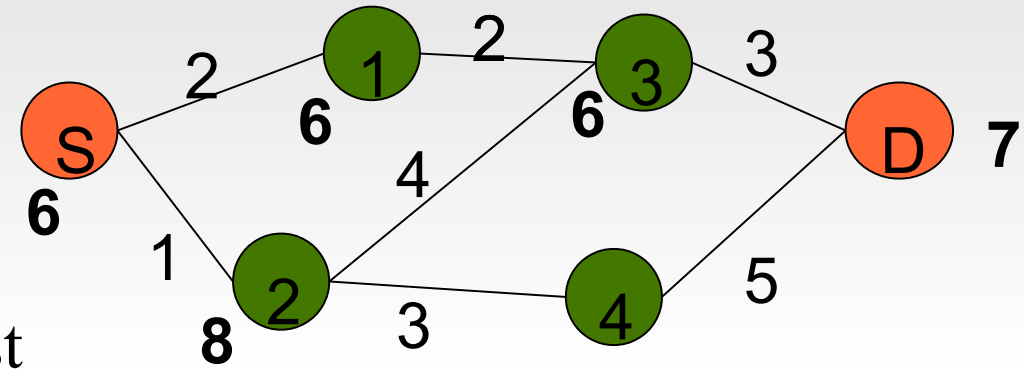


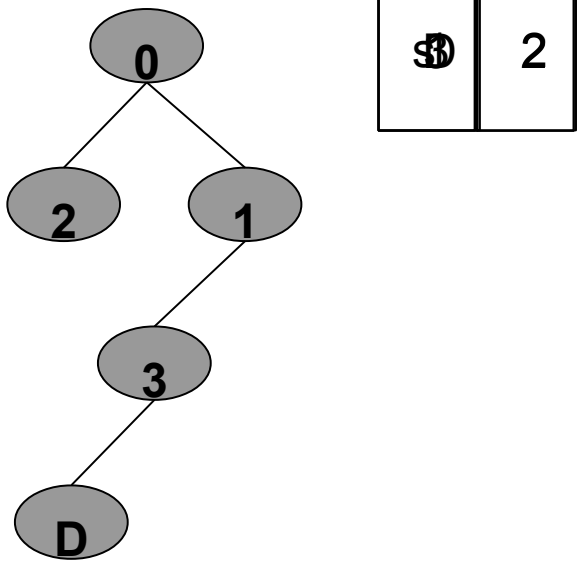
Dijkstra's Algorithm

- Initially d[src] =0, and infinite for other nodes
- At each iteration the node u with least d is chosen from the priority queue
  - For each neighbor z of u do edge relaxation
    - $d(z) \leftarrow \min\{d(z), d(u) + \text{weight}(e)\}$
    - Insert z into the priority queue if it is already not there
- Terminate when priority queue empty

A* Algorithm

- A* goal driven
  - Similar to Dijkstra 's but uses estimated distance to goal
  - Cost to source + heuristic cost to destination
- Heuristic
  - Euclidean distance to destination
- Priority Queue
  - Total cost





A* Algorithm

- A* goal driven
  - Used a lot in games and intelligent transport systems
  - Similar to Dijkstra 's but uses estimated distance to goal
- Cost Metric
  - f[u]=d[u]+h[u]
  - d[u] is the cost of the least-cost path to src node
  - h[u] is an estimate distance from u to the destination
    - Usually the straight line distance
  - f[u] is used in the priority queue

A* Algorithm

Procedure: AStarPath(v_{src}, v_{dest})

```
(1)  S_done ← ∅ //set of processed nodes
(2)  d[v_src] ← 0 // Least cost distance from source
(3)  Q ← v_src
(4)  while NotEmpty(Q)do
(5)     u ← Q.gettop
(6)     if u = v_dest then
(7)        return ReconstructPath(v_scr, v_dest)
(8)        S_done ← S_done ∪ u
(9)     for each v ∈ GetNeighbors(u)
(10)       if v ∈ S_done then
(11)          continue
(12)       if d[u]+ LinkCost(u, v) < d[v]
(13)          d[v]← d[u]+ LinkCost(u, v)
(14)          f[v]← d[v]+ HeuristicDist(v, d)
(15)          PREV(v)← u
(16)          if v ∈ Q
(17)             UpdateCost(v, d[v], f[v])
(18)          else INSERT(v,Q)
(19)  return FAILURE
```

Dijkstra vs A*

- Search Area
- A* optimality
  - Admissible heuristic

Bellman Ford

- A dynamic programming approach to finding shortest paths
  - First calculates the shortest distances which have at-most one edge in the path.
  - Next calculates shortest paths with atmost 2 edges, and so on.
  - After the i-th iteration, the shortest paths with at most i edges are calculated.

Bellman Ford - Basics

- detects negative cycles
- For every node
  - Edge relax all edges in the graph
  - O(mn)
- If edge relaxation continues
  - Negative cycle present

Bellman Ford

- ***BELMAN-FORD***(G, s)

  INIT(G, s) // set node wts of all except s to ∞

  for i ←1 to |V|-1 do

    for each edge (*u*, *v*) ∈ E do

      RELAX(u, v)

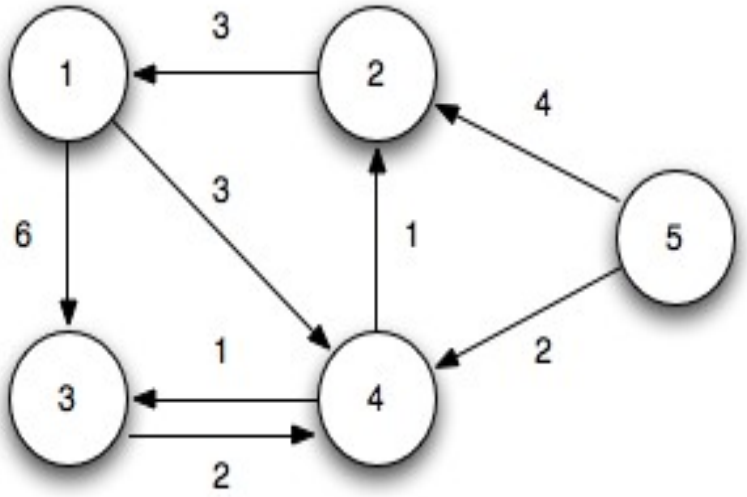
  for each edge (u, v) ∈ E do

    if d[v] > d[u]+w(u,v) then

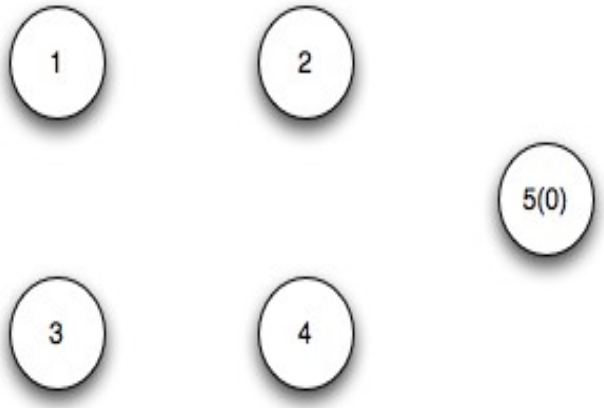
      return *FALSE* > neg-weight cycle

  return *TRUE*

Example



(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

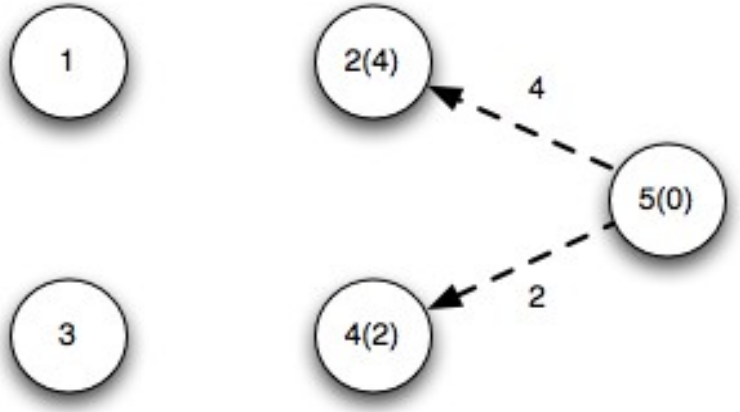


(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	∞	∞	∞	∞	**0**
π	/	/	/	/	/

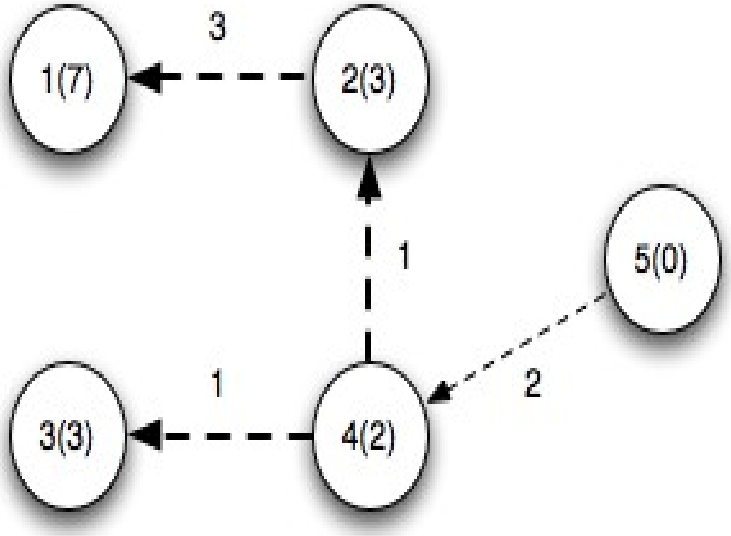
Src:http://faculty.ycp.edu/~dbabcock/PastCourses/cs360/lecture/lecture21.html

Example



(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	∞	**4**	∞	**2**	0
π	/	**5**	/	**5**	/

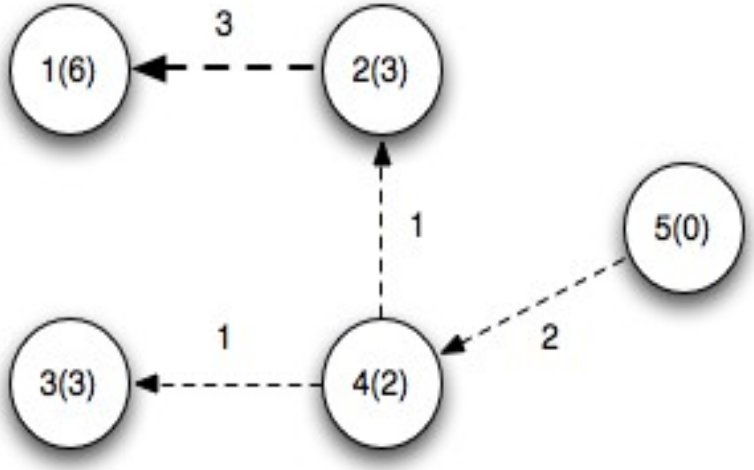


(1,3) = 6
(1,4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	**7**	**3**	**3**	2	0
π	**2**	**4**	**4**	5	/

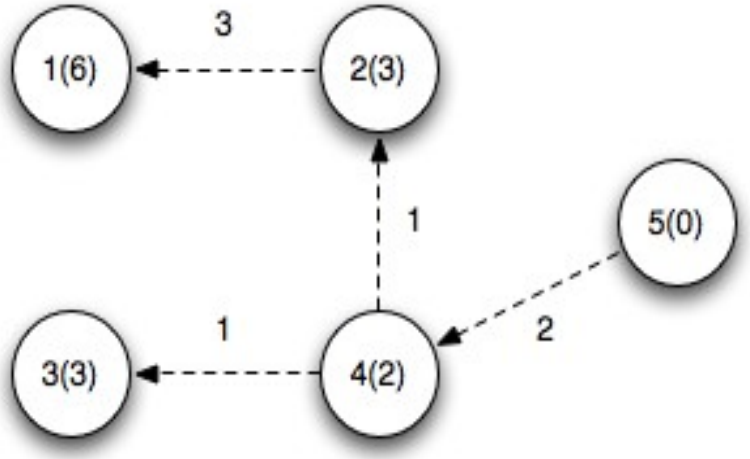
Src:http://faculty.ycp.edu/~dbabcock/PastCourses/cs360/lecture/lecture21.html

Example



(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	**6**	3	3	2	0
π	**2**	4	4	5	/

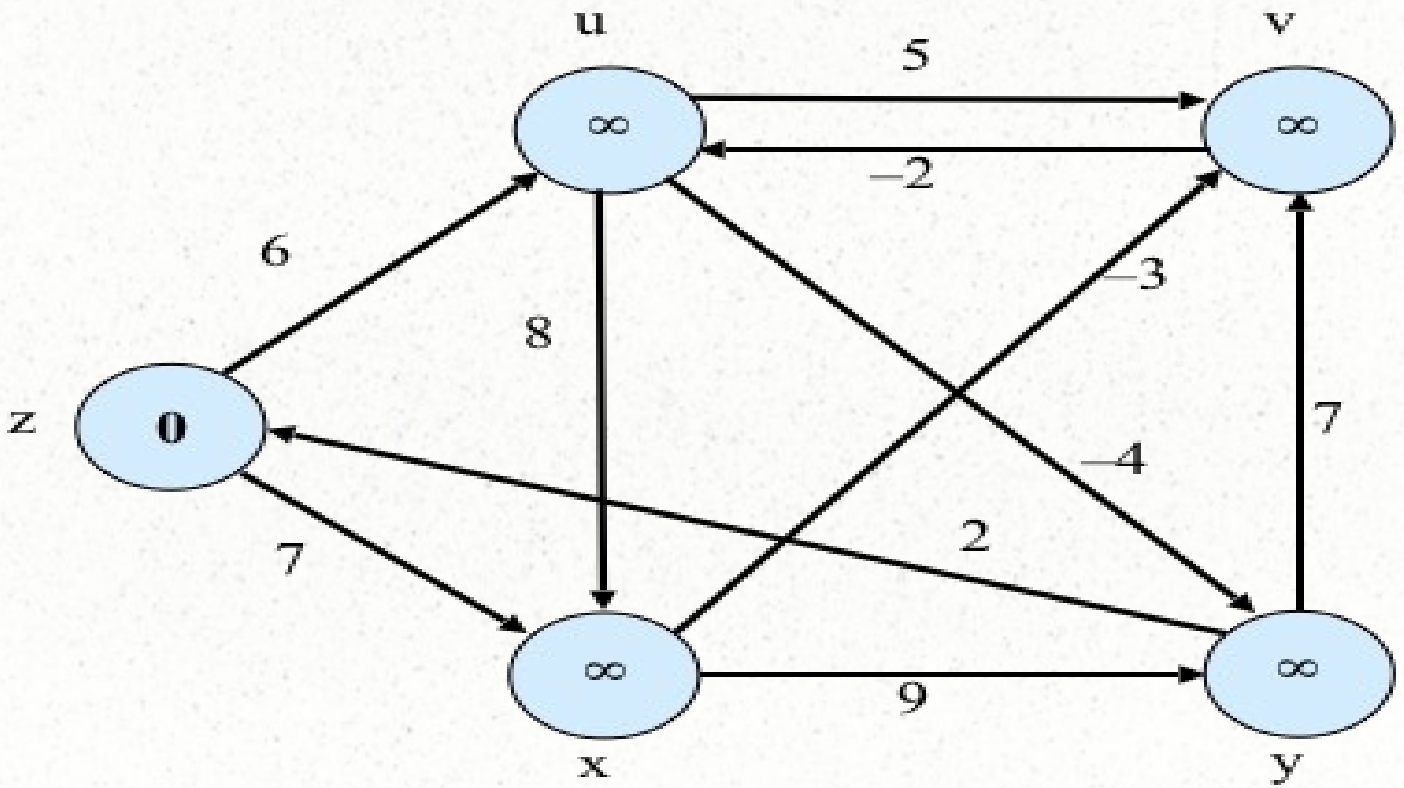


(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

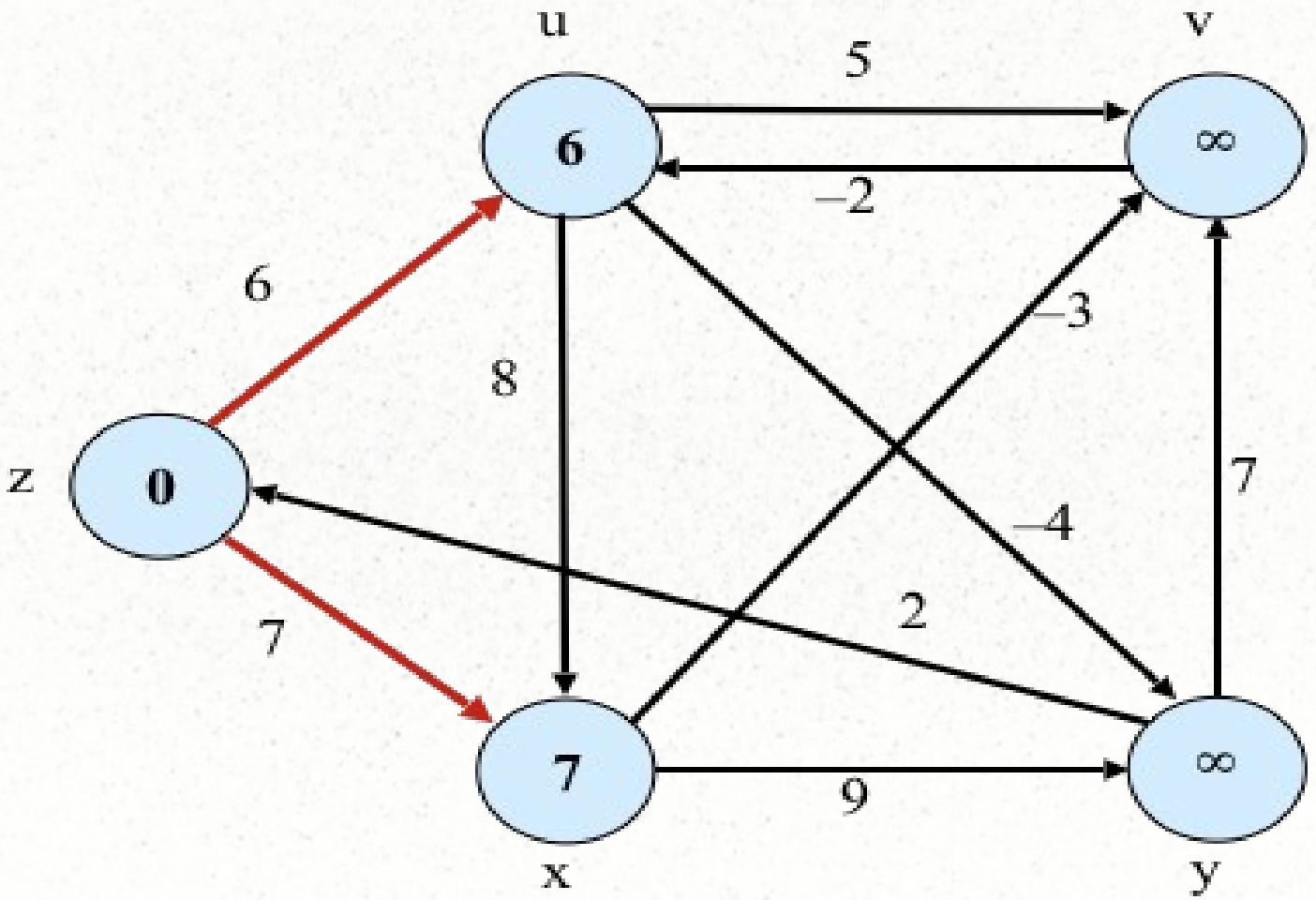
	1	2	3	4	5
d	6	3	3	2	0
π	2	4	4	5	/

Src:http://faculty.ycp.edu/~dbabcock/PastCourses/cs360/lecture/lecture21.html

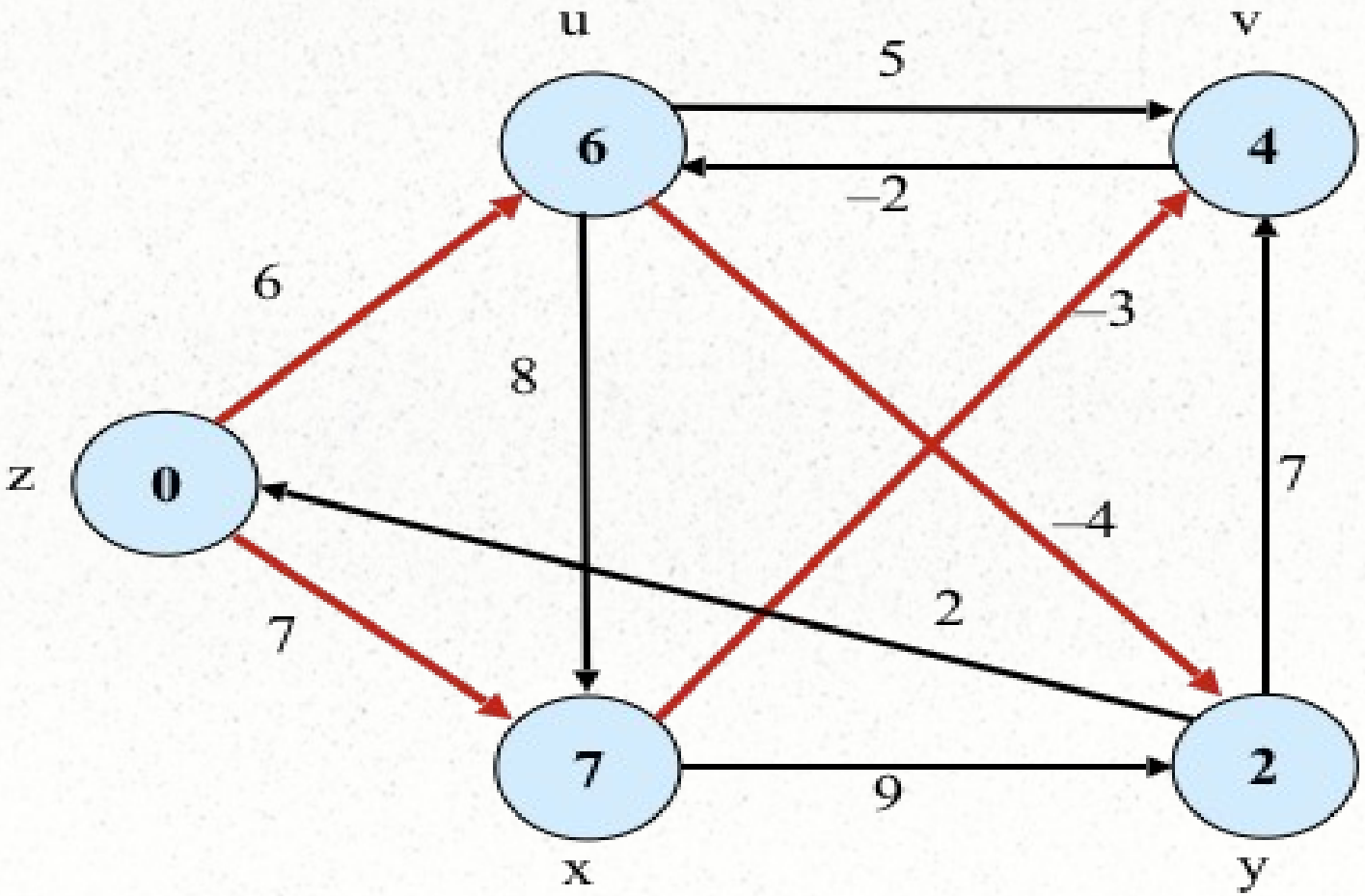
Example



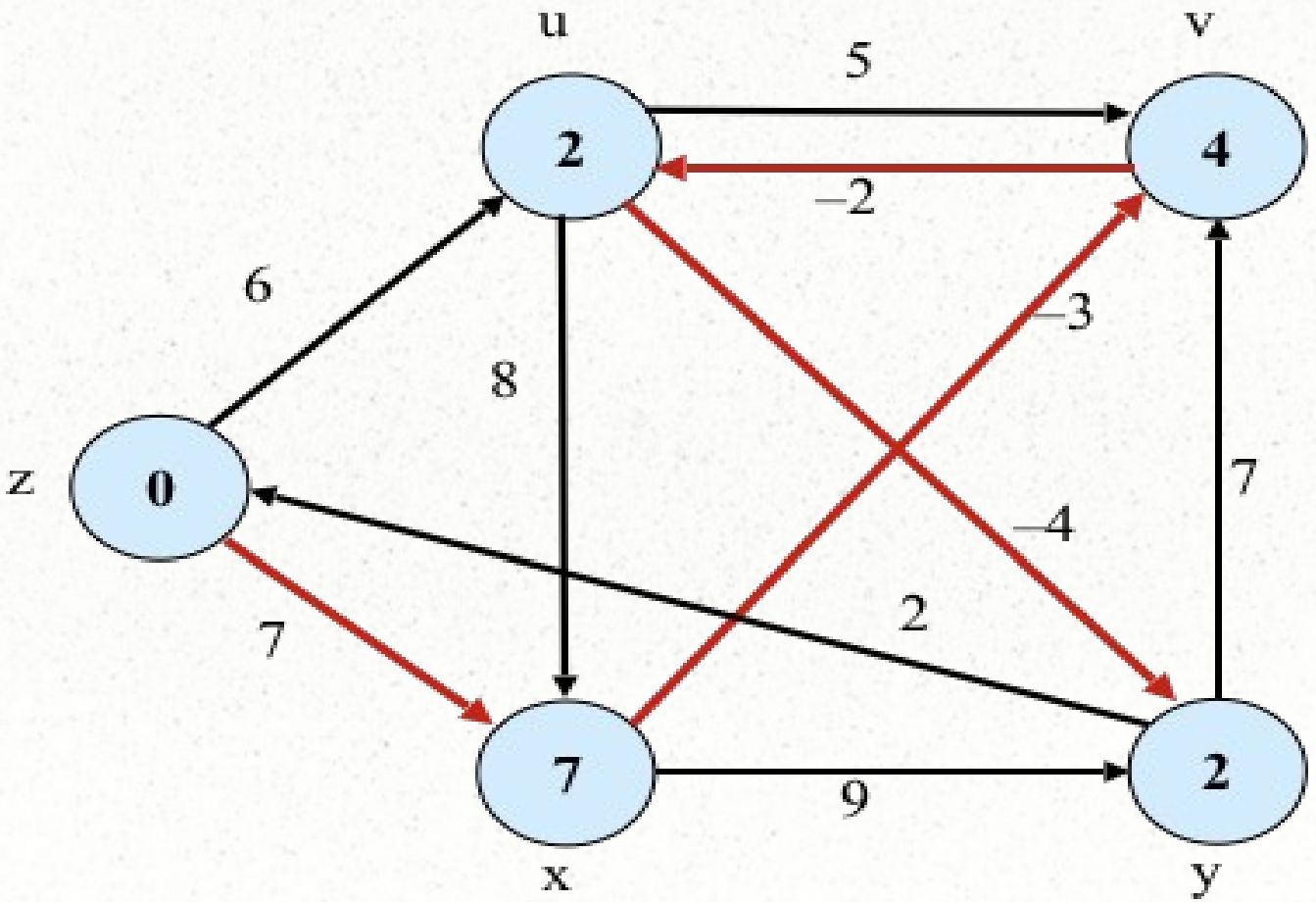
Example



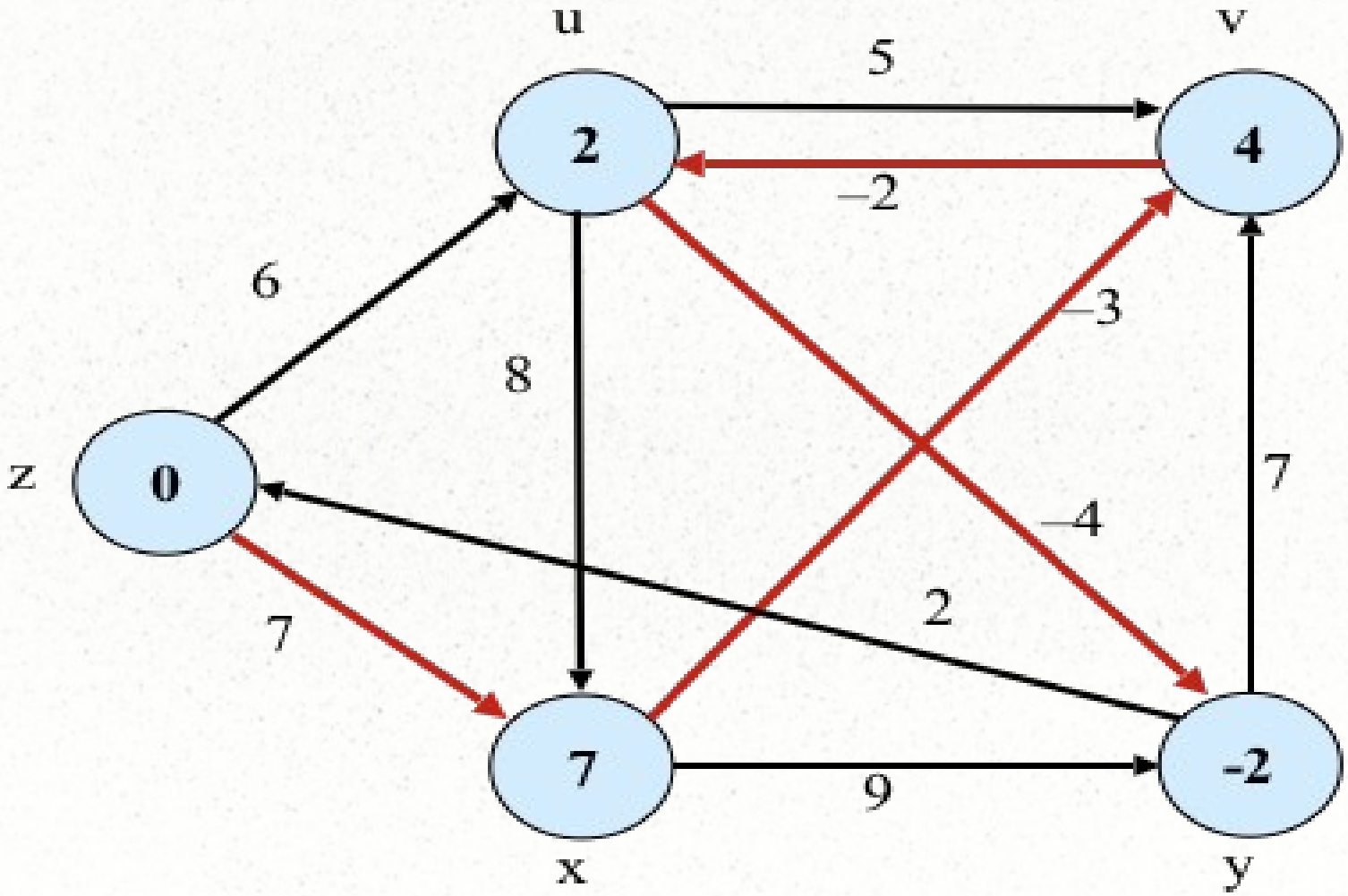
Example



Example

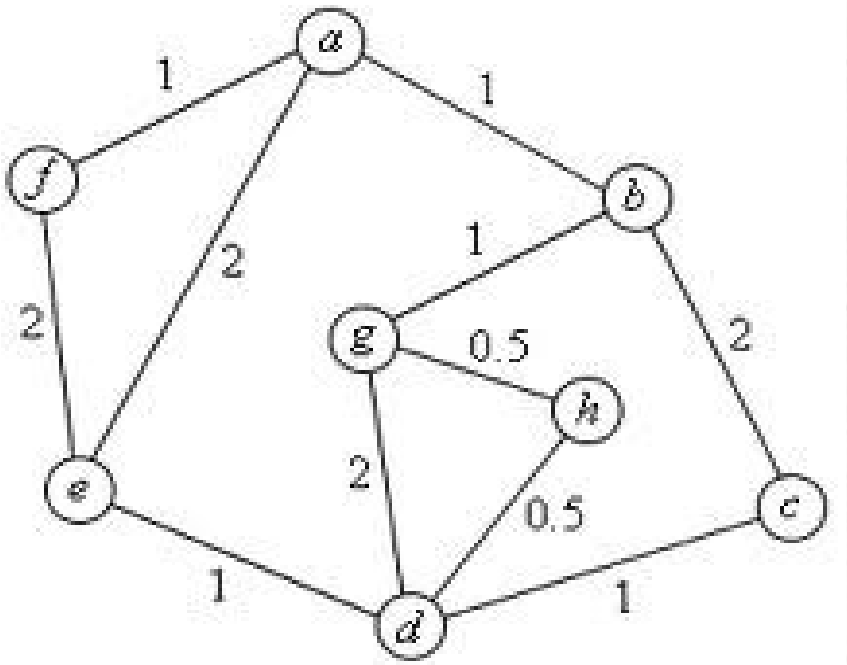


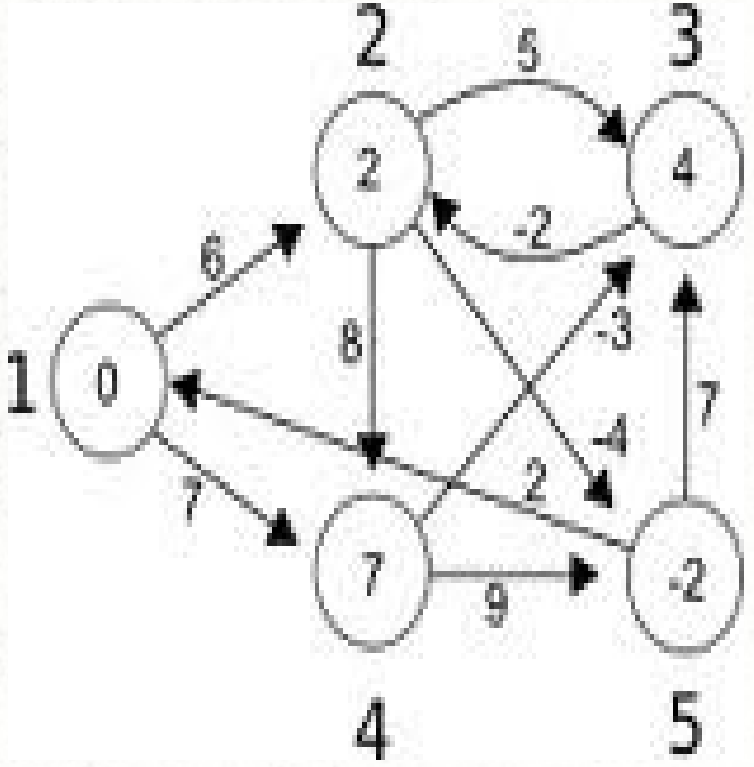
Example



Exercise

- Find shortest paths from node a, and 0





Floyd Warshall Algorithm

- Efficient for finding all-pair shortest paths in a graph
  - Graph can have negative weight edges, but not negative cycles
  - $O(n^3)$ algorithm
- Application of Dynamic Programming

Dynamic Programming Strategy

- Let *dist*(*k*,*i*,j) be the the length of the shortest path from i and j that uses only the vertices v_1, v_2, ..., v_k as intermediate vertices
  - If *k*=0, *dist*(*0*,*i*,*j*) is the length of the edge (*i*,*j*)if it exists, and infty otherwise
  - *dist*(*k*,*i*,*j*) = min(*dist*(*k* - 1,*i*,*k*) + *dist*(*k* - 1,*k*,*j*),*dist*(*k* – 1,*i*,*j*))
    - Shortest path between i and j does not involve k at all or is a combination of path between i to k and path from k to j
- *dist*(*N*,*i*,*j*) represents shortest distance b/w i and j.

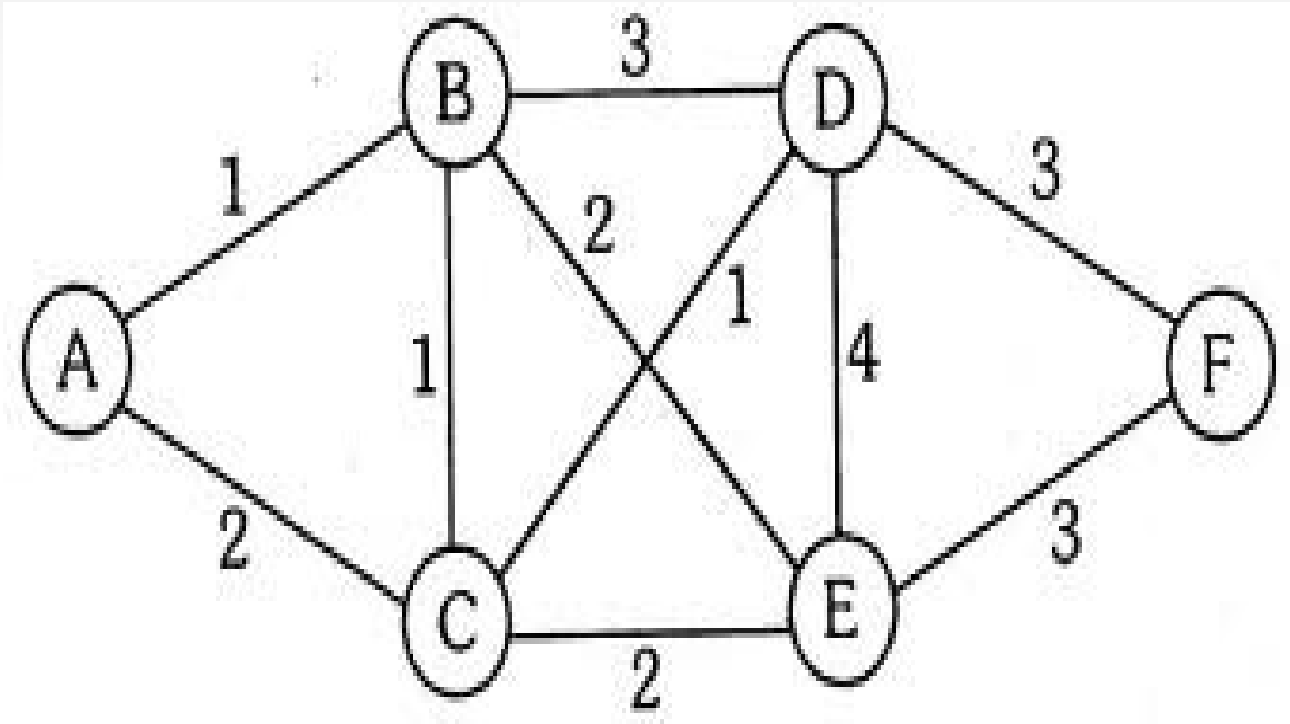
Floyd Warshall Algorithm

```
for i = 1 to N
   for j = 1 to N
      if there is an edge from i to j
         dist[0][i][j] = the length of the edge from i to j
      else
         dist[0][i][j] = INFINITY

for k = 1 to N
   for i = 1 to N
      for j = 1 to N
         dist[k][i][j] = min(dist[k-1][i][j], dist[k-1][i][k] +
    dist[k-1][k][j])
```

Exercise

- Find shortest paths between every node pair



Analysis

- Runs in O(n+m) time – n vertices and m edges
  - Initial computation of indegree O(n+m)
- Uses O(n) auxiliary space (stack)
- If some vertices are not numbered then there is a cycle
  - Any vertex on a directed cycle will not be visited
  - A vertex visited only when incounter is 0
    - Implies all its previous predecessors were previously visited