Chapter 9 Graph - part 2

South China University of Technology

College of Software Engineering

Huang Min

Topological Sort

Topological Sort

Problem: Given a set of jobs, courses, etc., with prerequisite constraints, output the jobs in an order that does not violate any of the prerequisites.

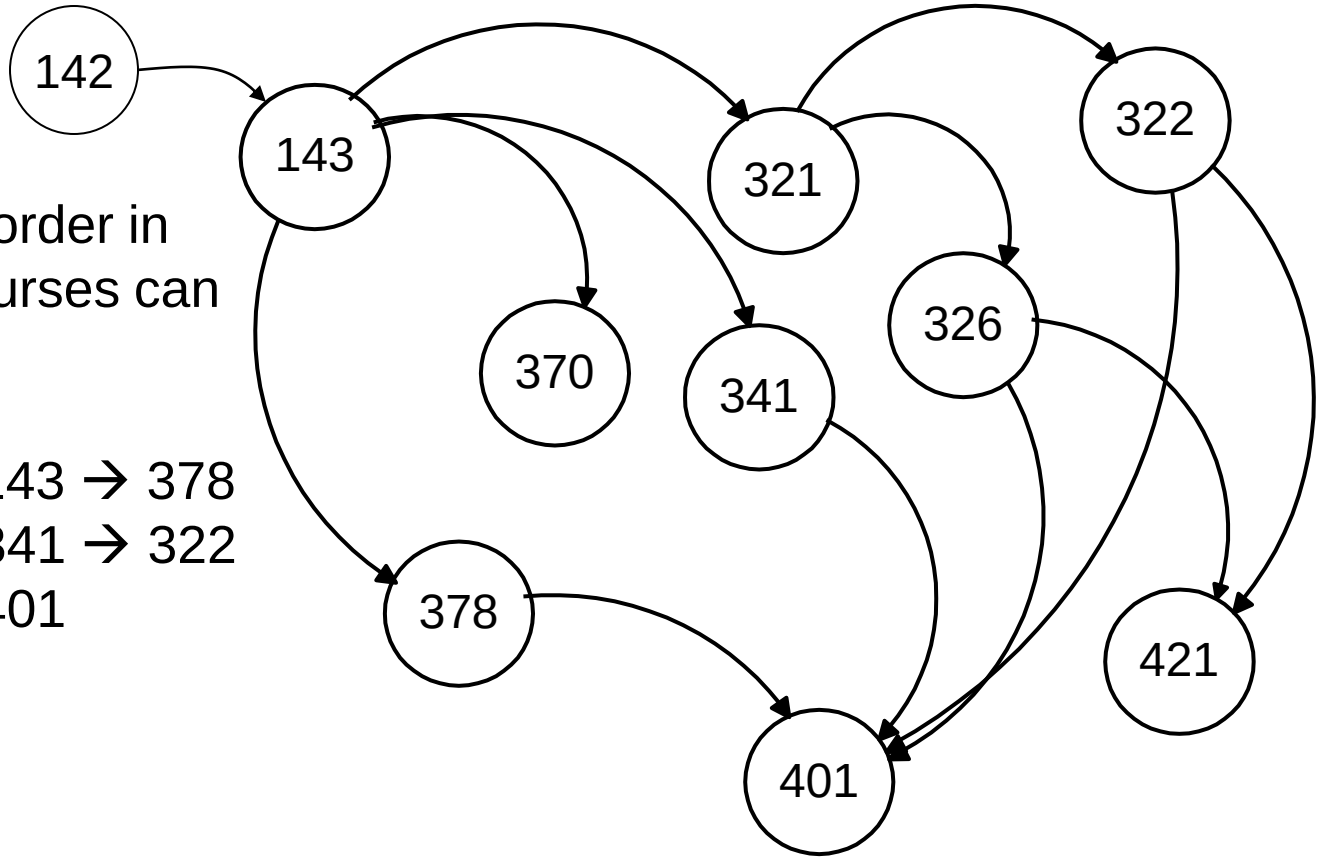
The process of laying out the vertices of a DAG(Directed Acyclic Graph) in linear order to meet the prerequisite rules is called a **topological sort.**

A topological sort may be found by performing a DFS (Depth First Search) on the graph.

Topological Sort

Problem: Find an order in which all these courses can be taken.

Example: 142 → 143 → 378 → 370 → 321 → 341 → 322 → 326 → 421 → 401

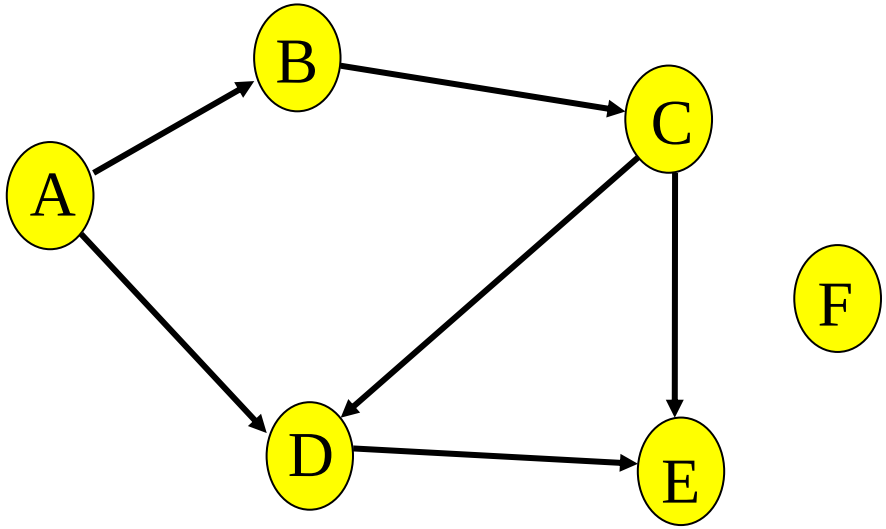


In order to take a course, you must take all of its prerequisites first

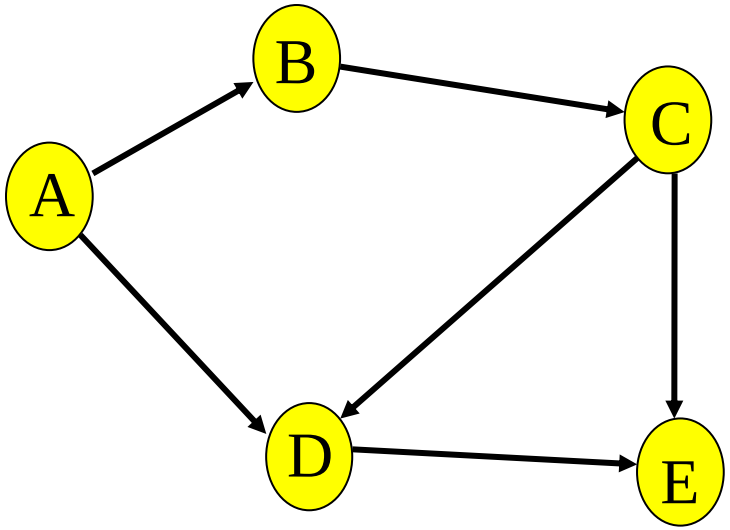
Topological Sort

Given a digraph $G = (V, E)$, find a linear ordering of its vertices such that:

for any edge (v, w) in E, v precedes w in the ordering

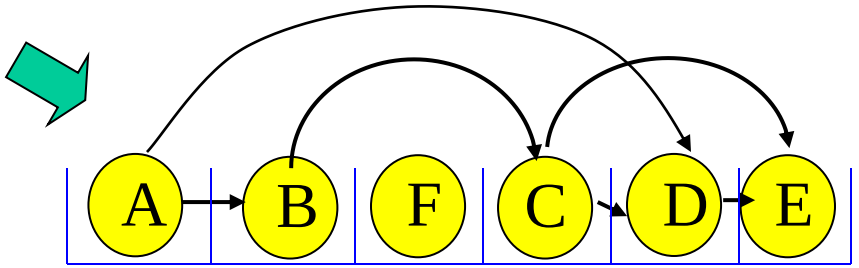


Topo sort - good example



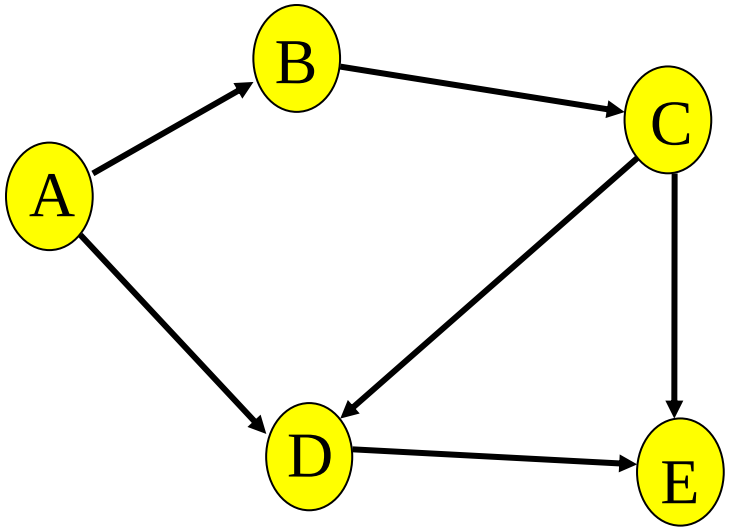
F

Any linear ordering in which all the arrows go to the right is a valid solution



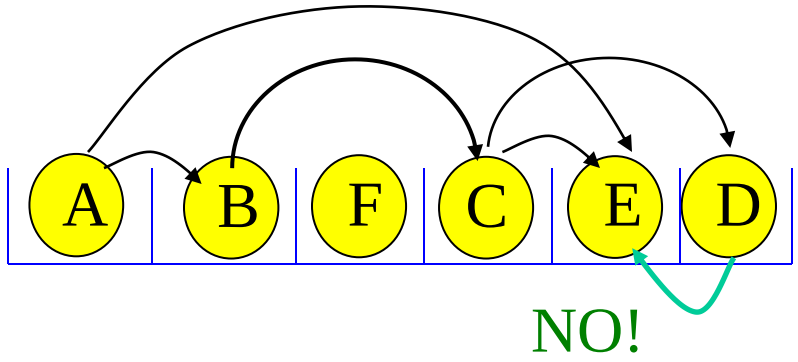
Note that F can go anywhere in this list because it is not connected.
Also the solution is not unique.

Topo sort - bad example



F

Any linear ordering in which an arrow goes to the left is not a valid solution

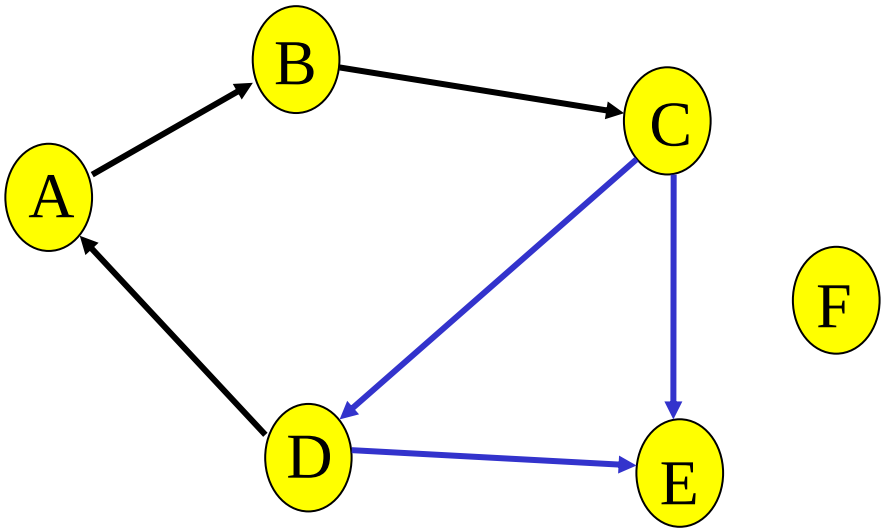


Paths and Cycles

- Given a digraph G = (V,E), a path is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that:
  - (v_i, v_{i+1}) in E for $1 \leq i < k$
  - path length = number of edges in the path
  - path cost = sum of costs of each edge
- A path is a cycle if :
  - $k > 1$; $v_1 = v_k$
- G is acyclic if it has no cycles.

Only acyclic graphs can be topo. sorted

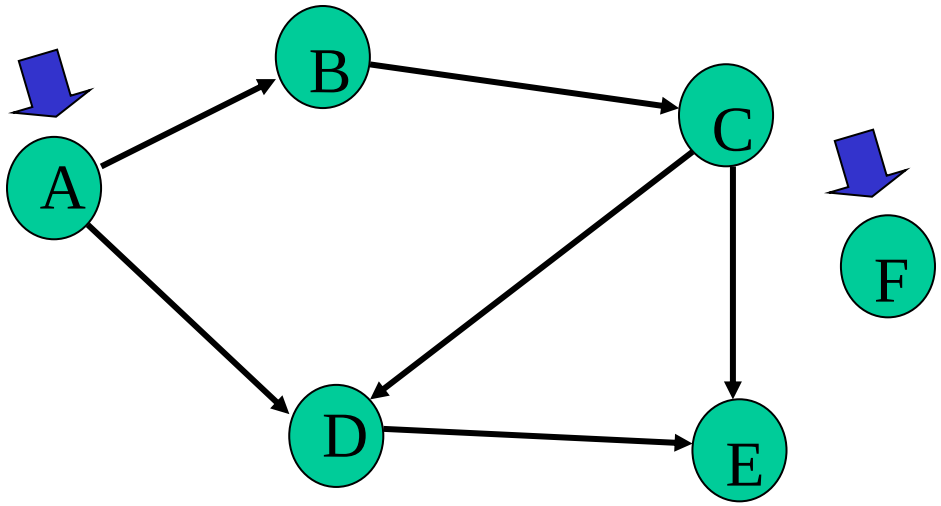
- A directed graph with a cycle cannot be topologically sorted.



Topo sort algorithm - 1

Step 1: Identify vertices that have no incoming edges

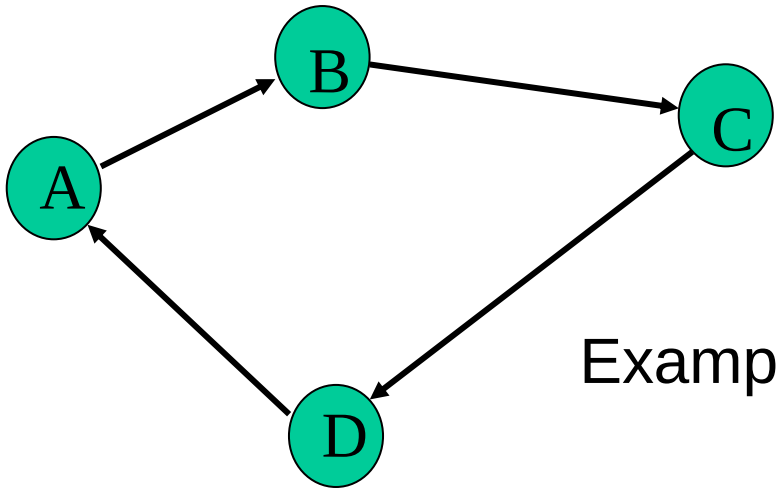
- The "in-degree" of these vertices is zero



Topo sort algorithm - 1a

Step 1: Identify vertices that have no incoming edges

- If *no such vertices*, graph has only cycle(s) (cyclic graph)
- Topological sort not possible – Halt.

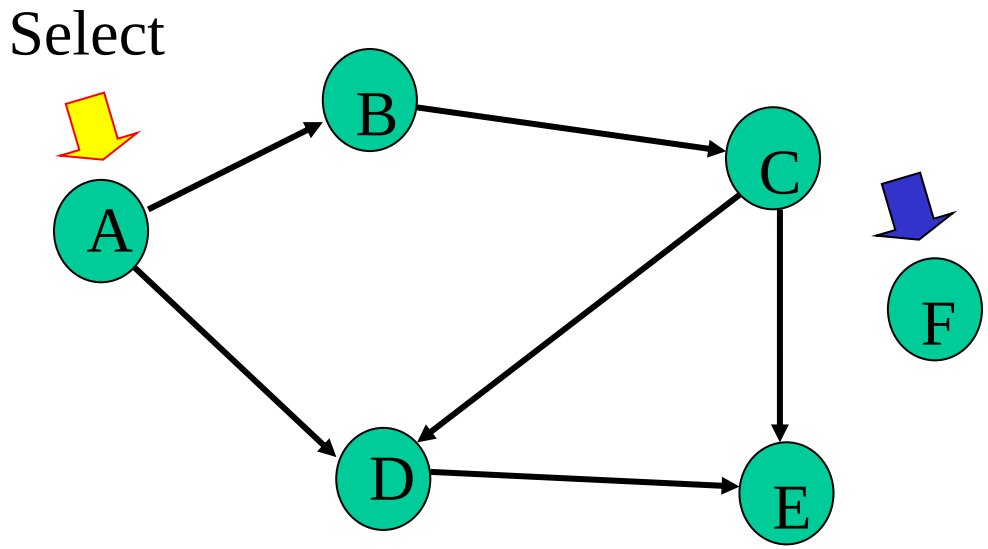


Example of a cyclic graph

Topo sort algorithm - 1b

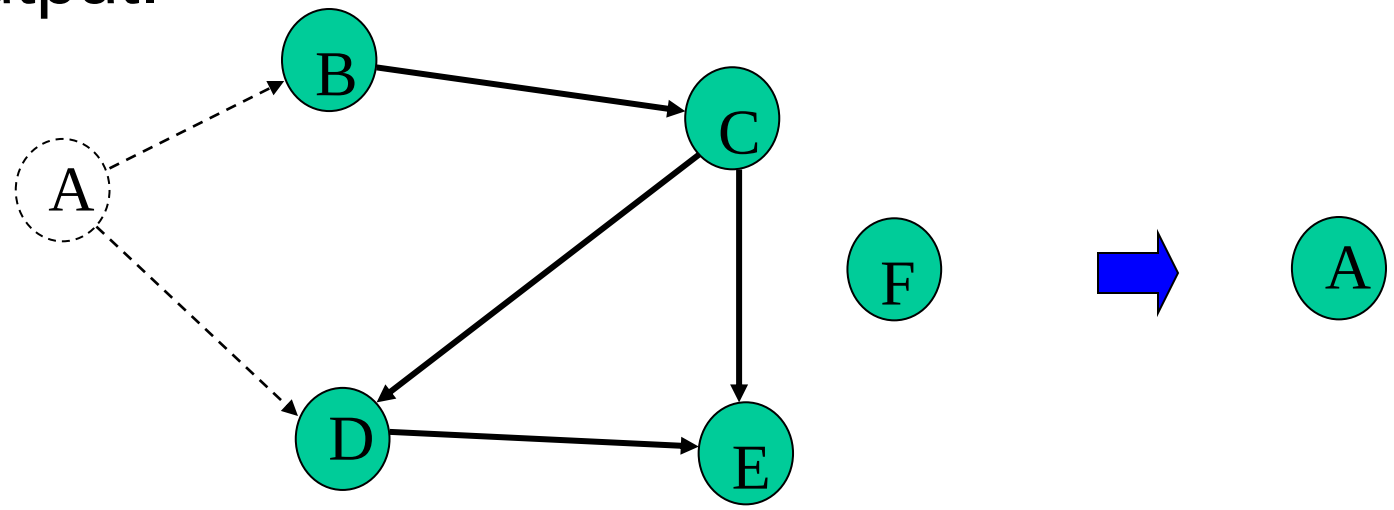
Step 1: Identify vertices that have no incoming edges

- Select one such vertex



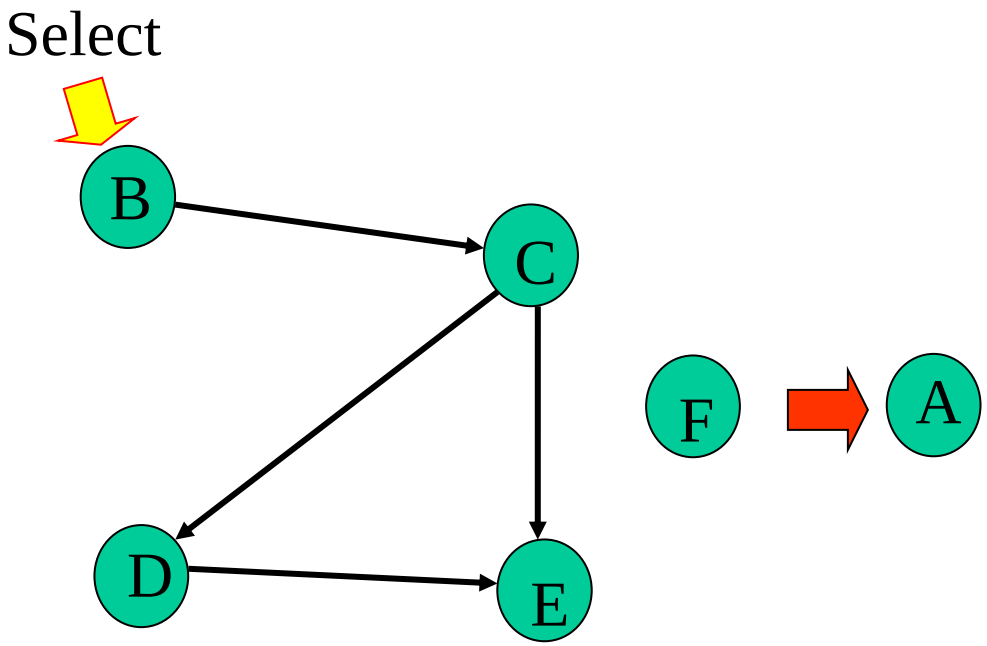
Topo sort algorithm - 2

Step 2: Delete this vertex of in-degree 0 and all its outgoing edges from the graph. Place it in the output.



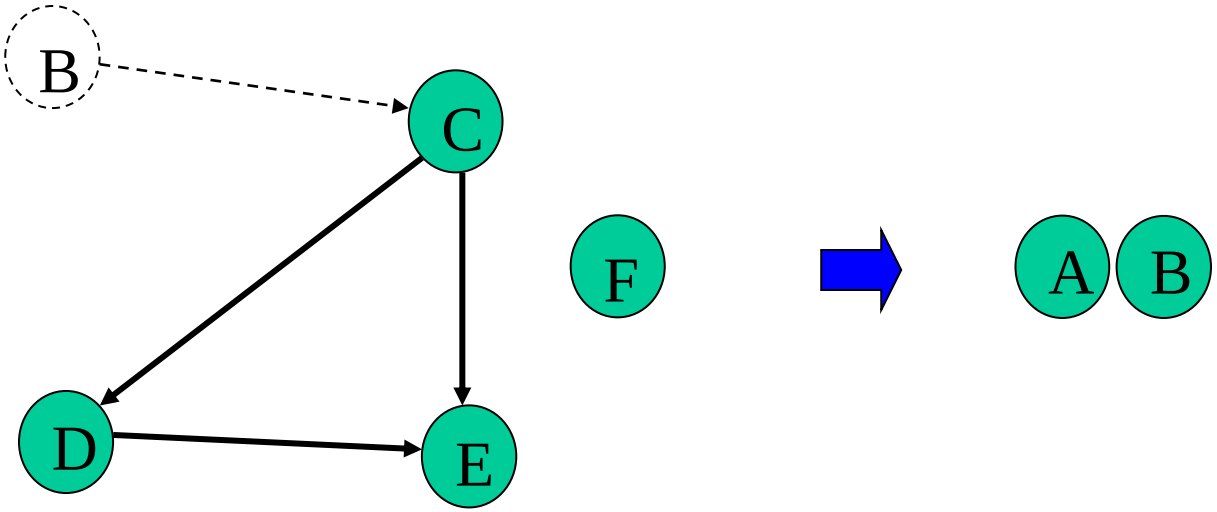
Continue until done

Repeat Step 1 and Step 2 until graph is empty



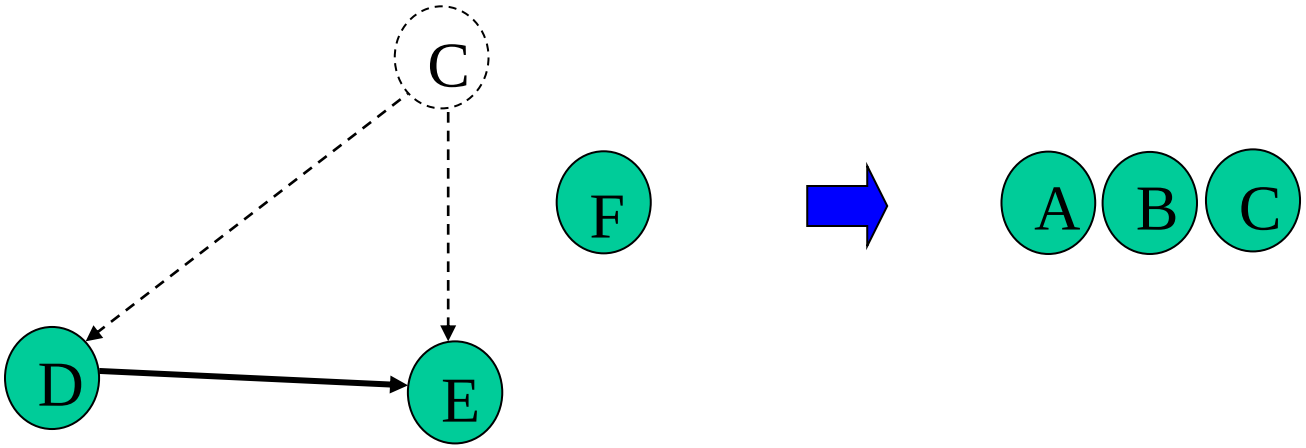
B

Select B. Copy to sorted list. Delete B and its edges.



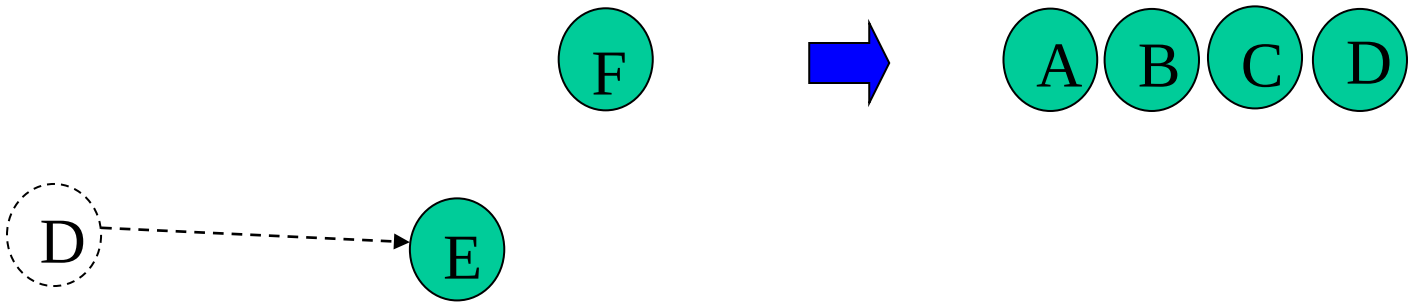
C

Select C. Copy to sorted list. Delete C and its edges.



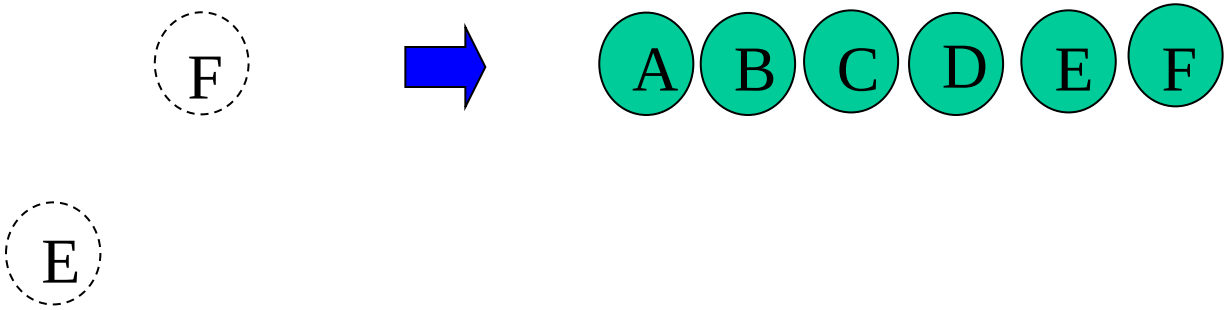
D

Select D. Copy to sorted list. Delete D and its edges.

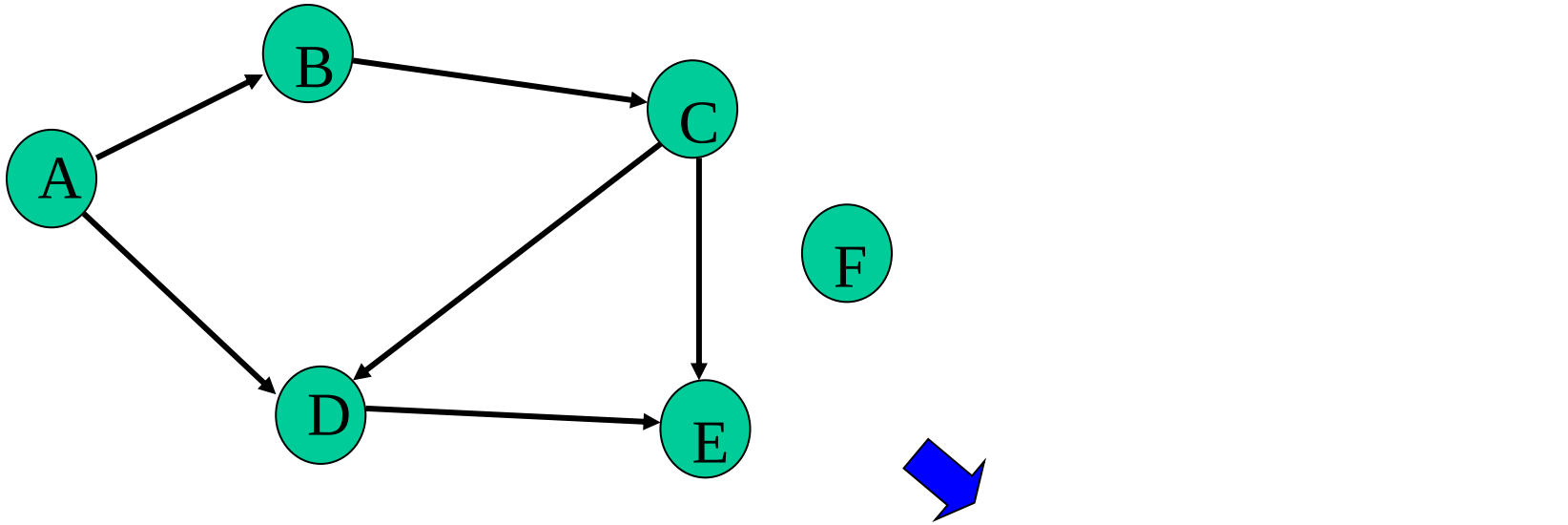


E, F

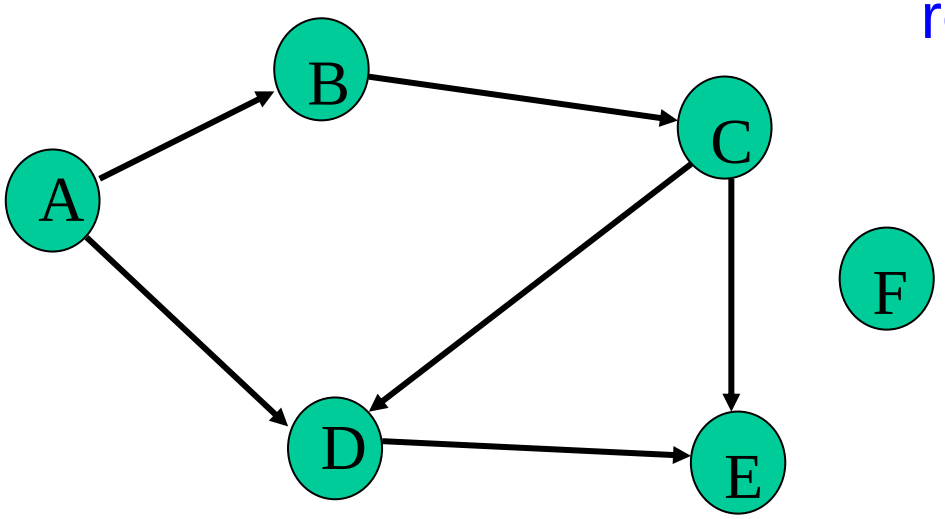
Select E. Copy to sorted list. Delete E and its edges.
Select F. Copy to sorted list. Delete F and its edges.



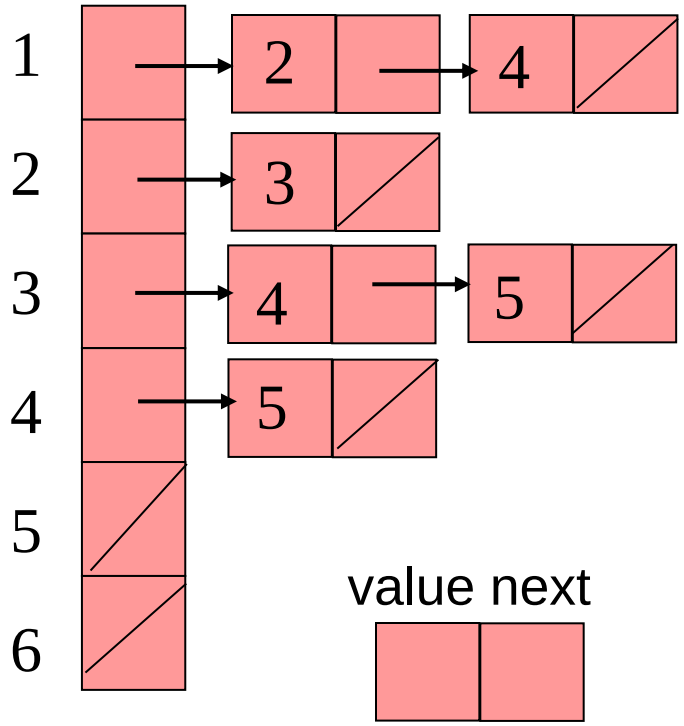
Done



Implementation



Assume adjacency list representation

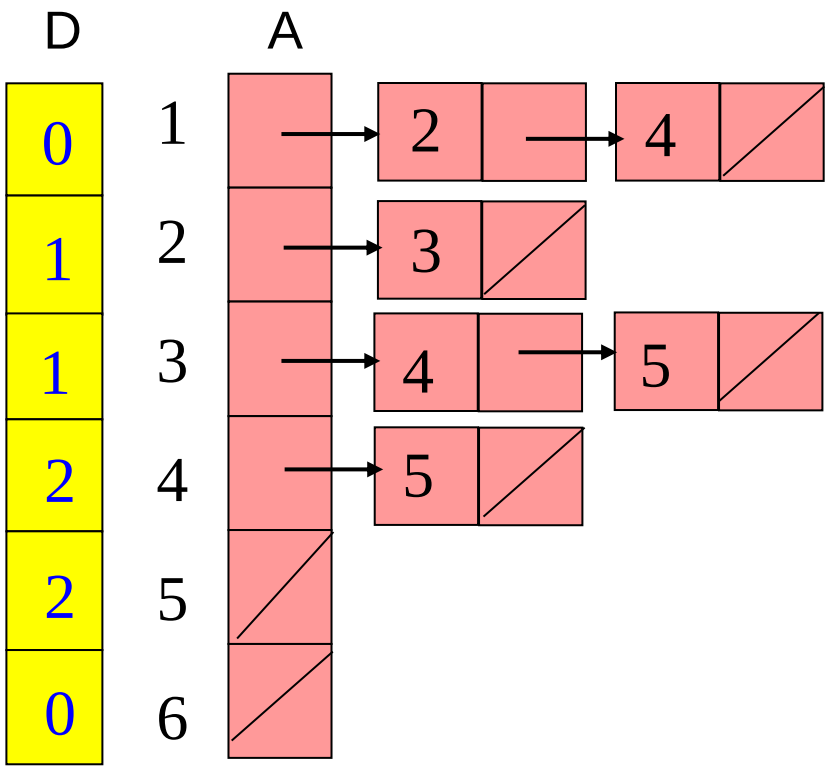


Translation array

1	2	3	4	5	6
A	B	C	D	E	F

Calculate In-degrees

In-Degree array; or add a field to array A

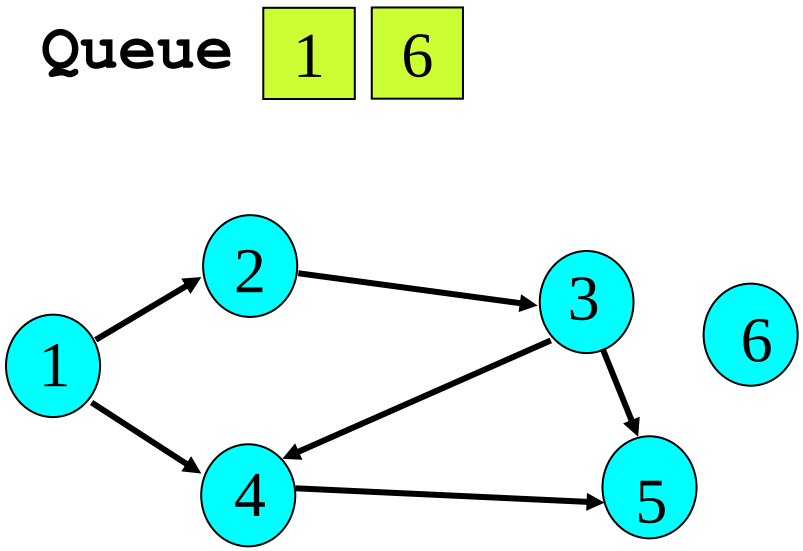


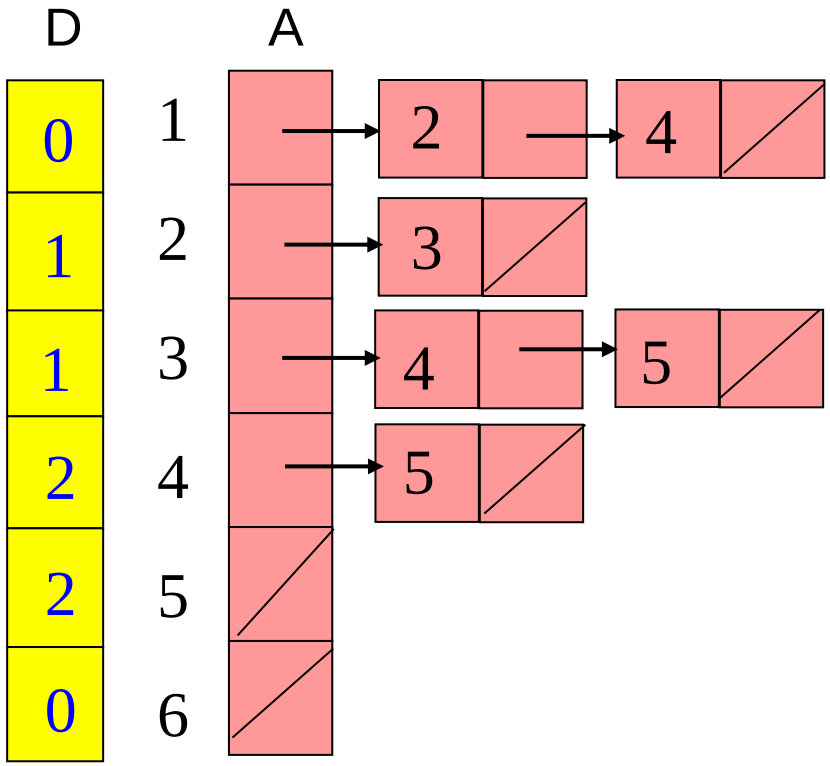
Calculate In-degrees

```
for i = 1 to n do D[i] := 0; endfor
for i = 1 to n do
  x := A[i];
  while x ≠ null do
    D[x.value] := D[x.value] + 1;
    x := x.next;
  endwhile
endfor
```

Maintaining Degree 0 Vertices

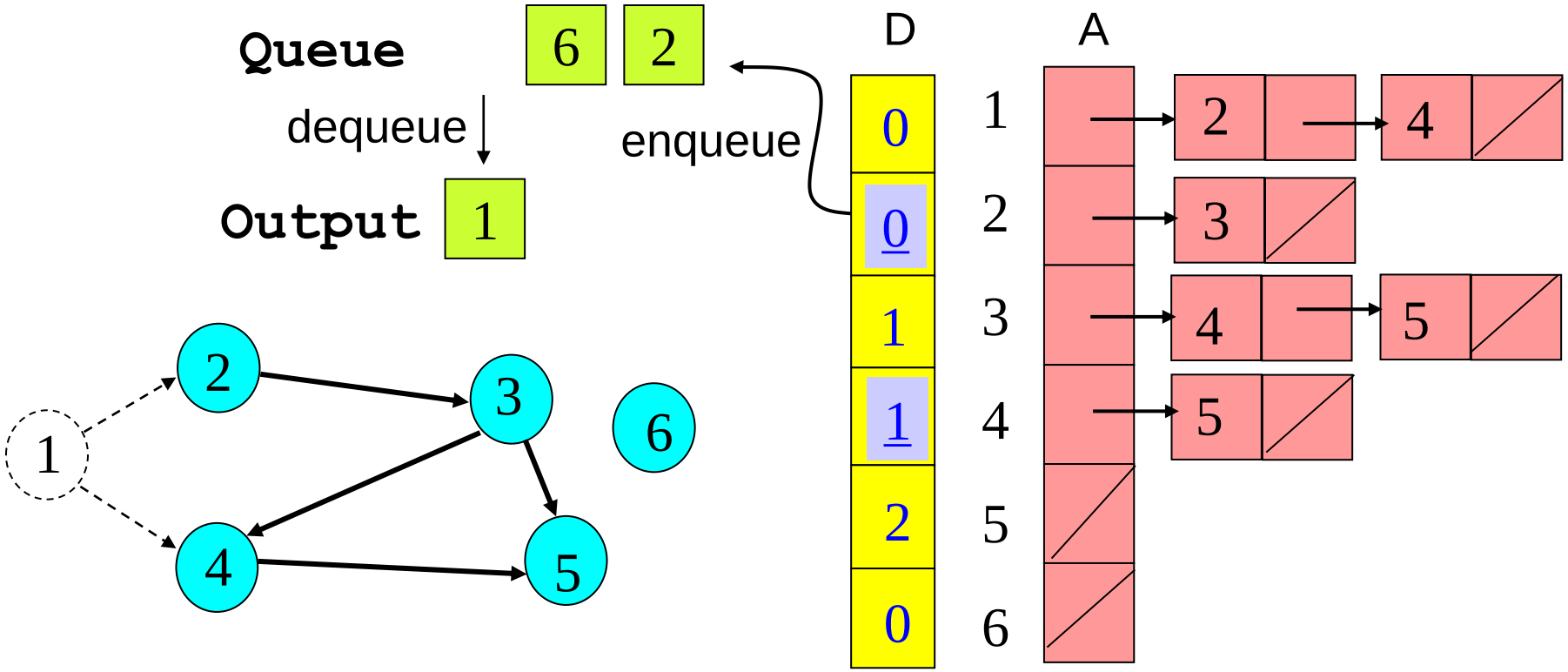
Key idea: Initialize and maintain a *queue (or stack)* of vertices with In-Degree 0





Topo Sort using a Queue (breadth-first)

After each vertex is output, when updating In-Degree array, *enqueue any vertex whose In-Degree becomes zero*



Topological Sort Algorithm

1. Store each vertex's In-Degree in an array D
2. Initialize queue with all "in-degree=0" vertices
3. While there are vertices remaining in the queue:
   (a) Dequeue and output a vertex
   (b) Reduce In-Degree of all vertices adjacent to it by 1
   (c) Enqueue any of these vertices whose In-Degree became zero
4. If all vertices are output then success, otherwise there is a cycle.

Some Detail

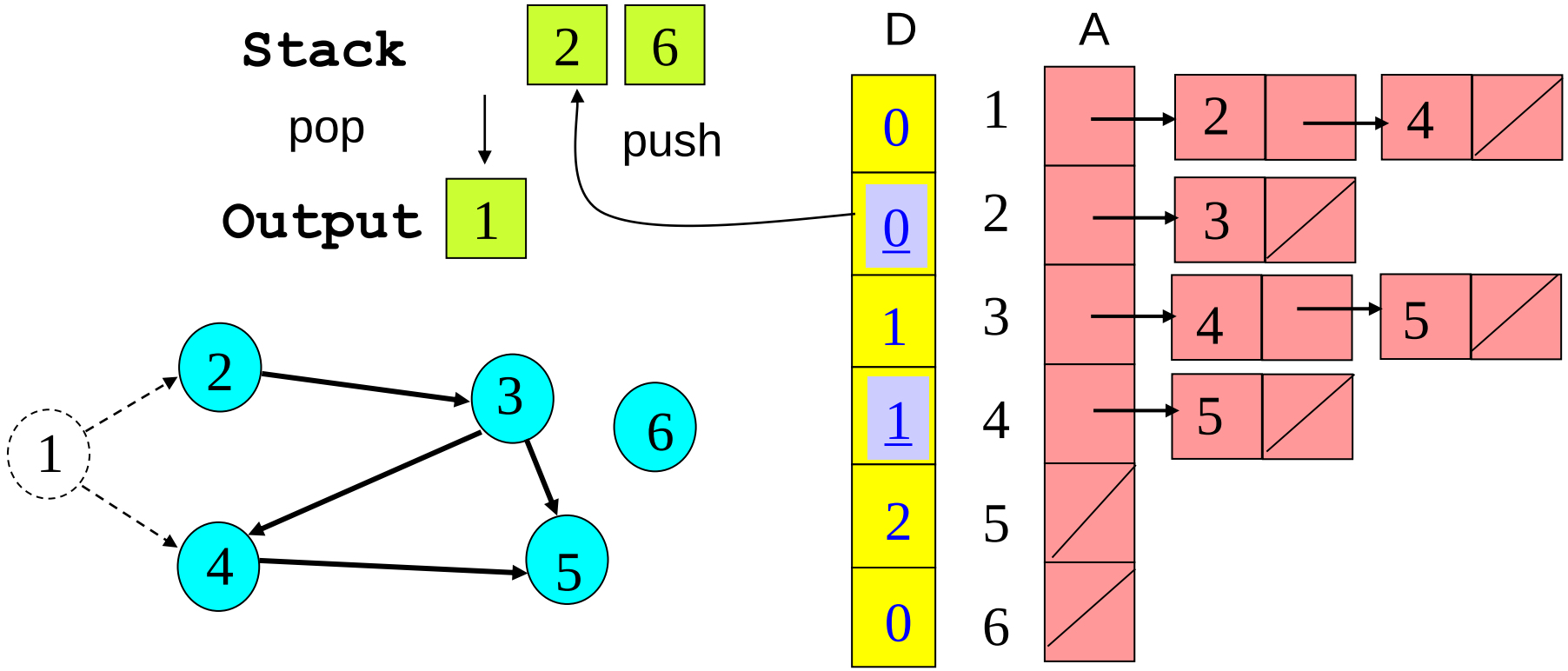
```
Main Loop
while notEmpty(Q) do
  x := Dequeue(Q)
  Output(x)
  y := A[x];
  while y ≠ null do
    D[y.value] := D[y.value] - 1;
    if D[y.value] = 0 then Enqueue(Q,y.value);
    y := y.next;
  endwhile
endwhile
```

Topological Sort Analysis

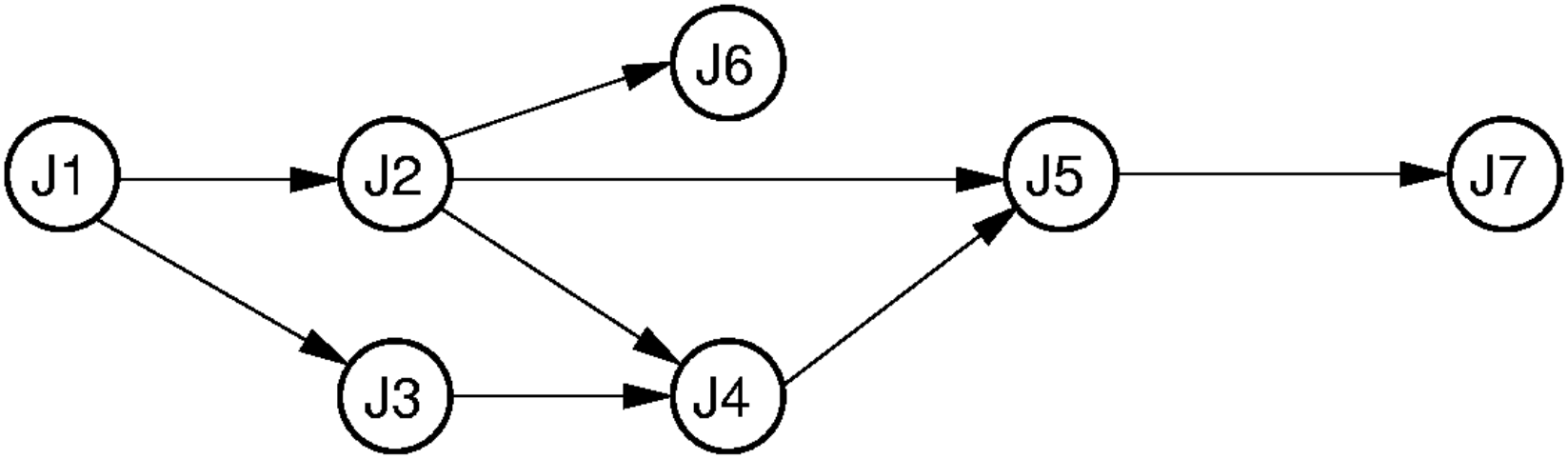
- Initialize In-Degree array: O(|V| + |E|)
- Initialize Queue with In-Degree 0 vertices: O(|V|)
- Dequeue and output vertex:
  - |V| vertices, each takes only O(1) to dequeue and output: O(|V|)
- Reduce In-Degree of all vertices adjacent to a vertex and Enqueue any In-Degree 0 vertices:
  - O(|E|)
- For input graph G=(V,E) run time = O(|V| + |E|)
  - Linear time!

Topo Sort using a Stack (depth-first)

After each vertex is output, when updating In-Degree array, *push any vertex whose In-Degree becomes zero*



Topological Sort

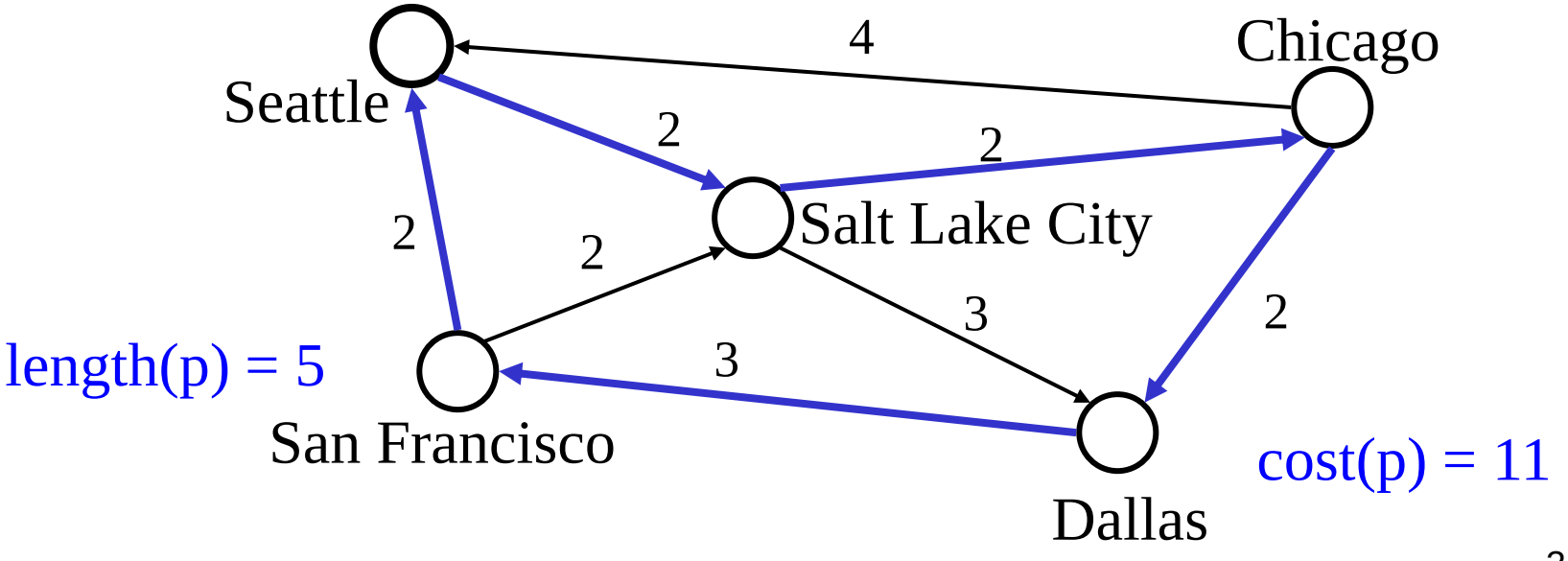


Prints in reverse order:
J7,J5,J4,J6,J2,J3,J1

Shortest path problems

Recall Path cost ,Path length

- Path cost: the sum of the costs of each edge
- Path length: the number of edges in the path
  - Path length is the unweighted path cost



Shortest path problems and Why study ?

- Given a graph G = (*V, E*) and a "source" vertex *s* in *V*, find the minimum cost paths from *s* to every vertex in *V*

- Traveling on a budget: What is the cheapest airline schedule from Seattle to city X?
- Optimizing routing of packets on the internet:
  - Vertices are routers and edges are network links with different delays. What is the routing path with smallest total delay?
- Shipping: Find which highways and roads to take to minimize total delay due to traffic

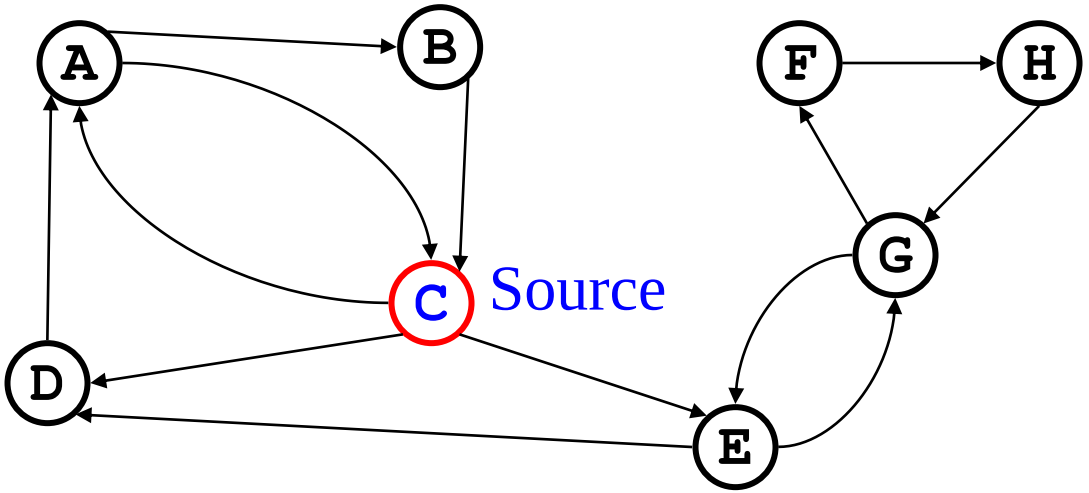
Unweighted Shortest Path

Problem: Given a "source" vertex *s* in an unweighted directed graph

G = (*V*,*E*), find the shortest path from *s* to all vertices in G

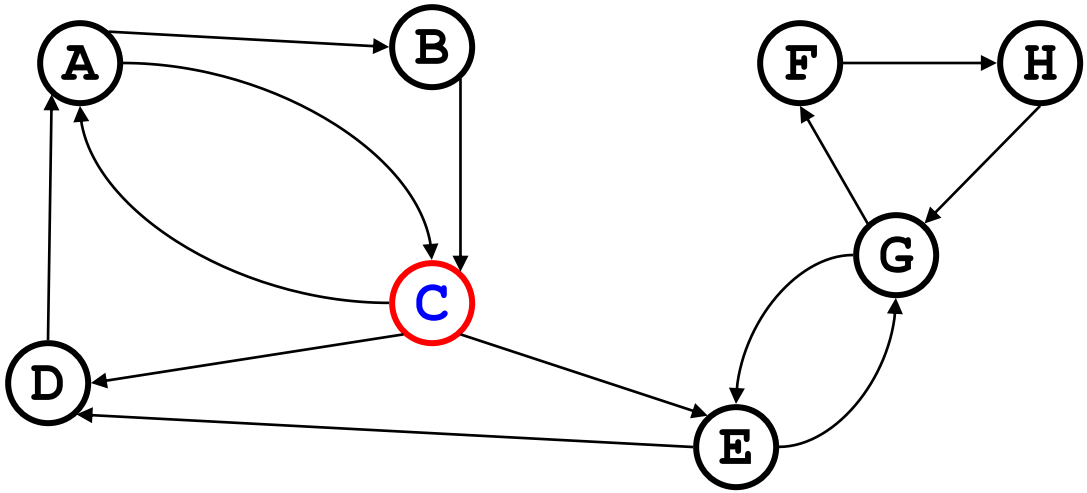
Only interested

in path lengths

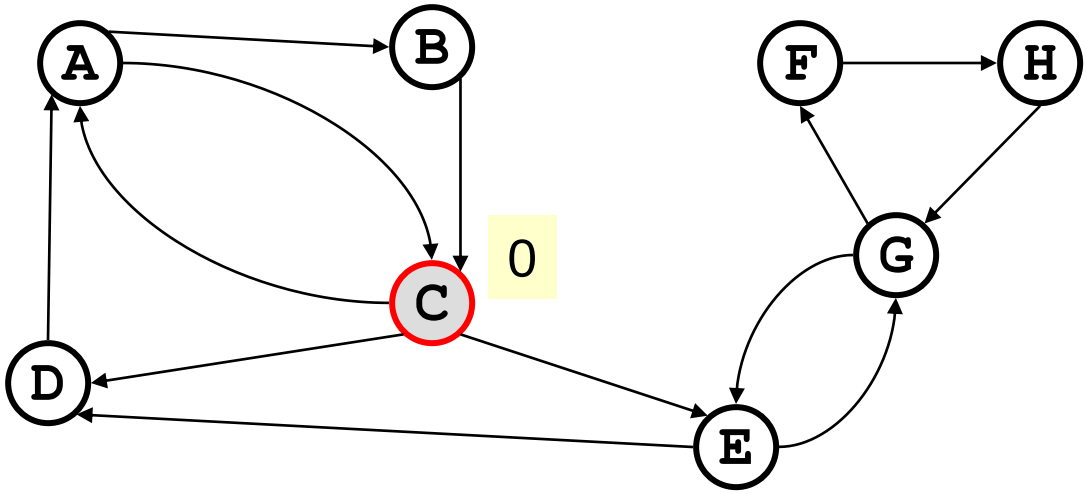


Breadth-First Search Solution

- Basic Idea: Starting at node s, find vertices that can be reached using 0, 1, 2, 3, …, N-1 edges (works even for cyclic graphs!)

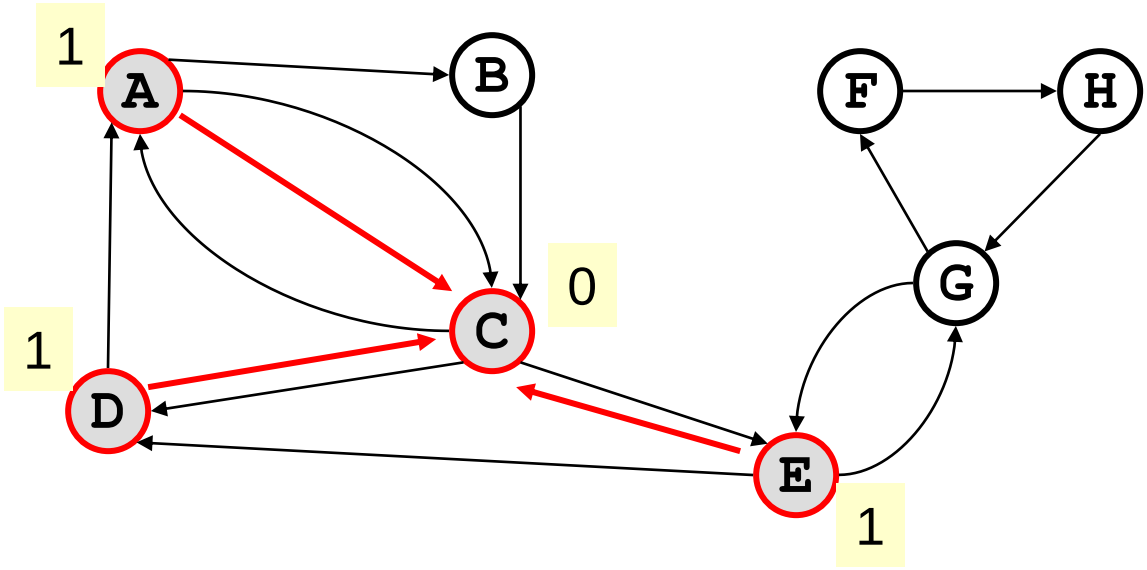


Example: Shortest Path length



Queue Q = C

Example (ct'd)

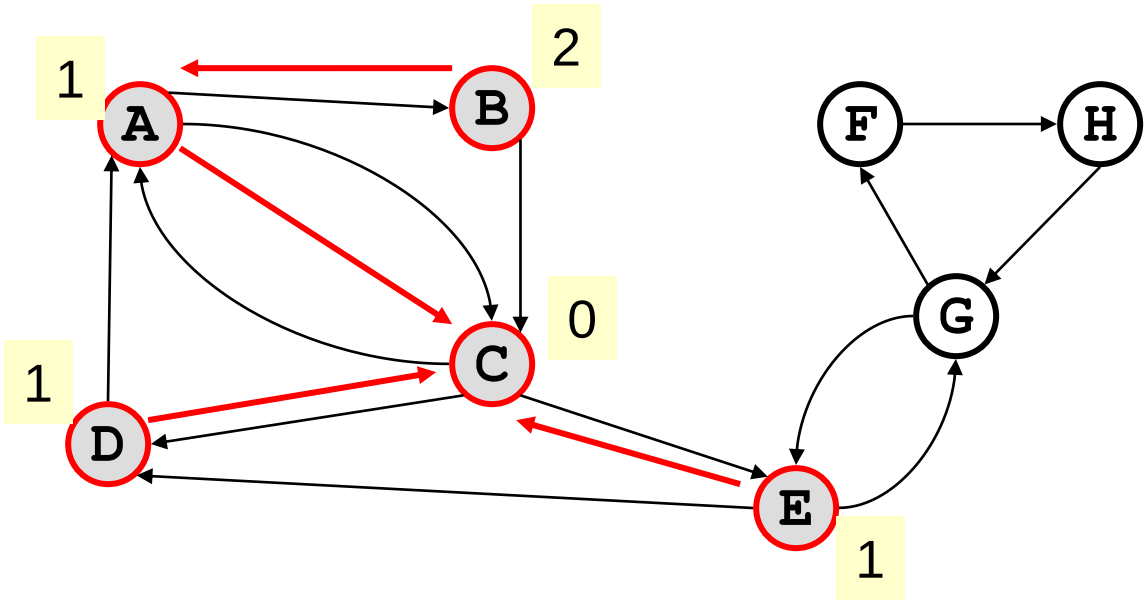


Queue Q = A D E

Previous pointer

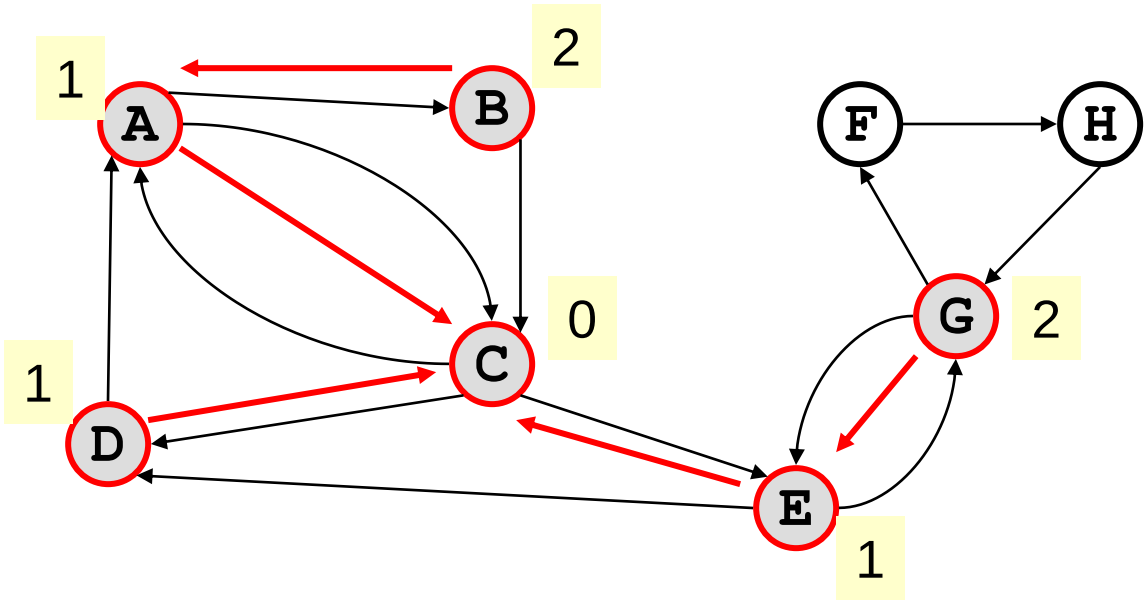
Indicates the vertex is marked

Example (ct'd)



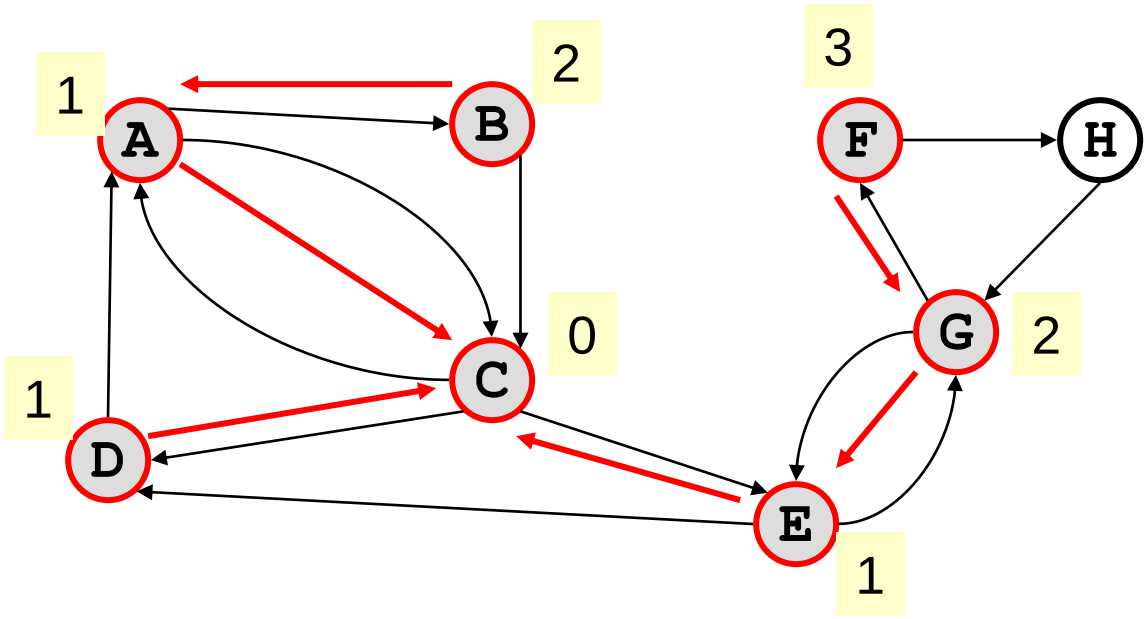
Q = D E B

Example (ct'd)



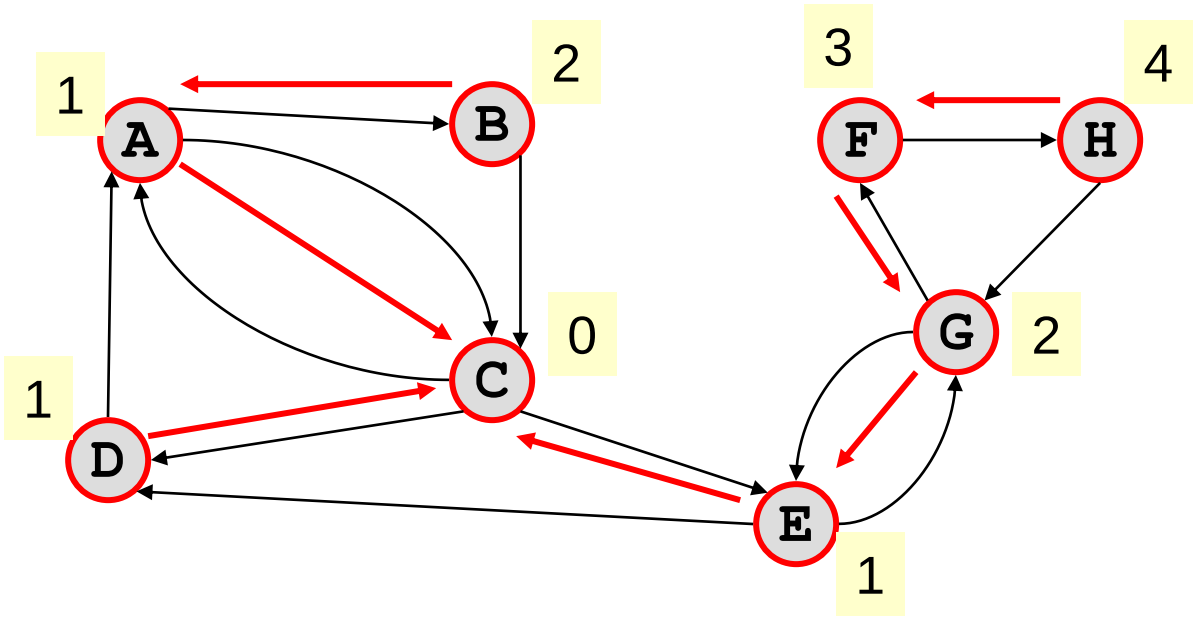
Q = B G

Example (ct'd)



Q = F

Example (ct'd)



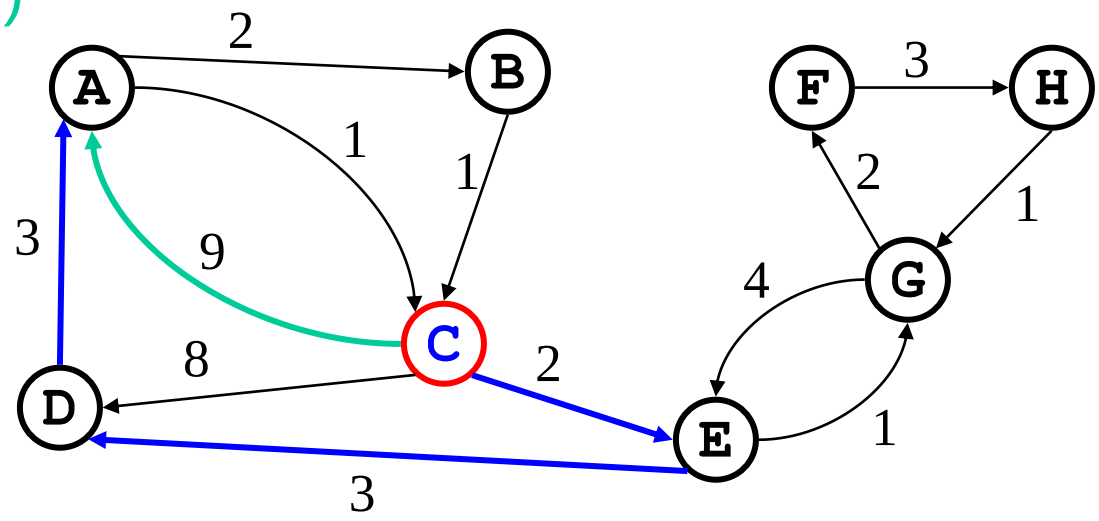
Q = H

What if edges have weights?

- Breadth First Search does not work anymore
  - minimum *cost* path may have more edges than minimum *length* path

Shortest path (length) from C to A:
C→A (cost = 9)

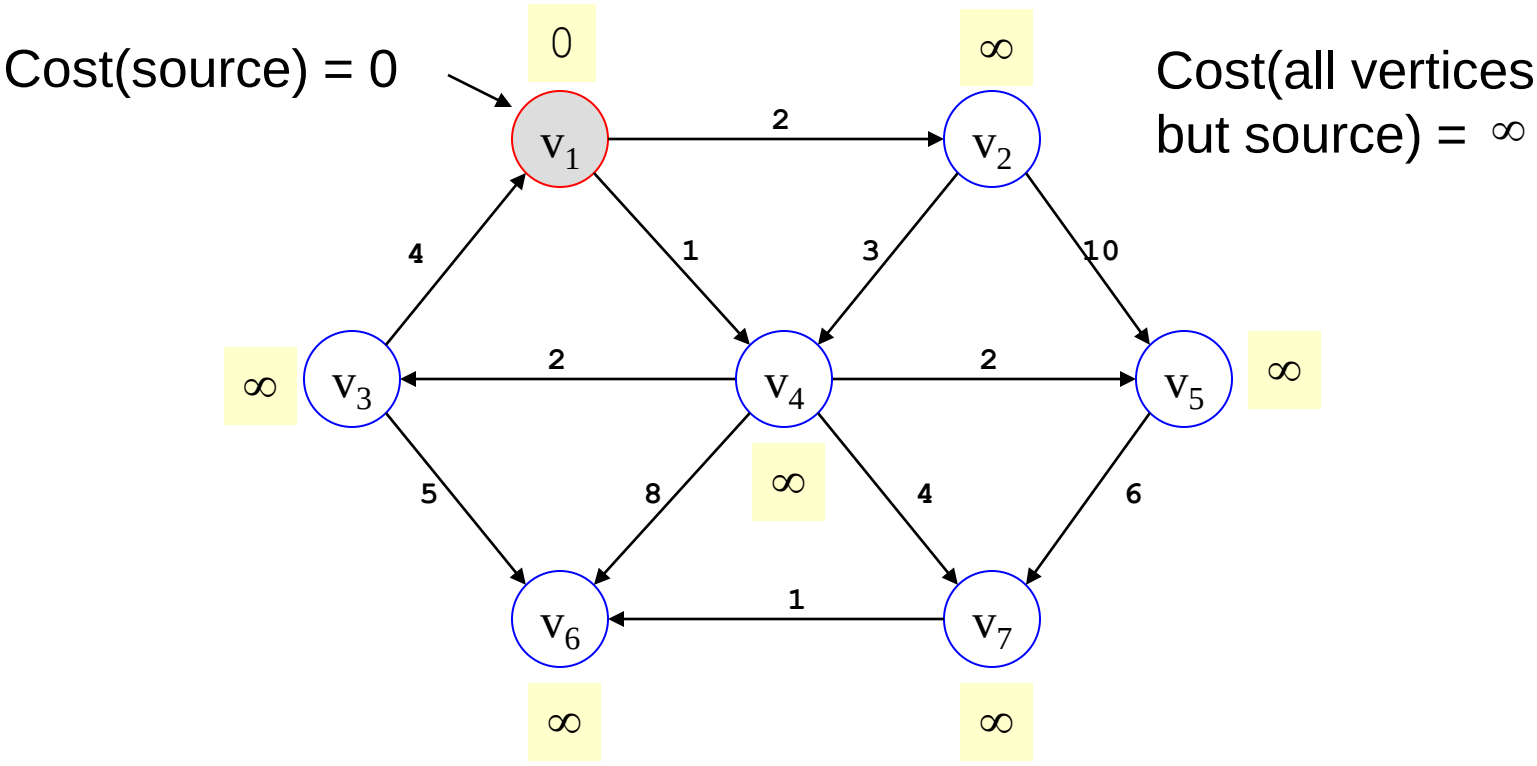
Minimum Cost Path = C→E→D→A (cost = 8)



Dijkstra's Shortest Path Algorithm

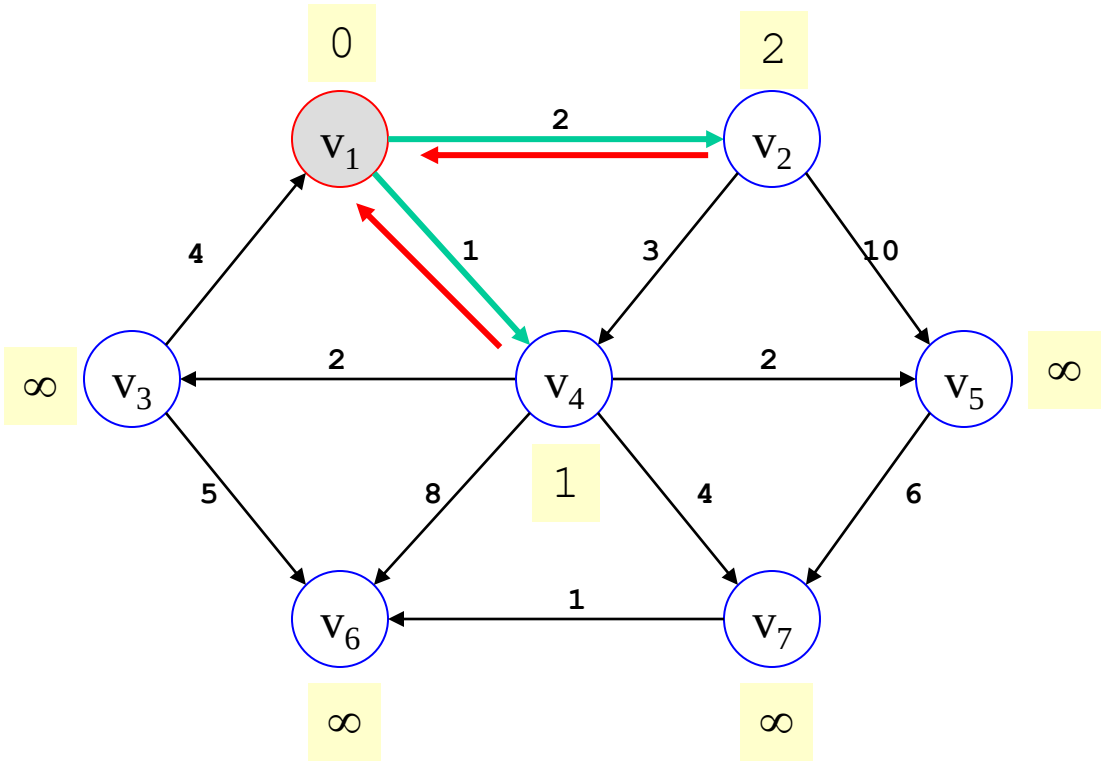
- Initialize the cost of s to 0, and all the rest of the nodes to ∞
- Initialize set S to be $\varnothing$
  - S is the set of nodes to which we have a shortest path
- While S is not all vertices
  - Select the node A with the lowest cost that is not in S and identify the node as now being in S
  - for each node B adjacent to A
    - if cost(A)+cost(A,B) < B's currently known cost
      - set cost(B) = cost(A)+cost(A,B)
      - set previous(B) = A so that we can remember the path

Example: Initialization



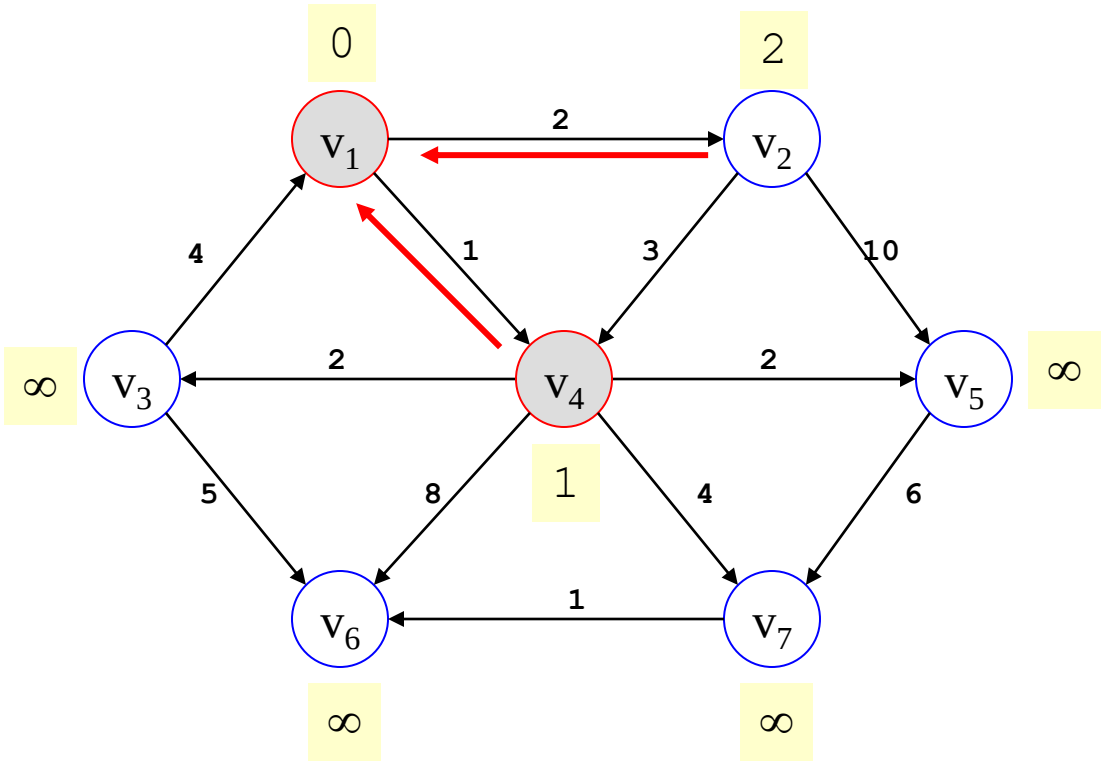
Pick vertex not in S with lowest cost.

Example: Update Cost neighbors



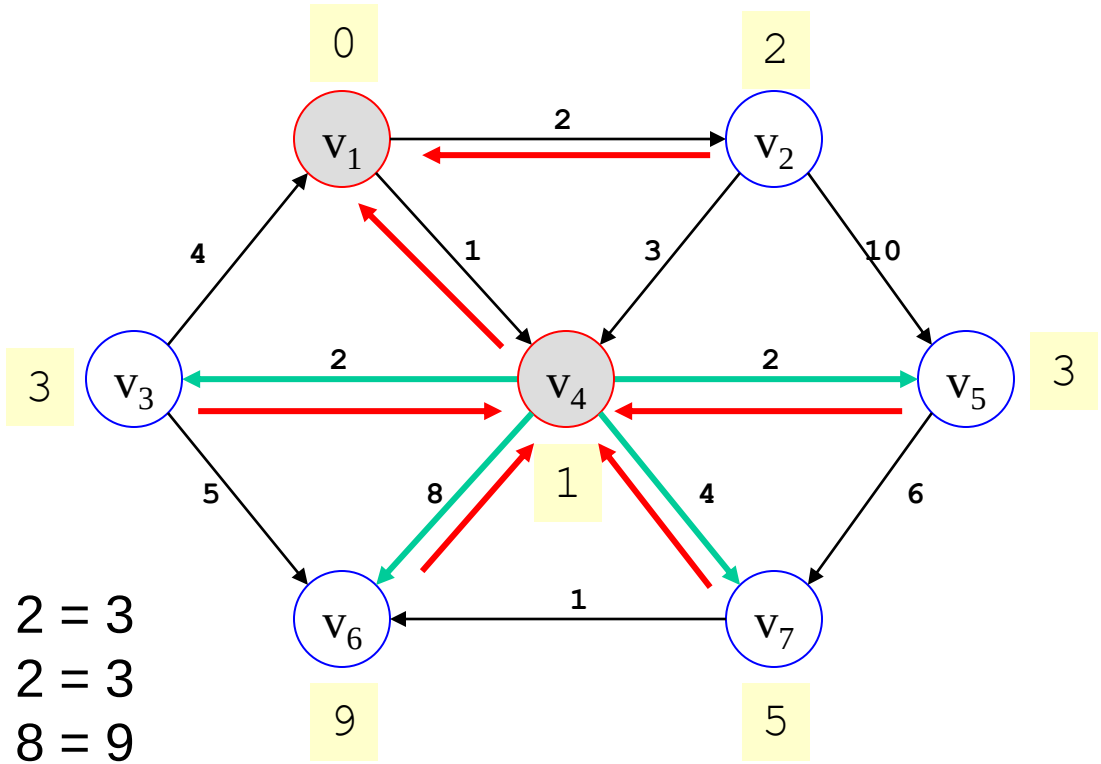
Cost(v_2) = 2
Cost(v_4) = 1

Example: pick vertex with lowest cost and add it to S



Pick vertex not in S with lowest cost, i.e., v_4

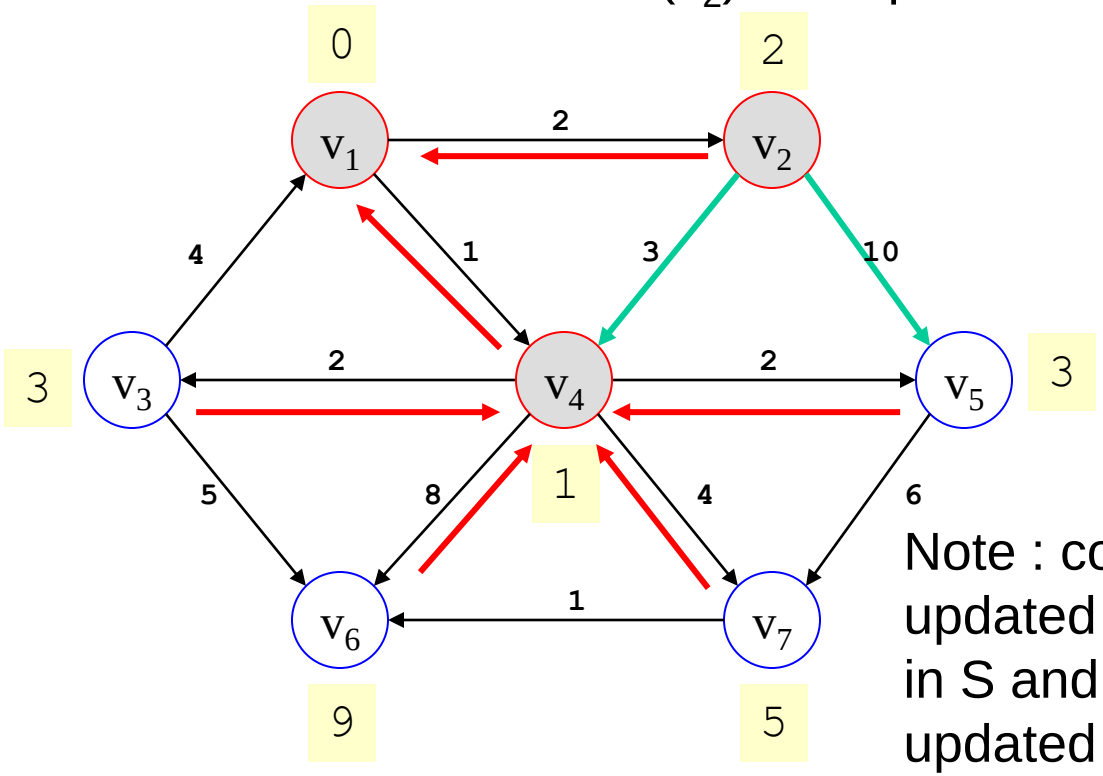
Example: update neighbors



Cost(v_3) = 1 + 2 = 3
Cost(v_5) = 1 + 2 = 3
Cost(v_6) = 1 + 8 = 9
Cost(v_7) = 1 + 4 = 5

Example (Ct'd)

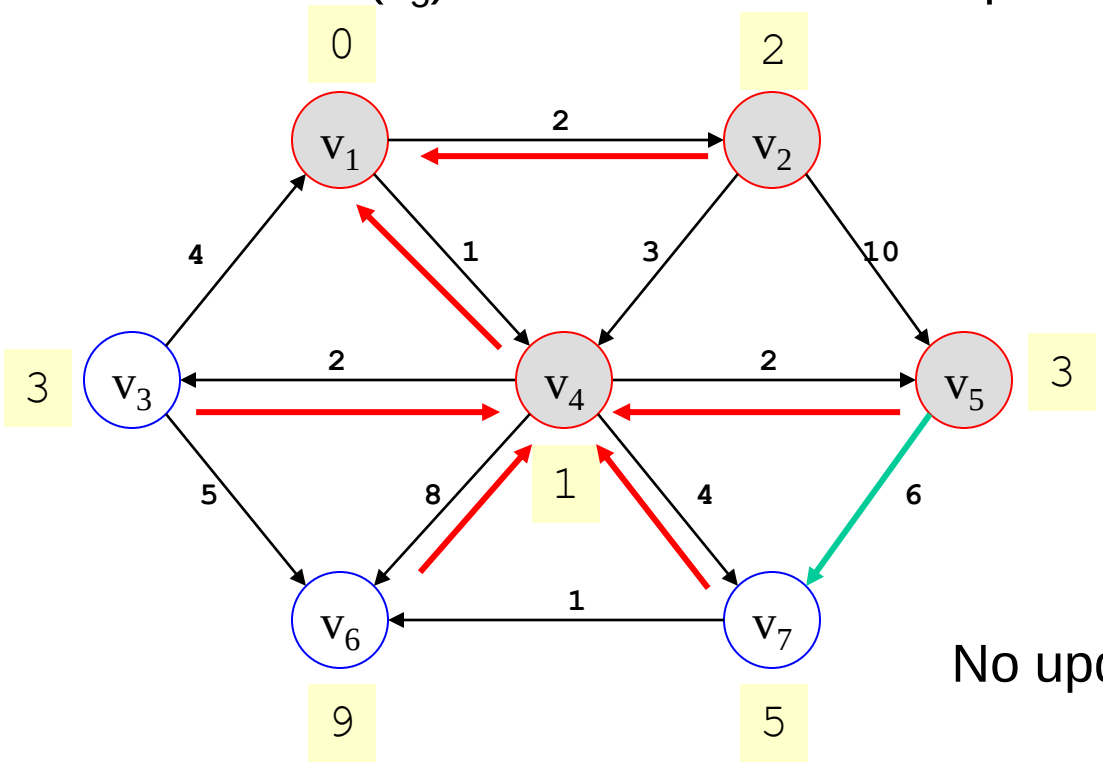
Pick vertex not in S with lowest cost (v_2) and update neighbors



Note : cost(v_4) not updated since already in S and cost(v_5) not updated since it is larger than previously computed

Example: (ct'd)

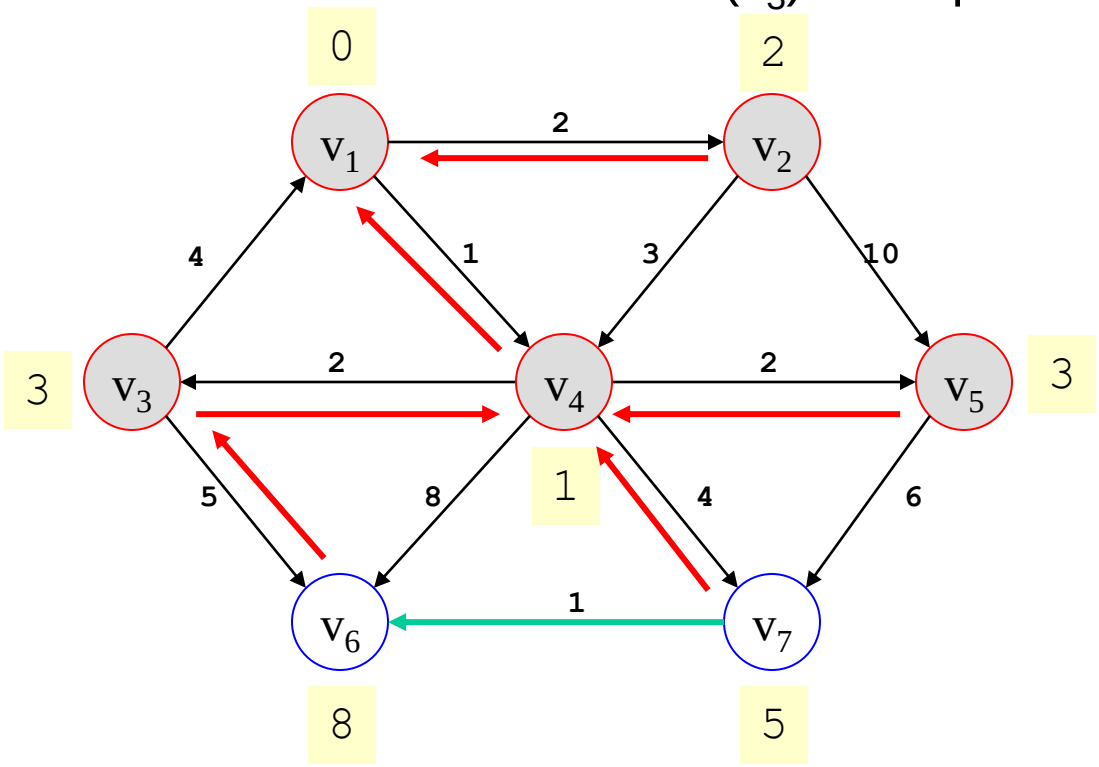
Pick vertex not in S (v_5) with lowest cost and update neighbors



No updating

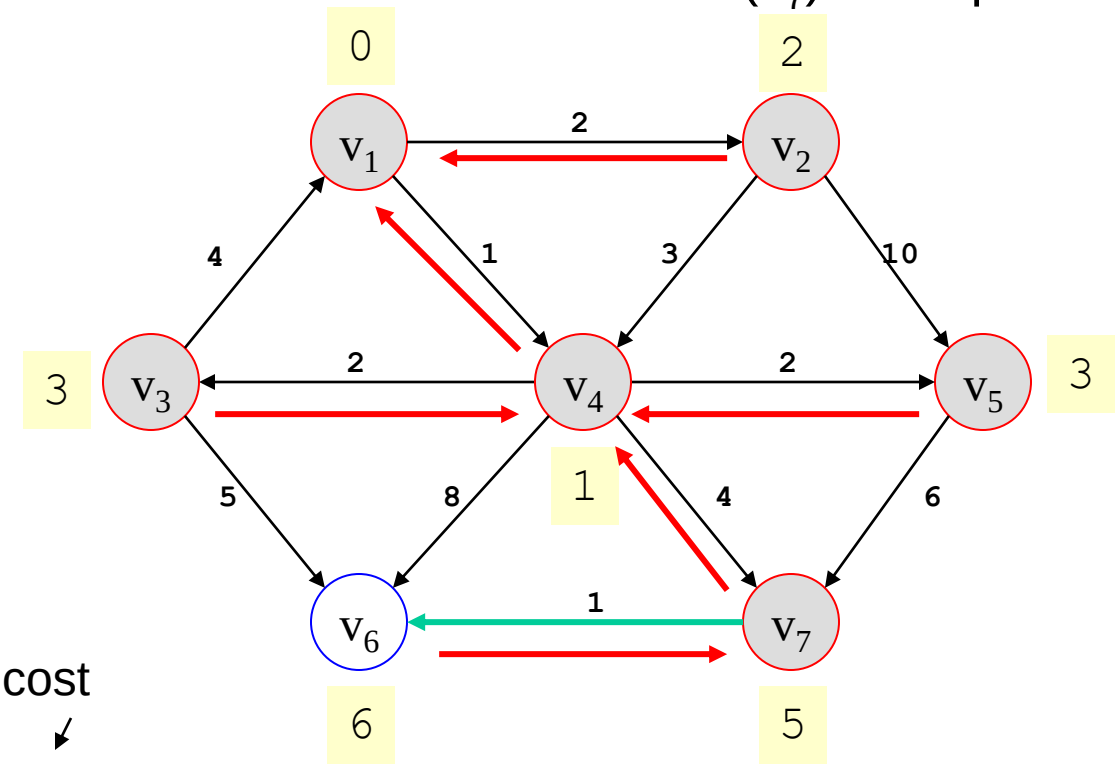
Example: (ct'd)

Pick vertex not in S with lowest cost (v_3) and update neighbors



Example: (ct'd)

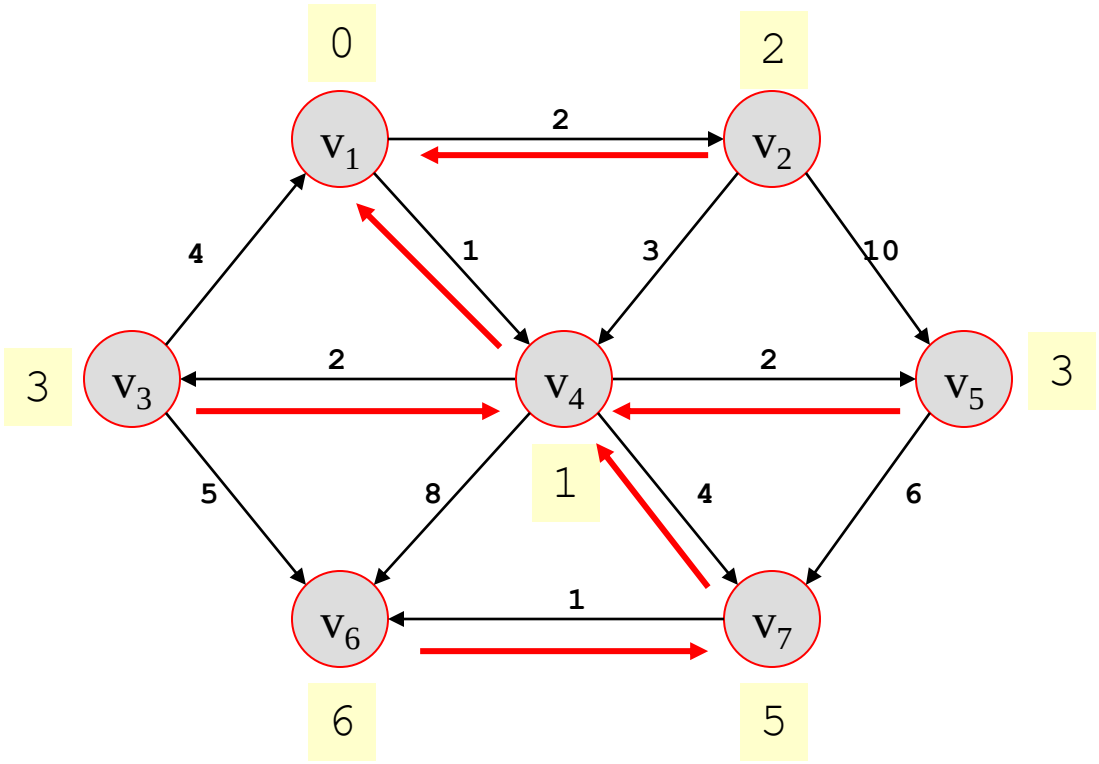
Pick vertex not in S with lowest cost (v_7) and update neighbors



Previous cost

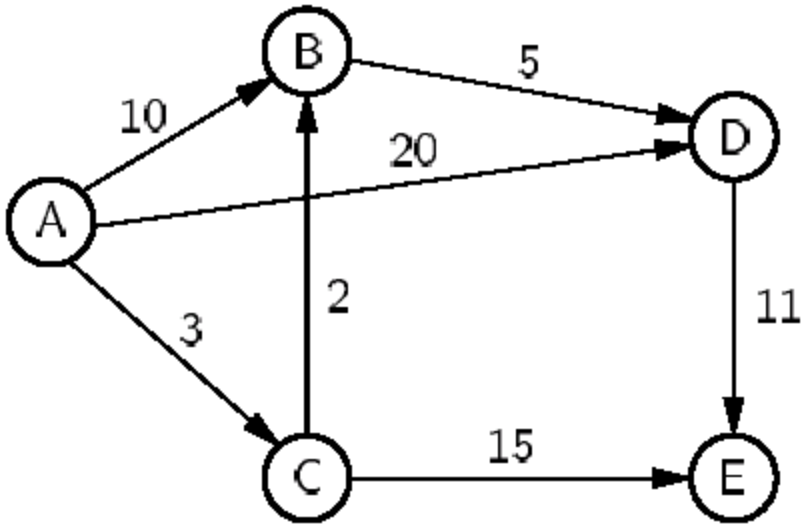
Cost(v_6) = min (8, 5+1) = 6

Example (end)



Pick vertex not in S with lowest cost (v_6) and update neighbors

Dijkstra's Algorithm Example



	A	B	C	D	E
Initial	0	∞	∞	∞	∞
Process A	0	10	3	20	∞
Process C	0	5	3	20	18
Process B	0	5	3	10	18
Process D	0	5	3	10	18
Process E	0	5	3	10	18

Visiting order of vertices: A, C, B, D, E

Minimal Cost Spanning Trees

Minimal Cost Spanning Trees

- Minimal Cost Spanning Tree (MST) is the graph containing the vertices of G along with the subset of G's edges.

  Input: An undirected, connected graph G.

  Output: The subgraph of G that

  1) has minimum total cost as measured by summing the values of all the edges in the subset

  2) keeps the vertices connected.

Applications:

Soldering the shortest set of wires needed to connect a set of terminals on a circuit board

Connecting a set of cities by telephone lines in such a way as to require the least amount of cable.

Recall Spanning Tree

- Given (connected) graph G(V,E),

  a spanning tree T(V',E'):

  › Spans the graph (V' = V)

  › Forms a tree (no cycle);

  › E' has |V| -1 edges

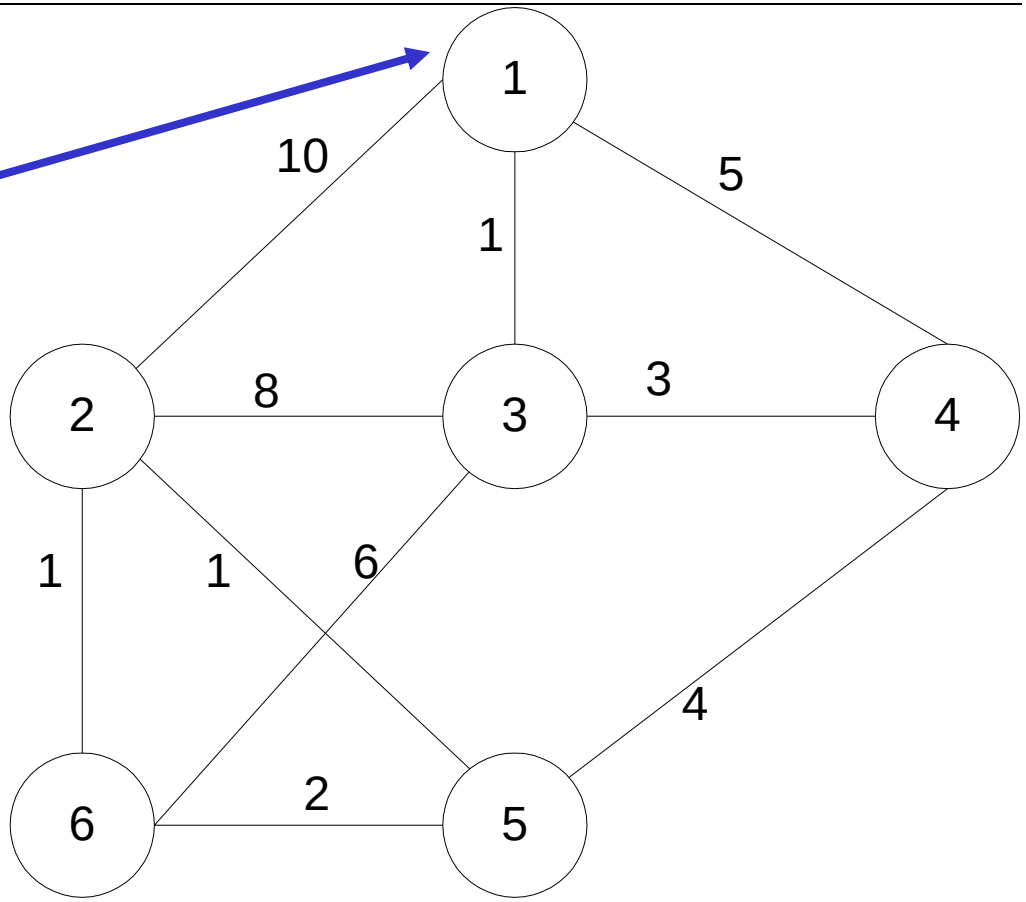
Two Algorithms

- Prim: (build tree incrementally)
  - Pick lower cost edge connected to known (incomplete) spanning tree that does not create a cycle and expand to include it in the tree
- Kruskal: (build forest that will finish as a tree)
  - Pick lower cost edge not yet in a tree that does not create a cycle and expand to include it somewhere in the forest

Prim's algorithm

Starting from empty T, choose a vertex at random and initialize

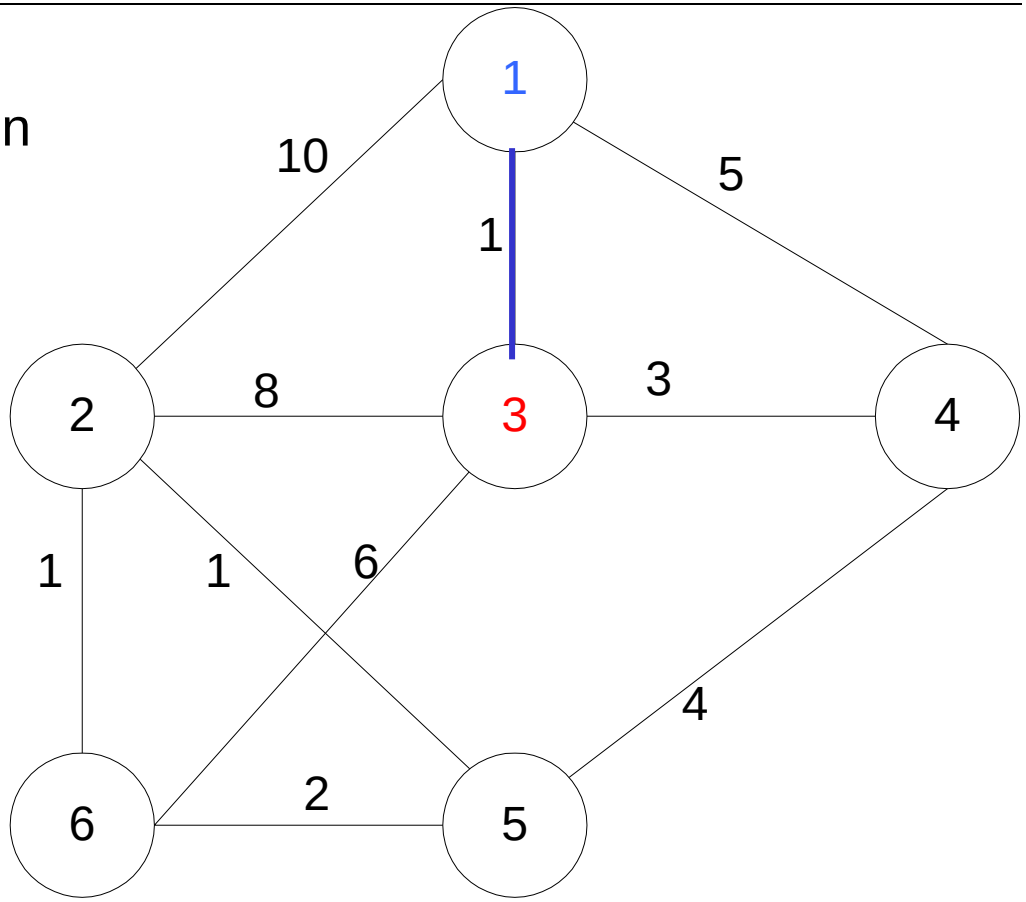
V = {1), E' ={}



Prim's algorithm

Choose the vertex u not in V such that edge weight from u to a vertex in V is minimal (greedy!)

V={1,3} E'= {(1,3) }



Prim's algorithm

Repeat until all vertices have been chosen

Choose the vertex u not in V such that edge weight from v to a vertex in V is minimal (greedy!)

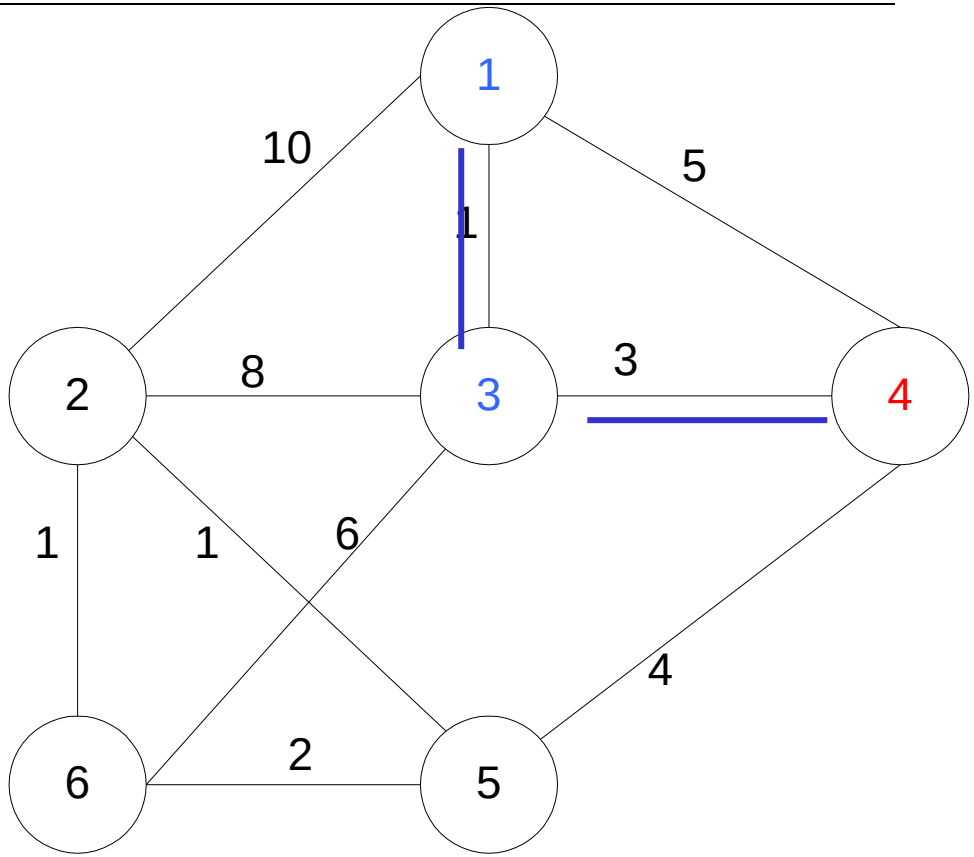
V= {1,3,4} E'= {(1,3),(3,4)}

V={1,3,4,5} E'={(1,3),(3,4),(4,5)}

….

V={1,3,4,5,2,6}

E'={(1,3),(3,4),(4,5),(5,2),(2,6)}



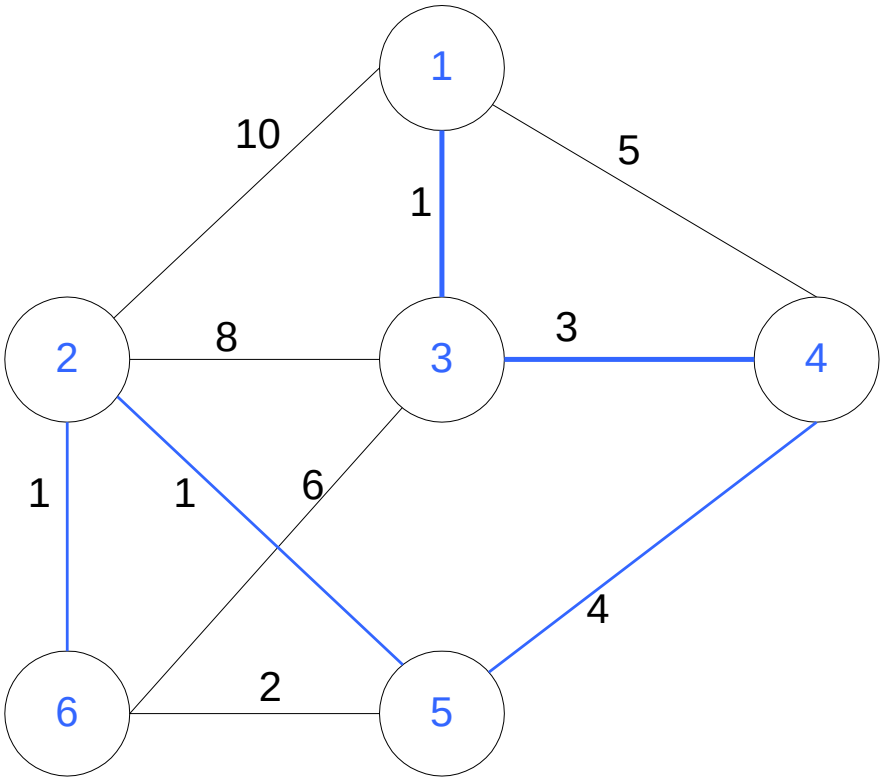
Prim's algorithm

Repeat until all vertices have been chosen

V={1,3,4,5,2,6}

E'={(1,3),(3,4),(4,5),(5,2),(2,6)}

Final Cost: 1 + 3 + 4 + 1 + 1 = 10



Prim's algorithm

Prim's algorithm:

1. Start with any Vertex N in the graph, setting the MST to be N initially.
2. Pick the least-cost edge connected to N and the edge connects N to another vertex; call this M, add Vertex M and Edge (N, M) to the MST.
3. Pick the least-cost edge coming from either N or M to any other vertex in the graph, and add this edge and the new vertex it reaches to the MST.
4. This process continues, at each step expanding the MST by selecting the least-cost edge from a vertex currently in the MST to a vertex not currently in the MST, until all of Vertex in the MST.

Prim's algorithm is quite similar to Dijkstra's algorithm for single source shortest paths. The primary difference is that we are seeking not the next closest vertex to the start vertex, but rather the next closest vertex to any vertex currently in the MST.

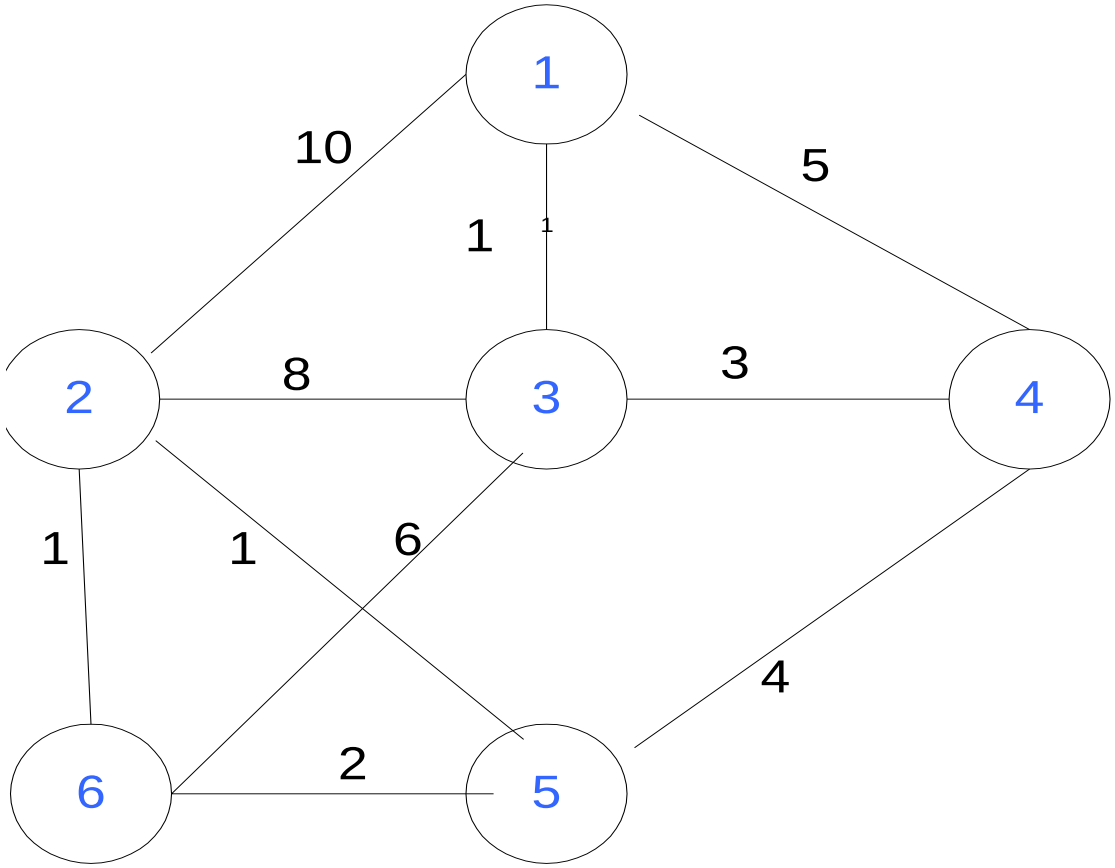
Kruskal's Algorithm

- Select edges in order of increasing cost
- Accept an edge to expand tree or forest only if it does not cause a cycle

Detecting Cycles

- If the edge to be added (u,v) is such that vertices u and v belong to the same tree, then by adding (u,v) you would form a cycle
  - Therefore to check, Find(u) and Find(v). If they are the same discard (u,v)
  - If they are different Union(Find(u),Find(v))

Example

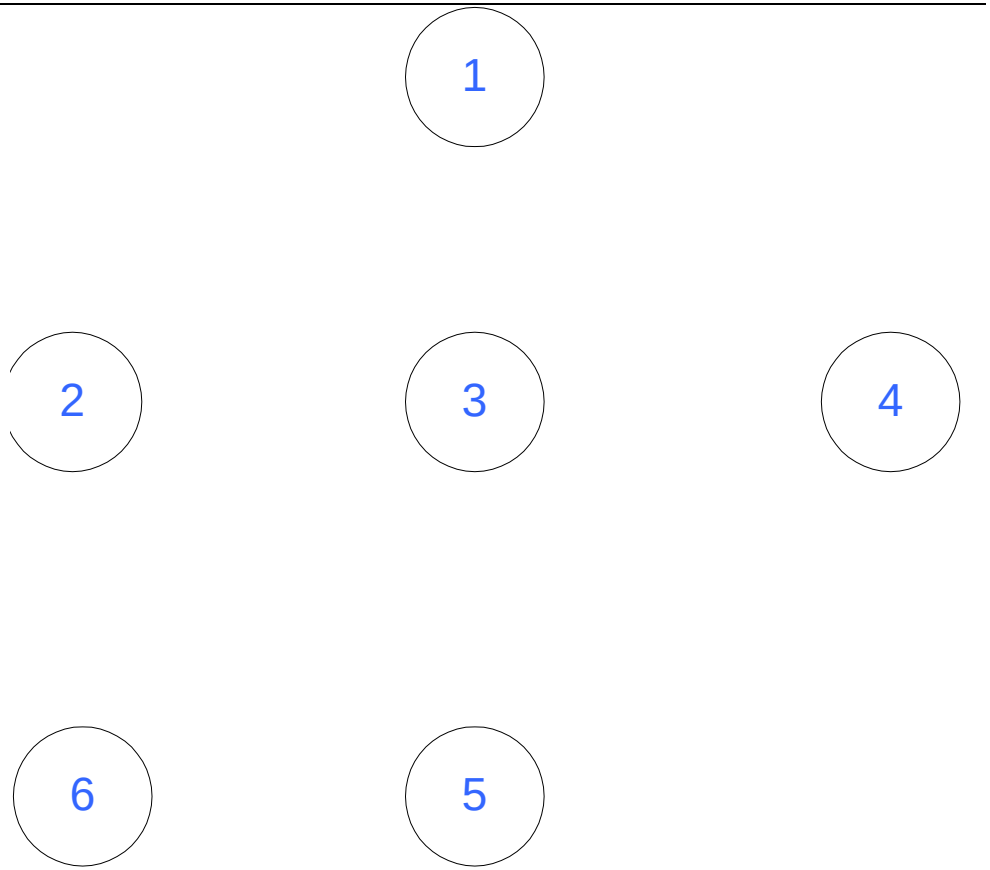


Initialization

Initially, Forest of 6 trees

F= {{1},{2},{3},{4},{5},{6}}

Edges in a heap (not shown)



Step 1

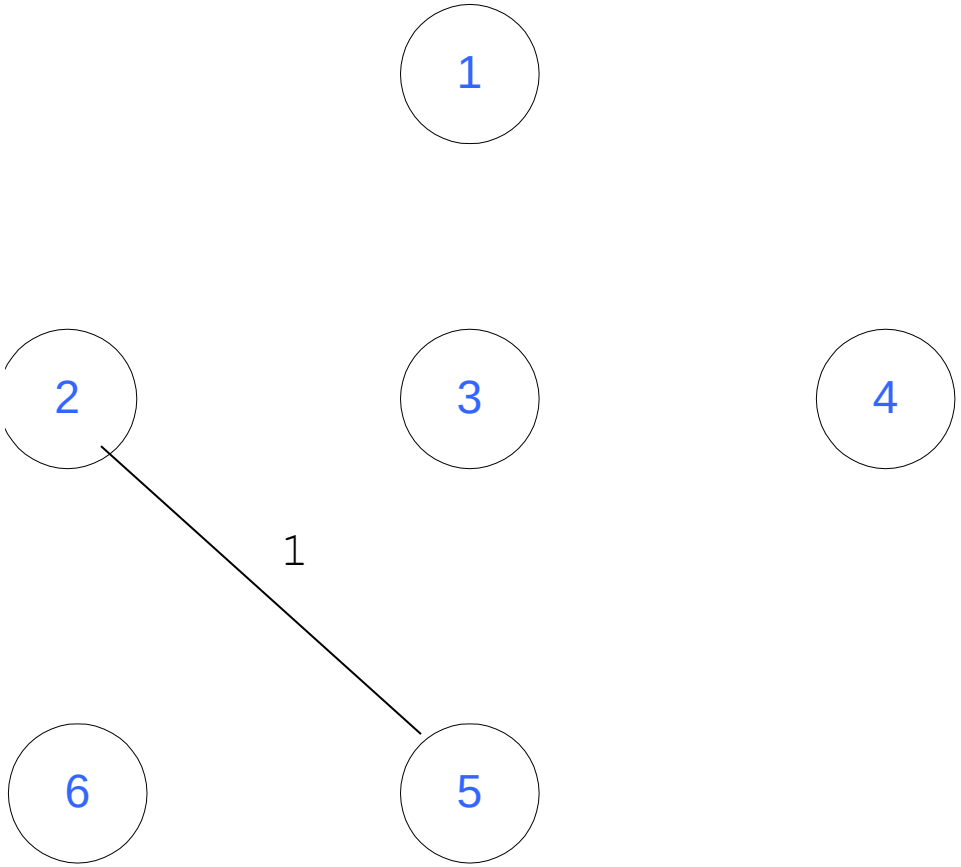
Select edge with lowest cost (2,5)

Find(2) = 2, Find (5) = 5

Union(2,5)

F= {{1},{2,5},{3},{4},{6}}

1 edge accepted



Step 2

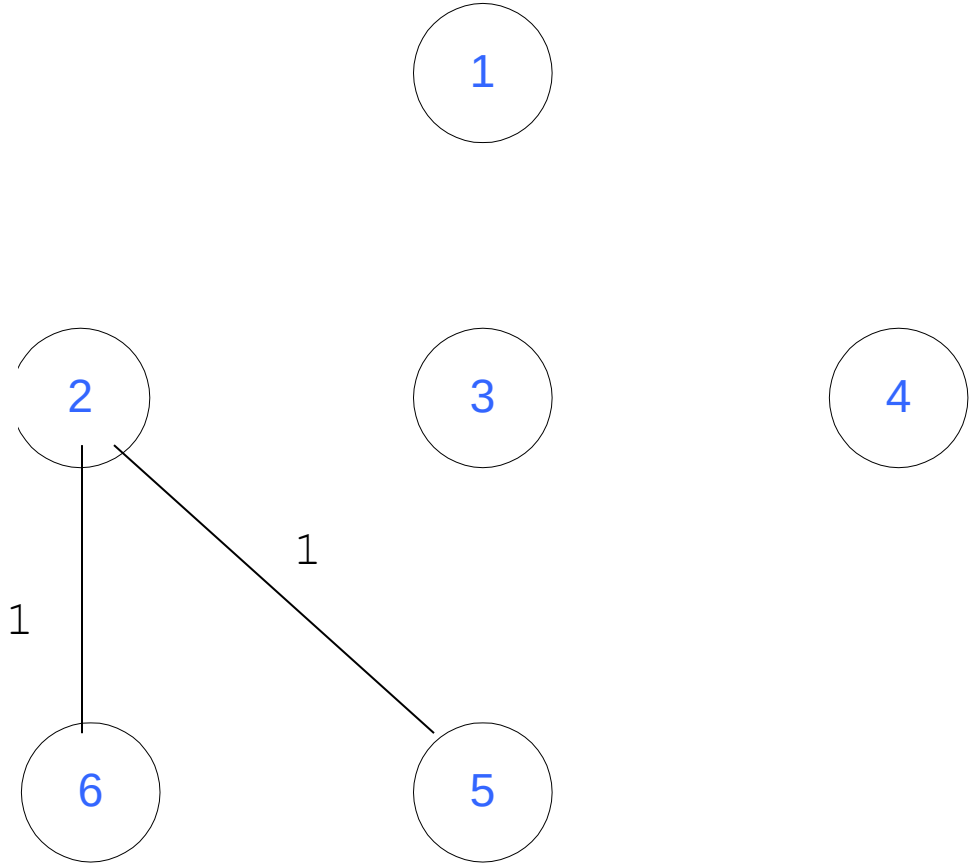
Select edge with lowest cost (2,6)

Find(2) = 2, Find (6) = 6

Union(2,6)

F= {{1},{2,5,6},{3},{4}}

2 edges accepted



Step 3

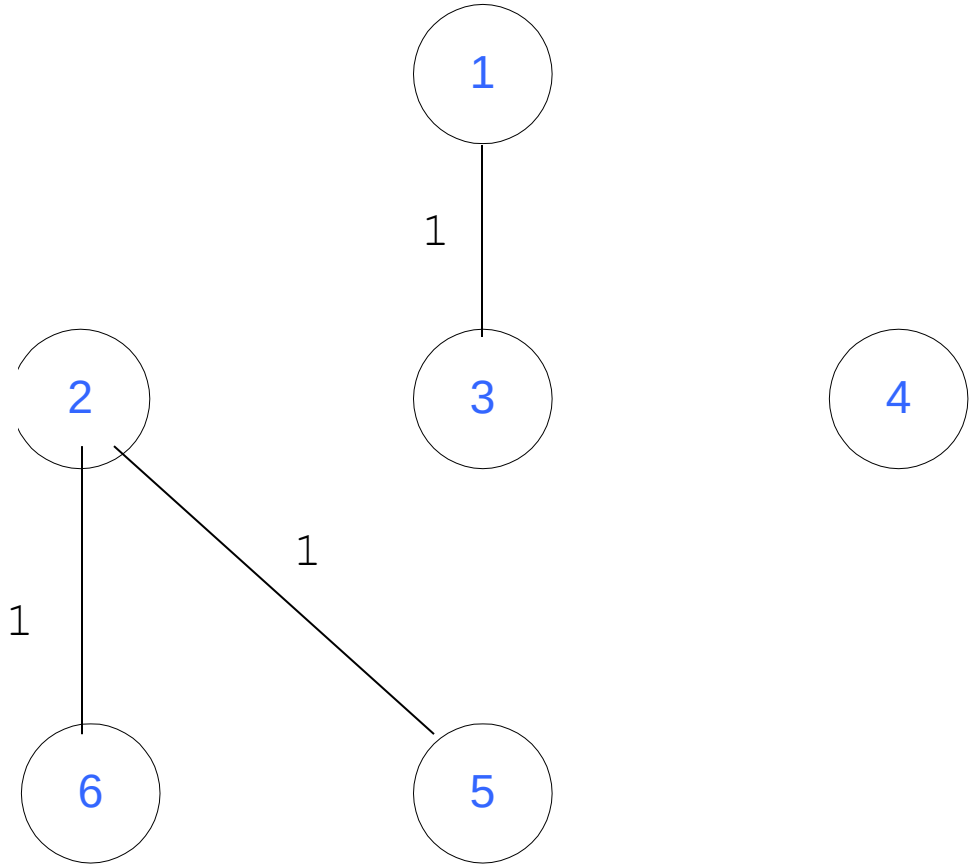
Select edge with lowest cost (1,3)

Find(1) = 1, Find (3) = 3

Union(1,3)

F= {{1,3},{2,5,6},{4}}

3 edges accepted



Step 4

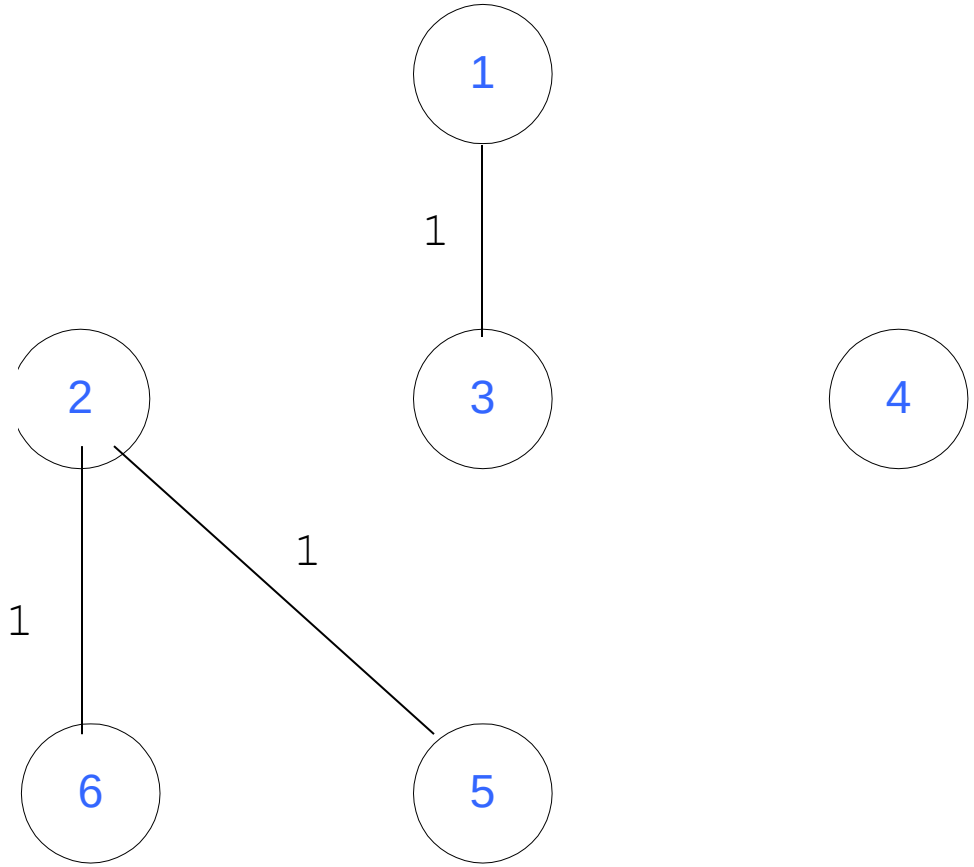
Select edge with lowest cost (5,6)

Find(5) = 2, Find (6) = 2

Do nothing

F= {{1,3},{2,5,6},{4}}

3 edges accepted



Step 5

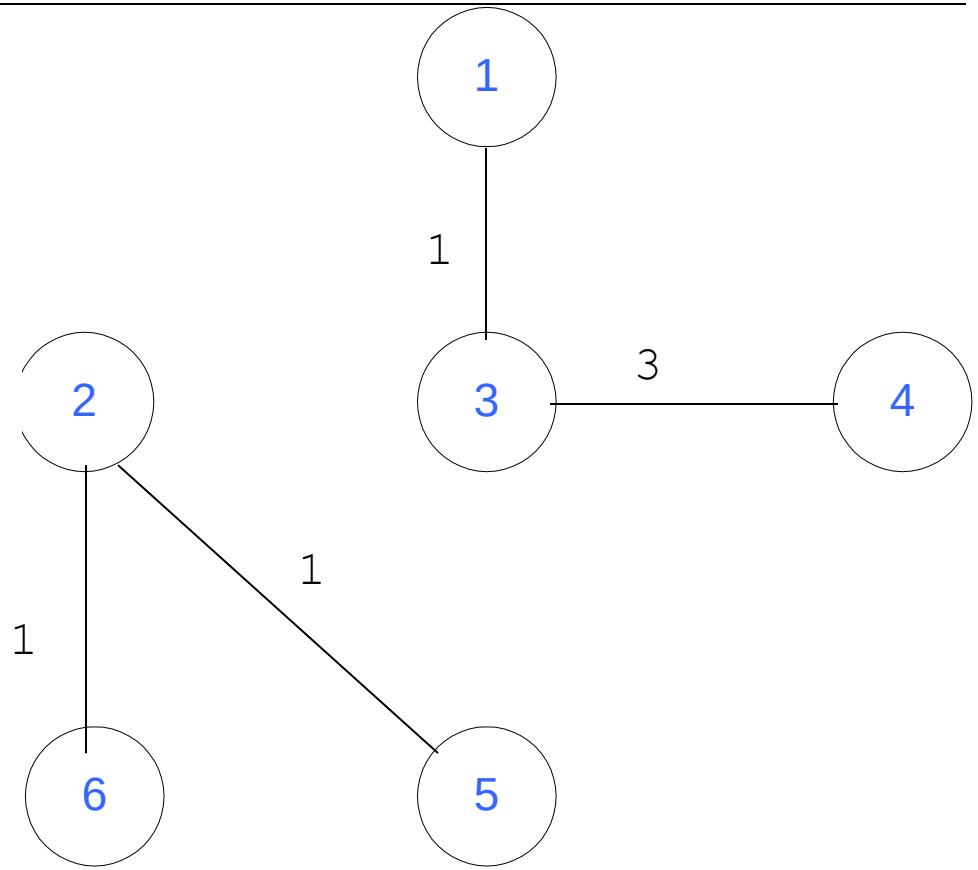
Select edge with lowest cost (3,4)

Find(3) = 1, Find (4) = 4

Union(1,4)

F= {{1,3,4},{2,5,6}}

4 edges accepted



Step 6

Select edge with lowest cost (4,5)

Find(4) = 1, Find (5) = 2

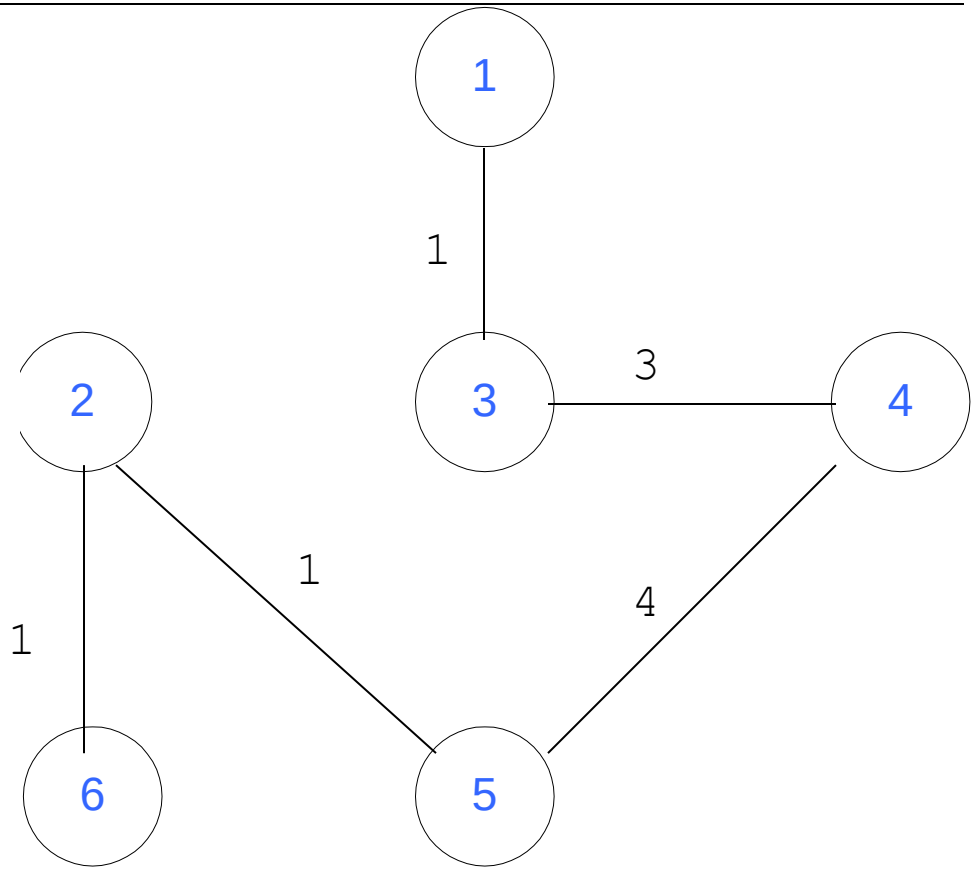
Union(1,2)

F= {{1,3,4,2,5,6}}

5 edges accepted : end

Total cost = 10

Although there is a unique spanning tree in this example, this is not generally the case



Kruskal's Algorithm

Kruskal's algorithm

1. First partition the set of vertices into |V| equivalence classes， each consisting of one vertex.
2. Then process the edges in order of weight.
3. An edge is added to the MST, and two equivalence classes combined, if the edge connects two vertices in different equivalence classes.
4. This process is repeated until only one equivalence class remains.