04-630
Data Structures and Algorithms for Engineers

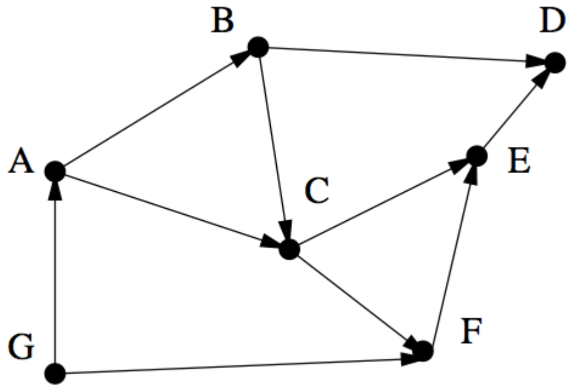
Lecture 18: Graph Algorithms

Previous

- Graphs basics
- Applications
- Traversal
    - BFS
    - DFS

Outline

- DAGs and Topological sorting
- Minimum spanning tree
    - Prims
    - Kruskall
- Shortest path algorithms
    - Dijkstras,
    - Floyds



Topological sorting

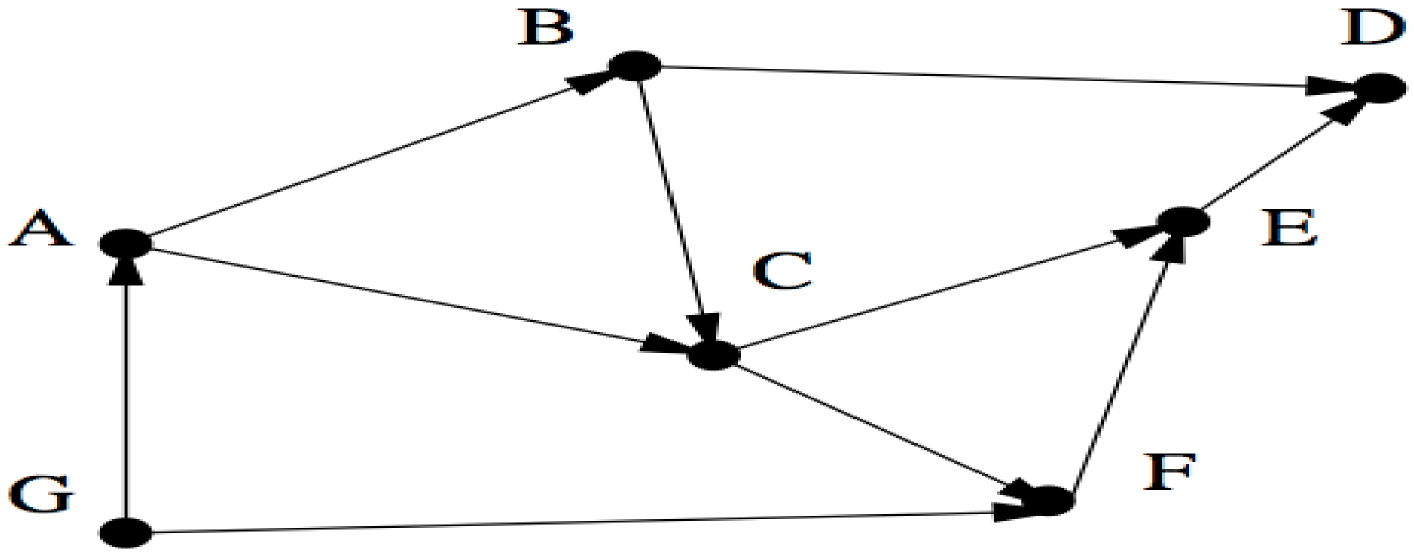
DAG & Topological sorting

- Directed acyclic graph (DAG): directed graph with no cycles.
- Can denote precedence among nodes.
- Using ***topological sorting***, we can obtain a ***total order***.
- Topological sorting:
  - involves sorting a DAG
  - Label the vertices in the ***reverse order*** in which they are ***processed*** (completed) to find the topological sort of a DAG
- ***Definition***: A topological sort of a DAG is a linear ordering of all its vertices such that for any edge (u,v) in the DAG, u appears before v in the ordering

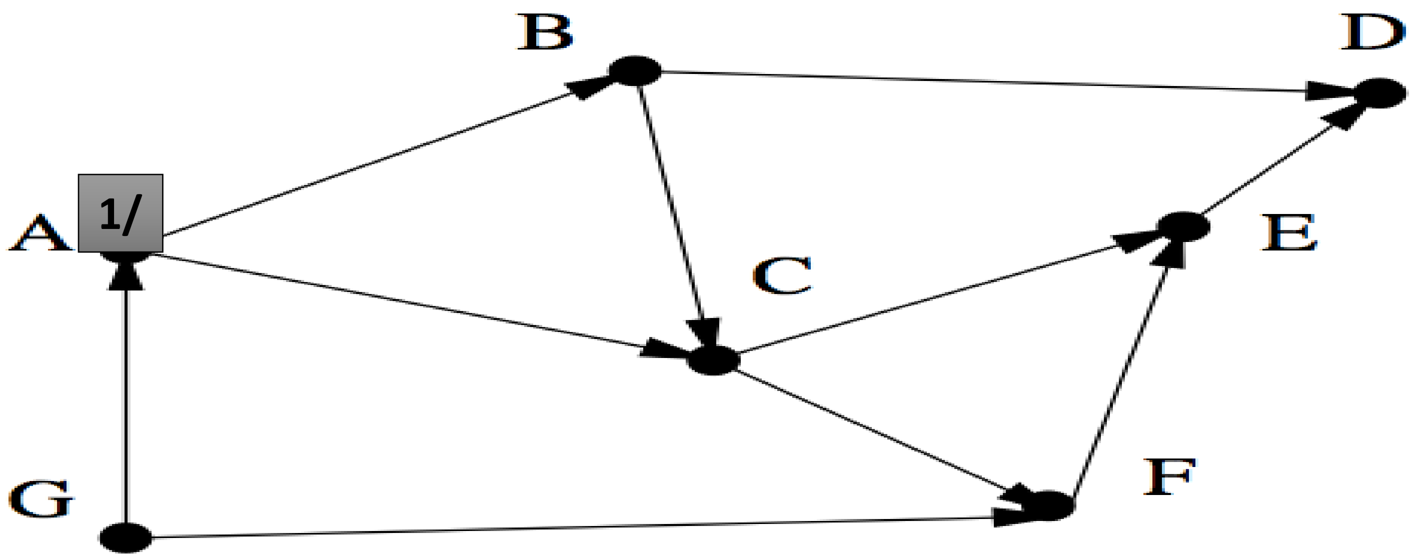
Topological sorting: algorithm

- TopologicalSort(G)
  - Execute DFS(G) to compute v.endtime for each vertex v
  - As each vertex is finished, insert it at the beginning of a linked list (*or insert it on the stack*)
  - Return the linked list *(or stack)* of vertices

Topological sorting: worked example

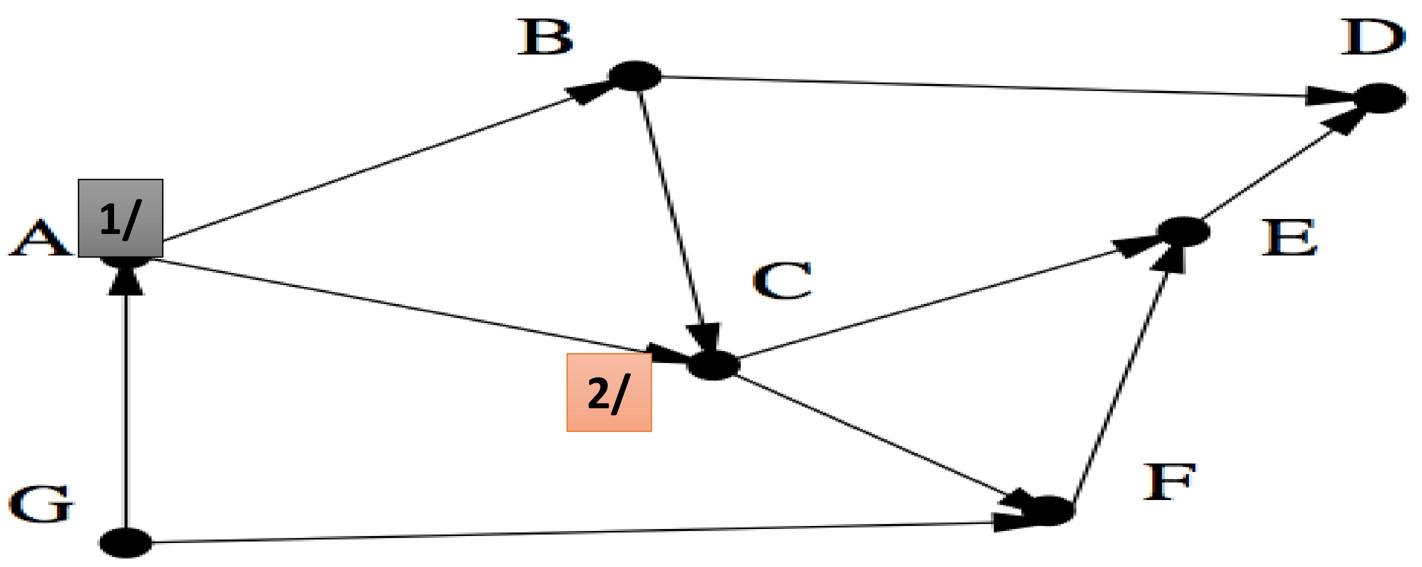


Topological sorting: worked example(1/14)



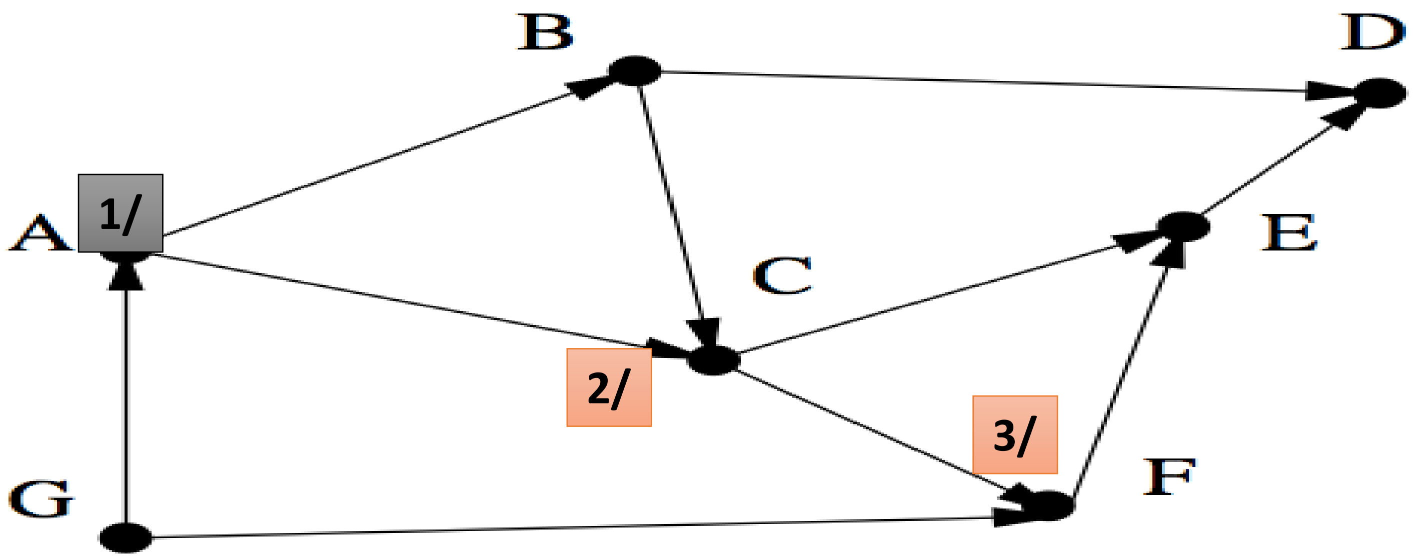
Stack

Topological sorting: worked example (2/14)



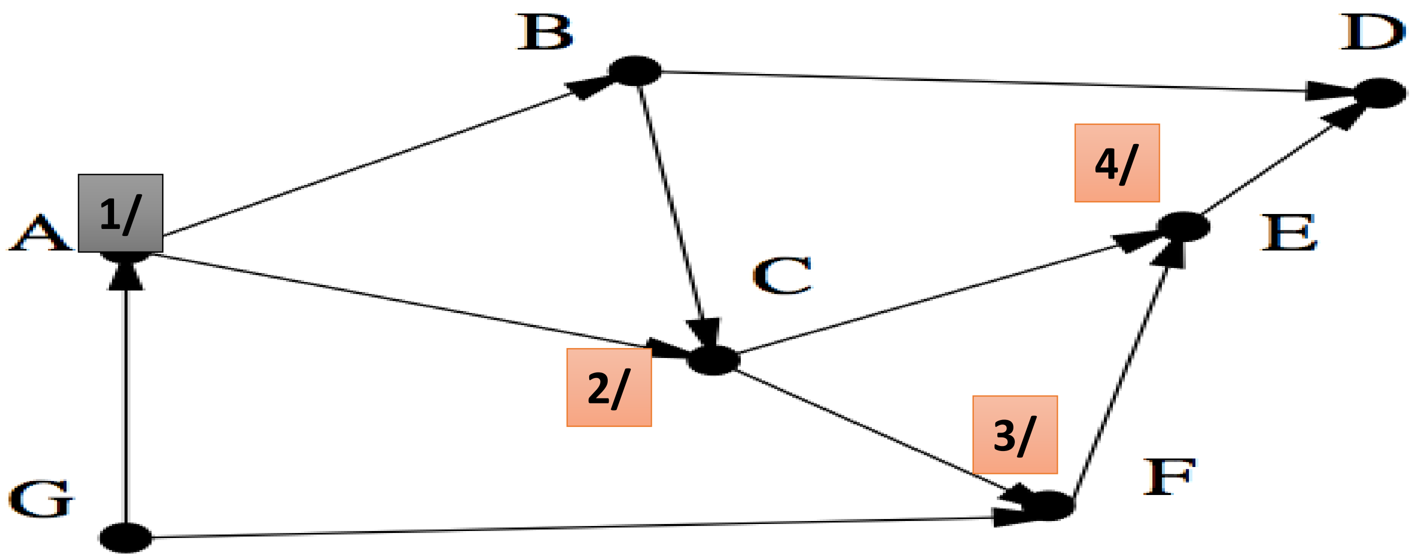
Stack

Topological sorting: worked example (3/14)



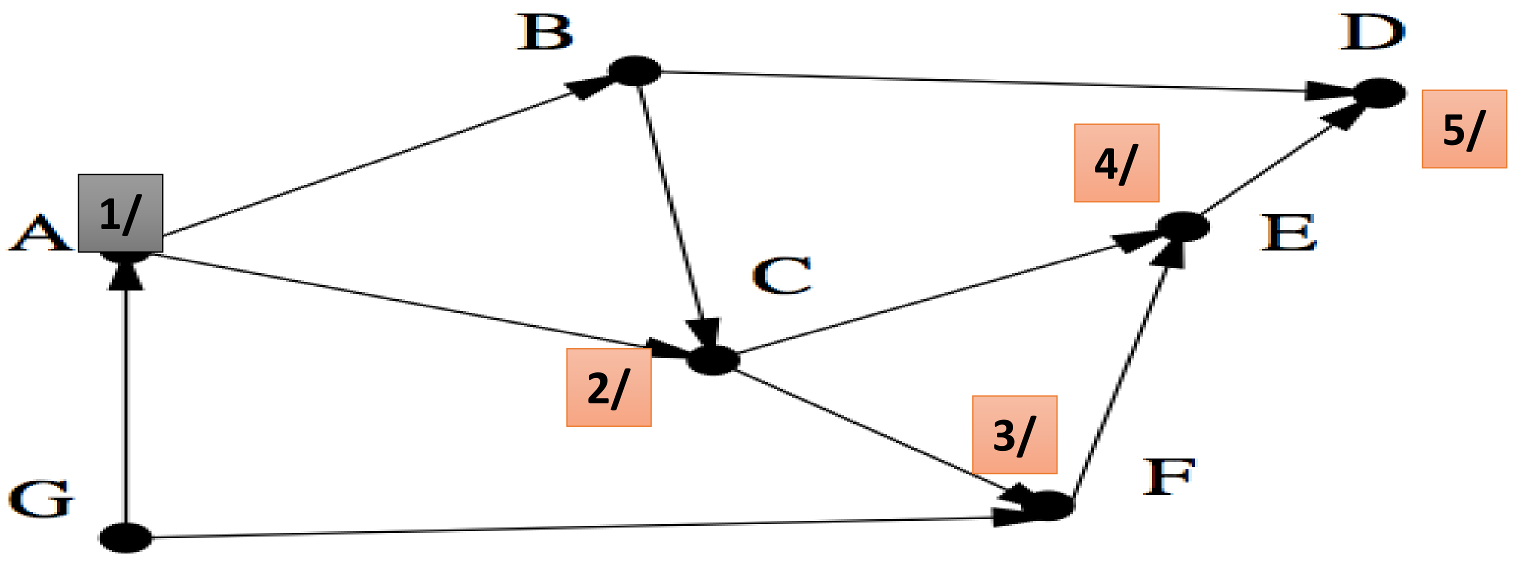
Stack

Topological sorting: worked example (4/14)



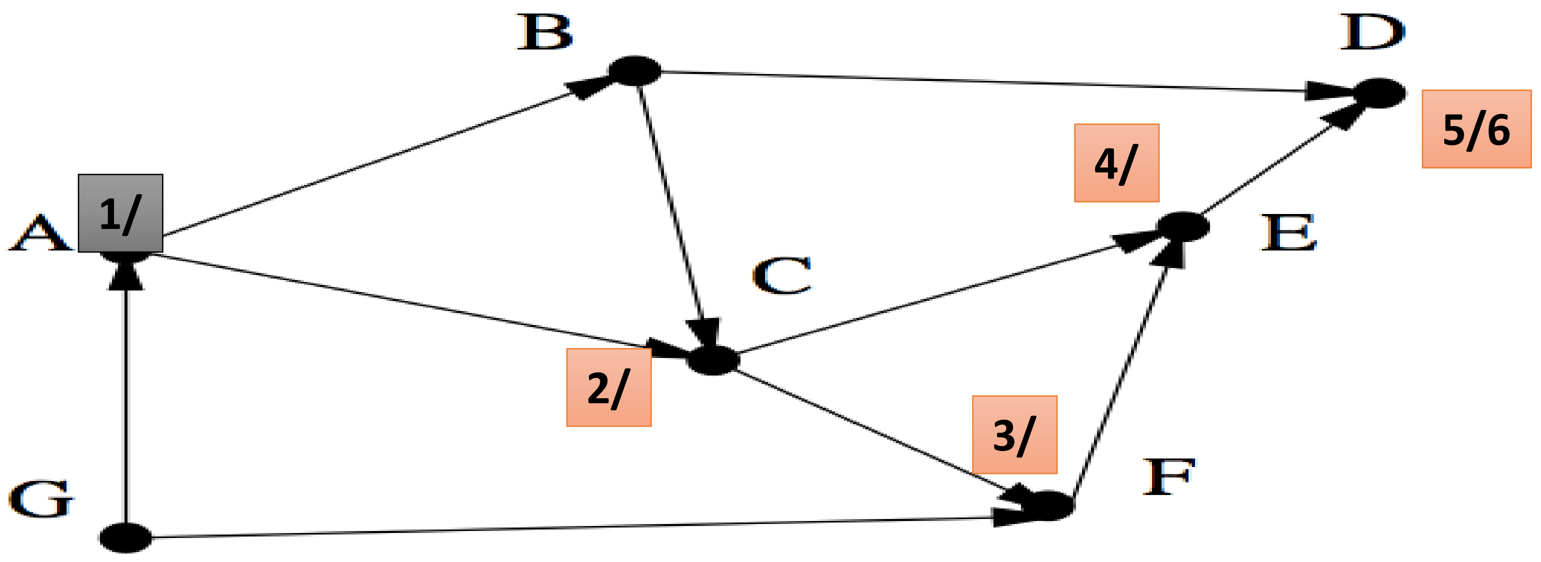
Stack

Topological sorting: worked example (5/14)



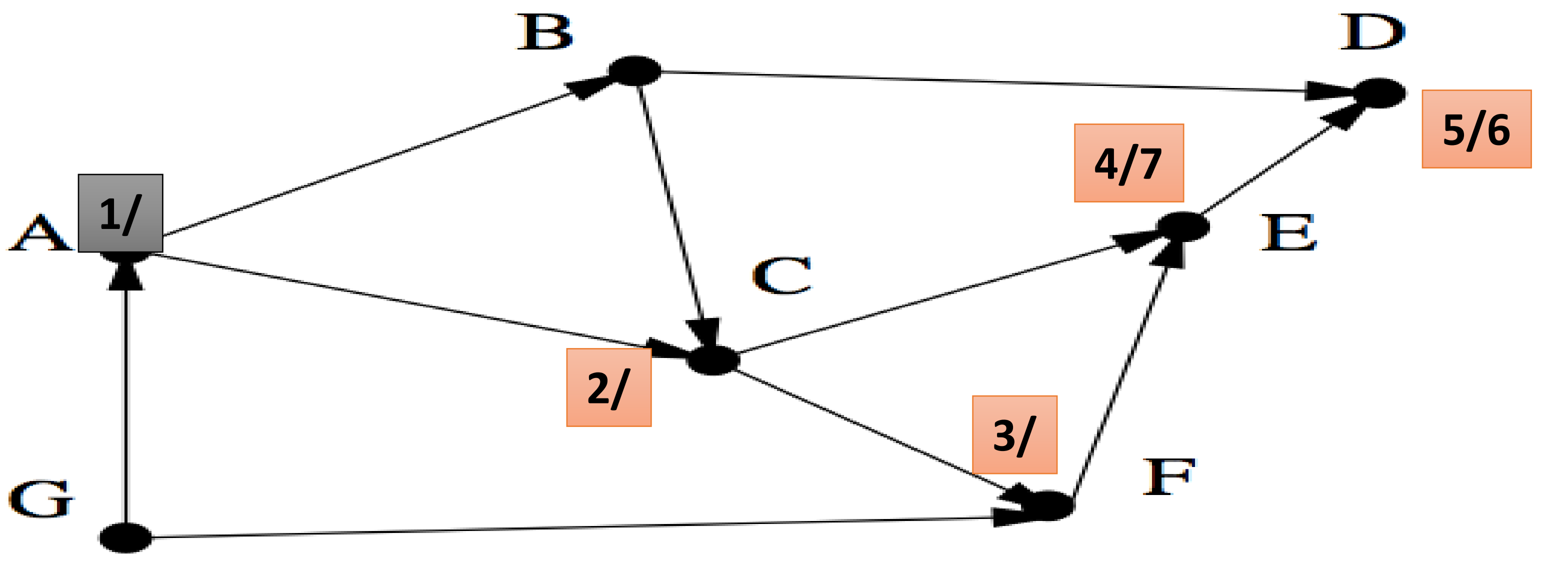
Stack

Topological sorting: worked example (6/14)



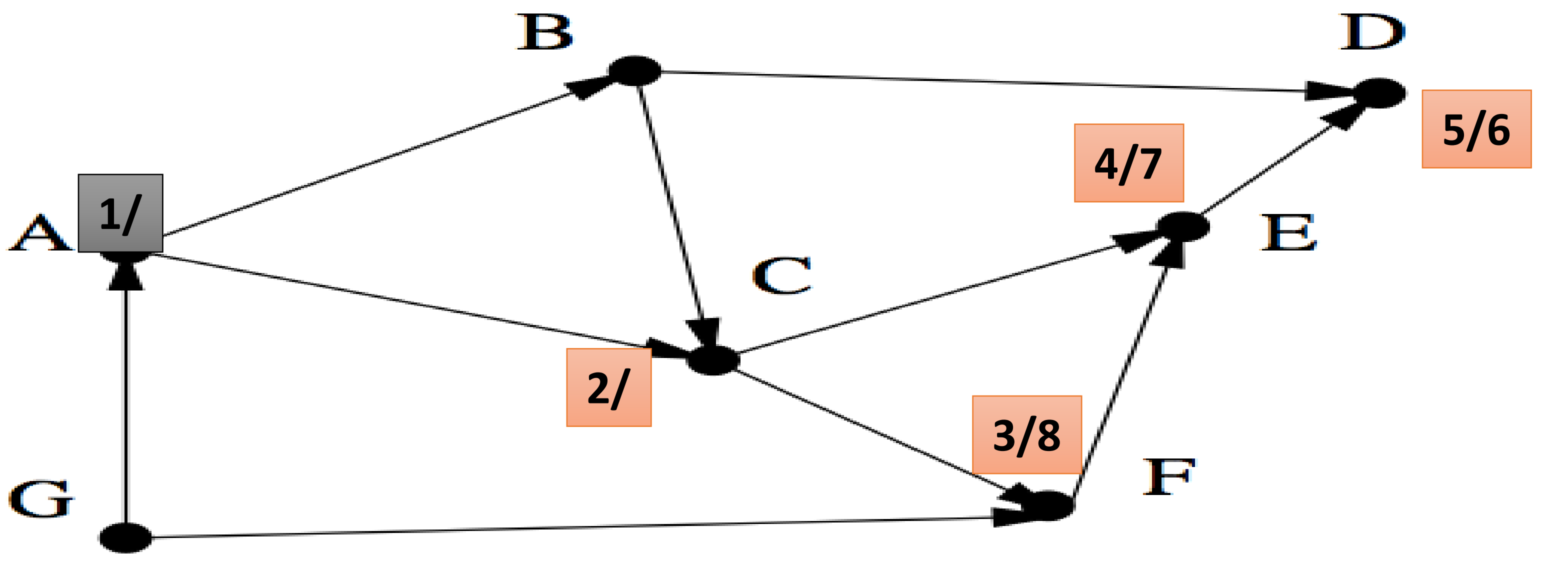
Stack
D

Topological sorting: worked example (7/14)



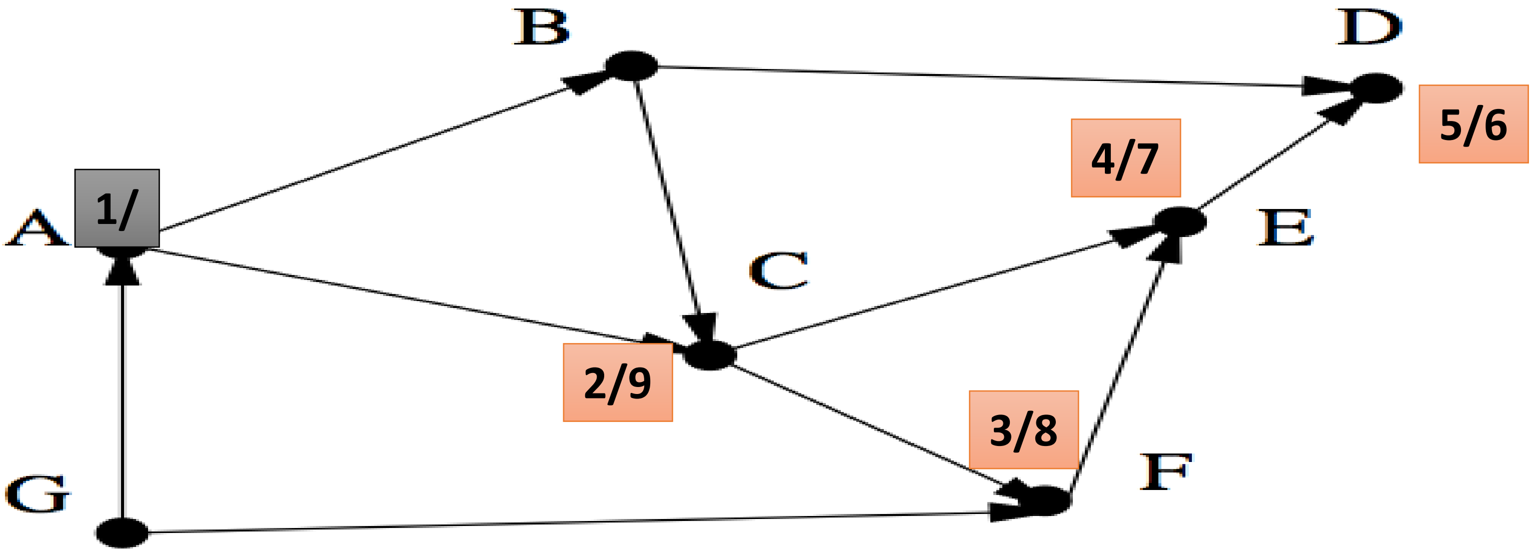
Stack
E
D

Topological sorting: worked example (8/14)



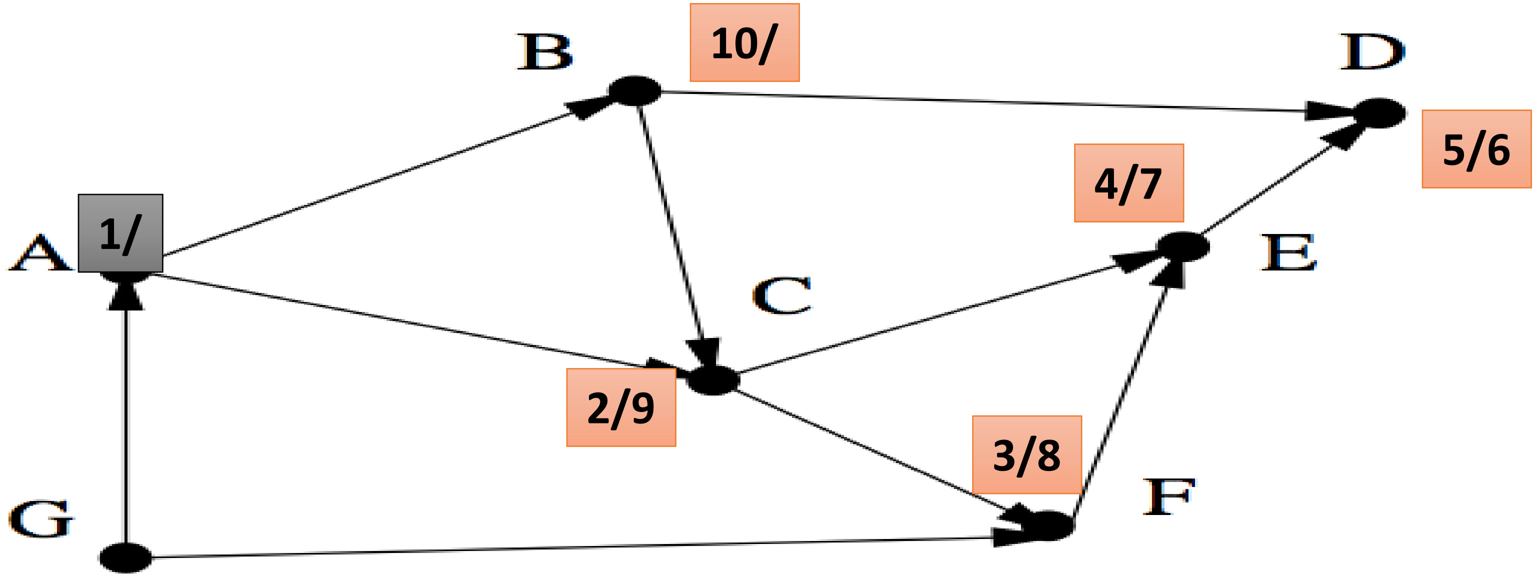
Stack
F
E
D

Topological sorting: worked example (9/14)



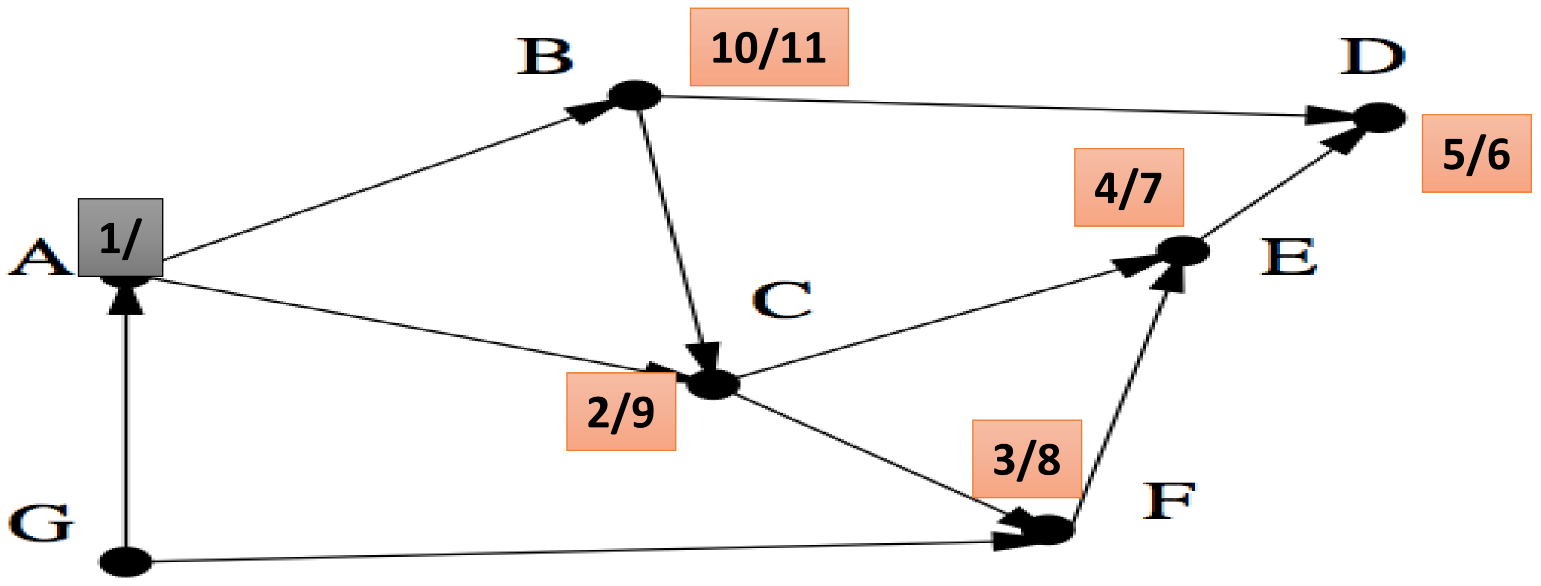
Stack
C
F
E
D

Topological sorting: worked example (10/14)



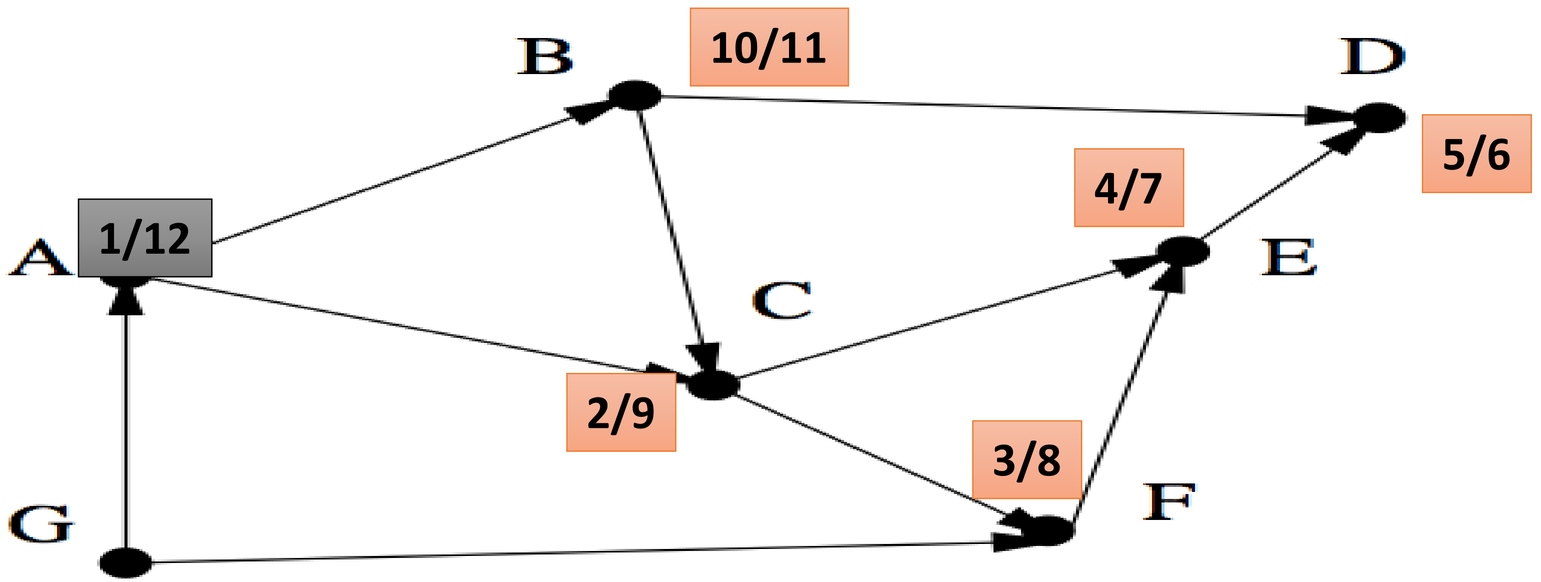
Stack
C
F
E
D

Topological sorting: worked example (11/14)



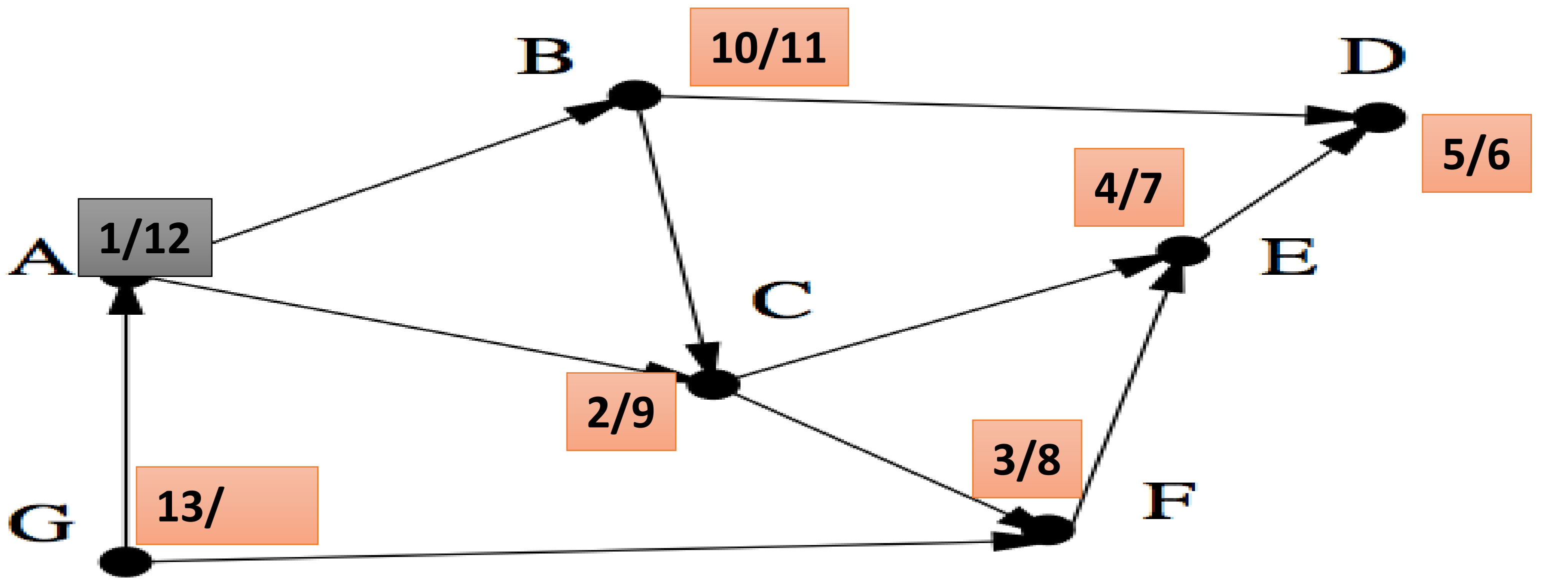
Stack
B
C
F
E
D

Topological sorting: worked example (12/14)



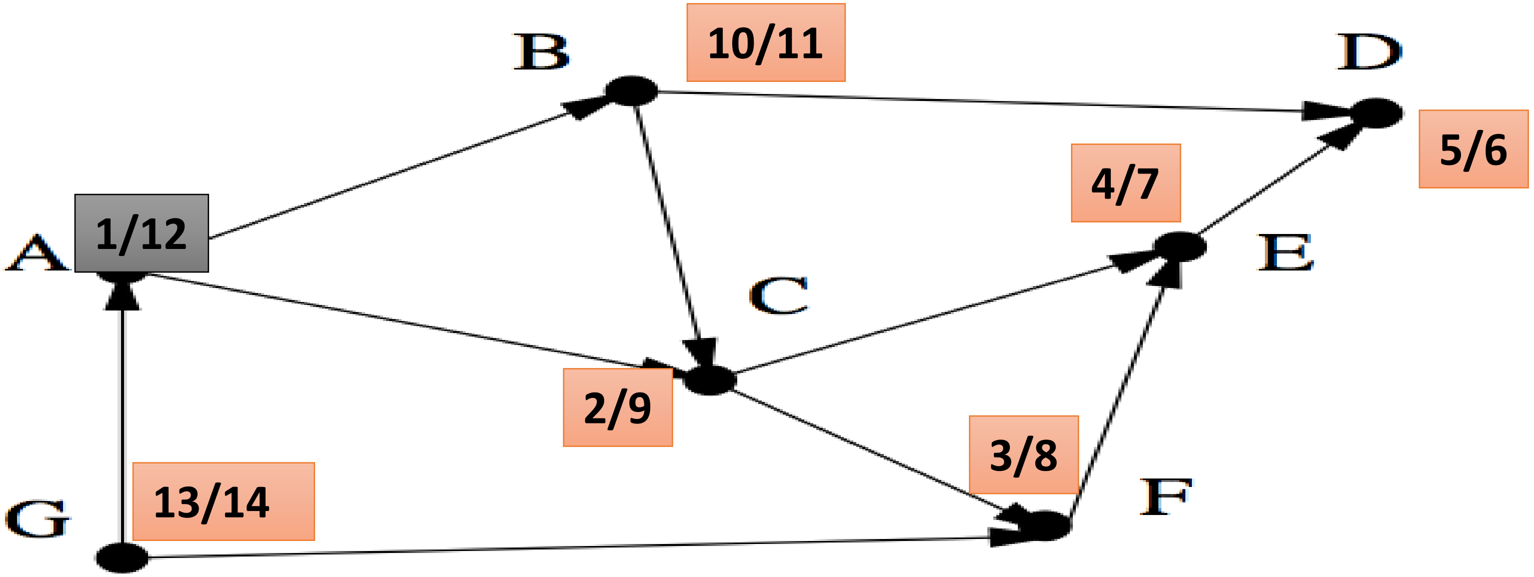
Stack
A
B
C
F
E
D

Topological sorting: worked example (13/14)



Stack
A
B
C
F
E
D

Topological sorting: worked example (14/14)



Stack
G
A
B
C
F
E
D

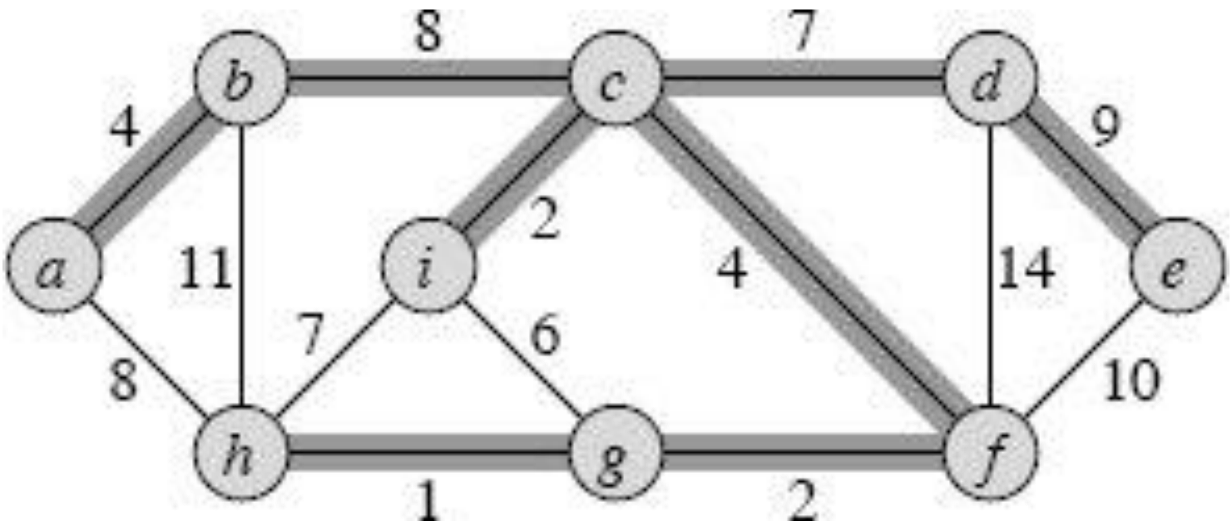
Topological order: G, A, B, C, F, E, D

Topological sorting: applications

- In applications where precedence ordering is needed, e.g.:
  - Dressing up
  - Preparing a recipe
  - Choosing courses (based on prerequisites).
  - Scheduling jobs or tasks where there are dependencies among jobs or tasks.
- See more examples.

Quiz

- Comment on the performance of topological sorting algorithm.

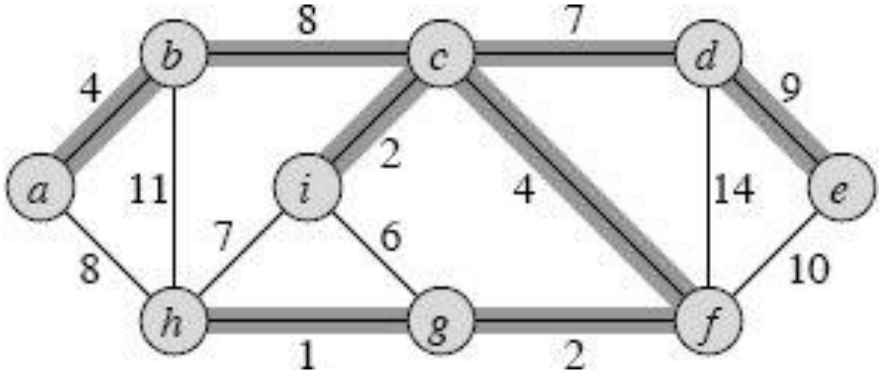


Minimum Spanning Tree

Prims, Kruskall

MST

- Spanning forest
  - If a graph is not connected, then there is a spanning tree for each connected component of the graph



Spanning Tree

- A tree which contains all the vertices of the graph
- Given (a connected) graph G(V,E), a spanning tree T(V',E'):
  - Is a subgraph of G; such that, $V' \subseteq V$, $E' \subseteq E$, and $V' = V$
  - T forms a tree (i.e., no cycle); and
  - $|E'|=|V| -1$ edges

Minimum Spanning Tree

- Minimum Spanning Tree
  - Spanning tree with the **minimum sum of weights.**
  - There may be more than one MST for a graph.
- Given weighted edges:
  - find the minimum cost spanning tree
- Process:
  - Add an edge of minimum cost that does not create a cycle (greedy algorithm)
  - Repeat **|V| -1** times

MST: definitions

- ***Definition***: A **cut** (S, V −S) of an undirected graph is a partition of the set of vertices into the sets S and V − S.
- ***Definition***: A cut **respects** a set of edges A if no edge in A crosses the cut. That is, none of the edges have one vertex in S and the other vertex in V − S.
- ***Definition***: An edge is a **light edge** satisfying a property if it has the smallest weight out of all edges that satisfy that property
  - Specifically, an edge is a ***light edge*** crossing a cut if it has the smallest weight out of all edges that cross the cut.

MST: definitions

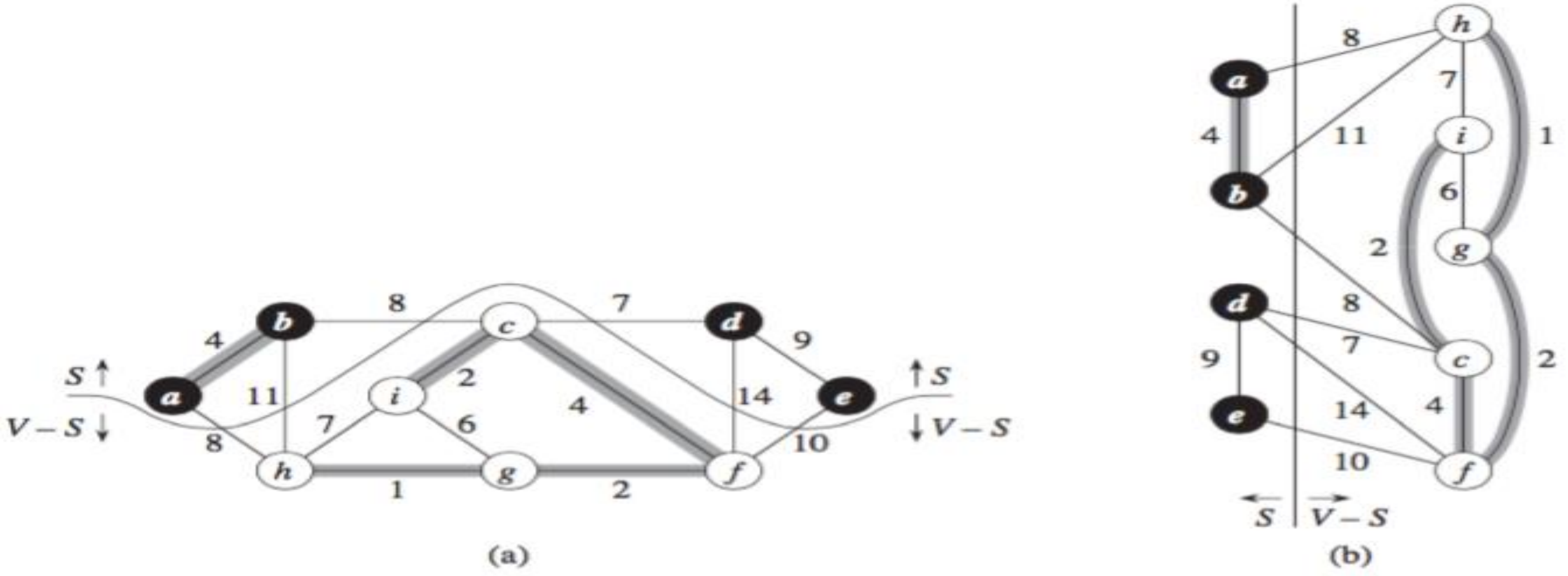
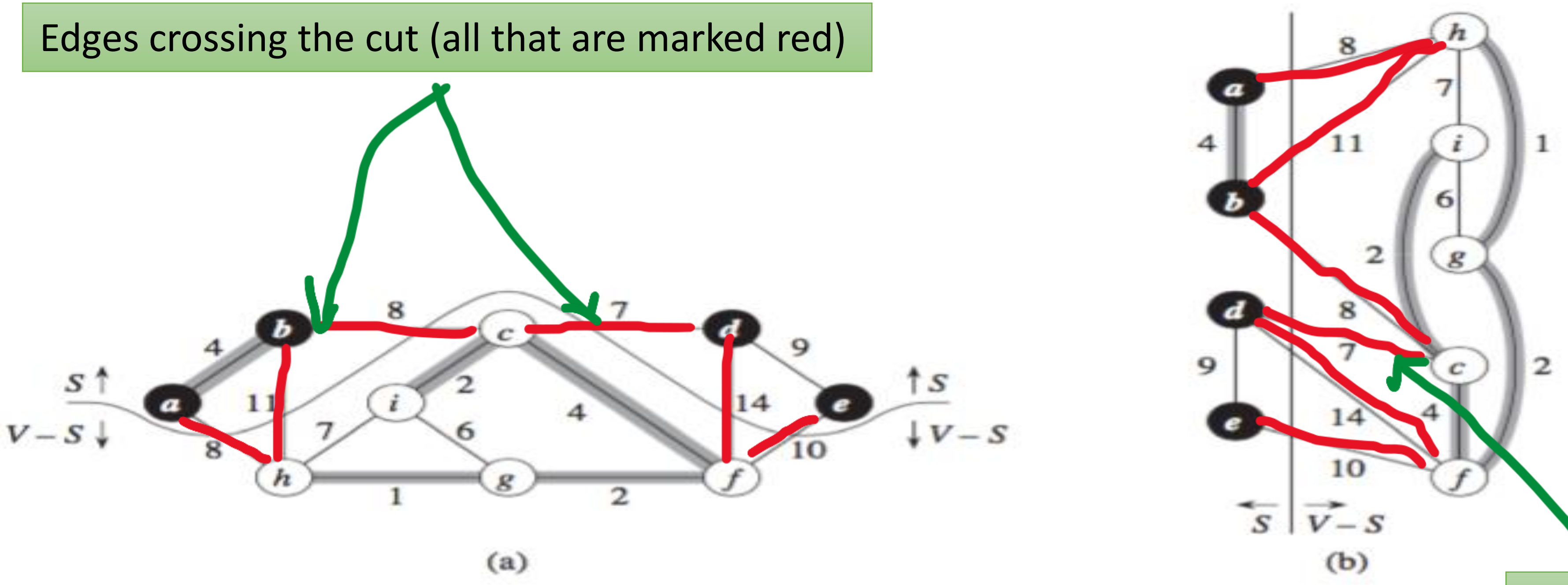


Figure 23.2 Two ways of viewing a cut $(S, V - S)$ of the graph from Figure 23.1. **(a)** Black vertices are in the set S, and white vertices are in $V - S$. The edges crossing the cut are those connecting white vertices with black vertices. The edge (d, c) is the unique light edge crossing the cut. A subset A of the edges is shaded; note that the cut $(S, V - S)$ respects A, since no edge of A crosses the cut. **(b)** The same graph with the vertices in the set S on the left and the vertices in the set $V - S$ on the right. An edge crosses the cut if it connects a vertex on the left with a vertex on the right.

Thomas H. Cormen … [et al. 2009], Introduction to algorithms

MST: definitions(2)

Edges crossing the cut (all that are marked red)



dc: light edge

Figure 23.2 Two ways of viewing a cut $(S, V - S)$ of the graph from Figure 23.1. **(a)** Black vertices are in the set S, and white vertices are in $V - S$. The edges crossing the cut are those connecting white vertices with black vertices. The edge (d, c) is the unique light edge crossing the cut. A subset A of the edges is shaded; note that the cut $(S, V - S)$ respects A, since no edge of A crosses the cut. **(b)** The same graph with the vertices in the set S on the left and the vertices in the set $V - S$ on the right. An edge crosses the cut if it connects a vertex on the left with a vertex on the right.

Thomas H. Cormen … [et al. 2009], Introduction to algorithms

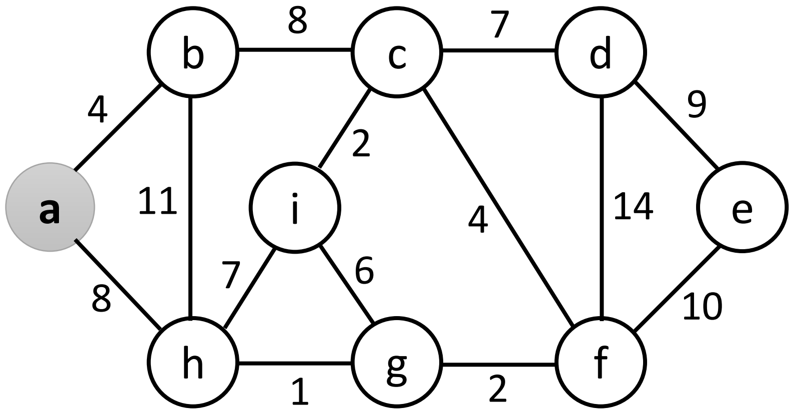
MST Algorithms

- Prim's algorithm:
  - build tree incrementally
- Kruskal's algorithm:
  - build forest that will finish as a tree.

MST: Prim’s Algorithm

- Repeatedly select the smallest weight edge that increases the number of vertices in the tree.
    1. Start from any vertex
    2. Grow the rest of the tree, one edge at a time
    3. Until all vertices are included.

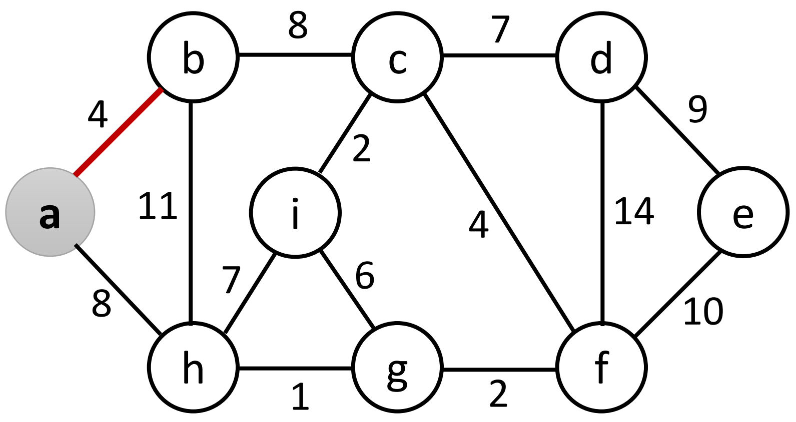
Prim’s Algorithm example



Choose a vertex at random and initialize

e.g. Select a.   Initialize: V={a}, E’={}

Prim’s Algorithm example



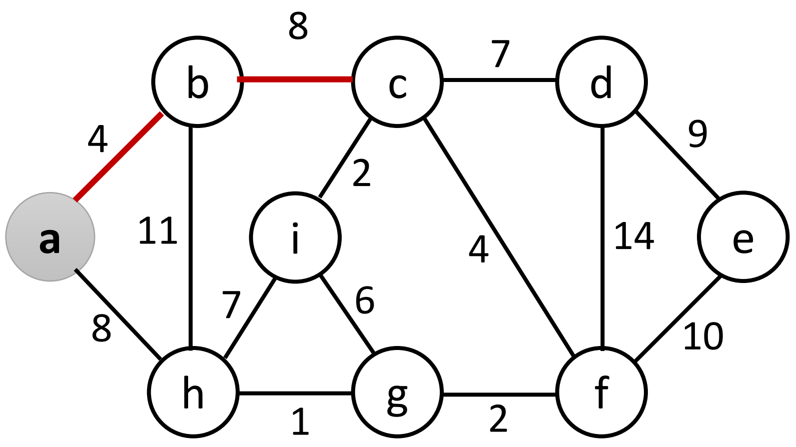
Choose the vertex u not in V’ such that
edge weight from u to a vertex in V’ is minimal,

Choose b.

V’={a,b}

E’={{a,b}}

Prim’s Algorithm example



Repeat until all vertices have been chosen

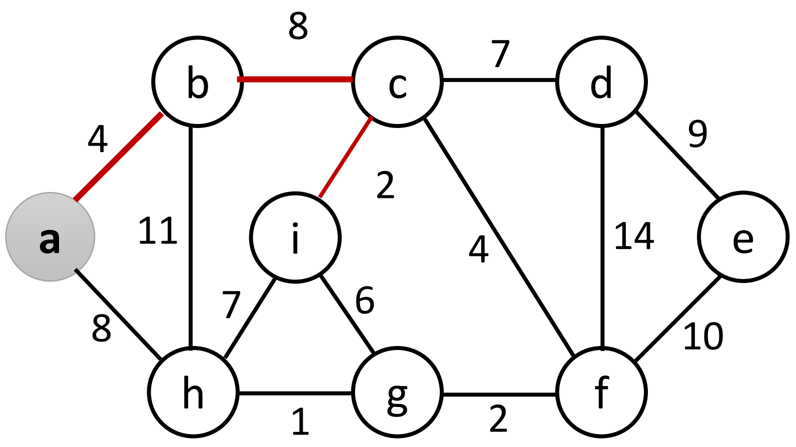
Choose the vertex u not in V’ such that
edge weight from u to a vertex in V’ is minimal}

Choose c.

V’={a,b,c}

E’={(a,b),(b,c)}

Prim's Algorithm example



Repeat until all vertices have been chosen

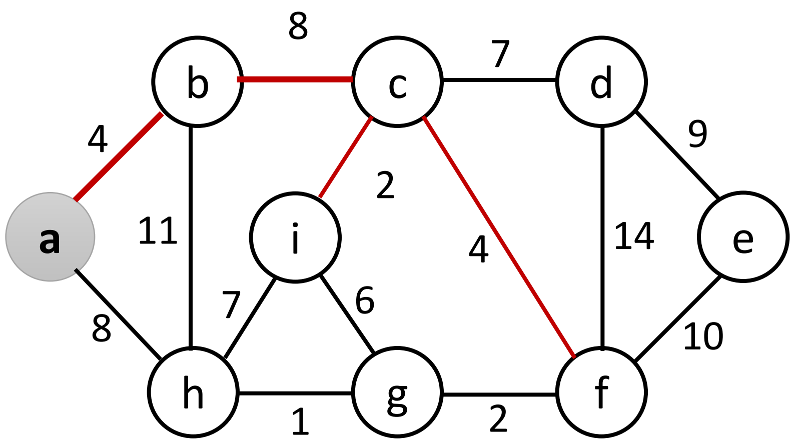
Choose the vertex u not in V' such that
edge weight from u to a vertex in V' is minimal}

Choose i.

V'={a,b,c, i}

E'={(a,b),(b,c), (c,i)}

Prim’s Algorithm example



Repeat until all vertices have been chosen

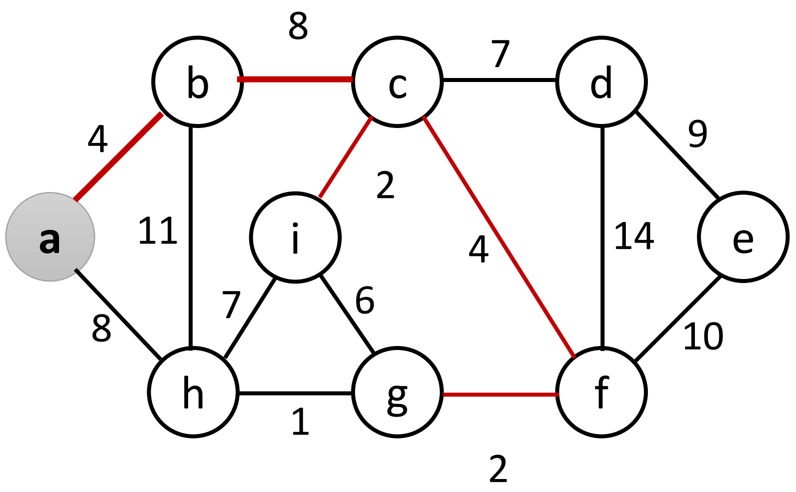
Choose the vertex u not in V’ such that
edge weight from u to a vertex in V’ is minimal}

Choose f.

V’={a,b,c, i, f}

E’={(a,b),(b,c), (c,i), (c,f)}

Prim’s Algorithm example



Repeat until all vertices have been chosen

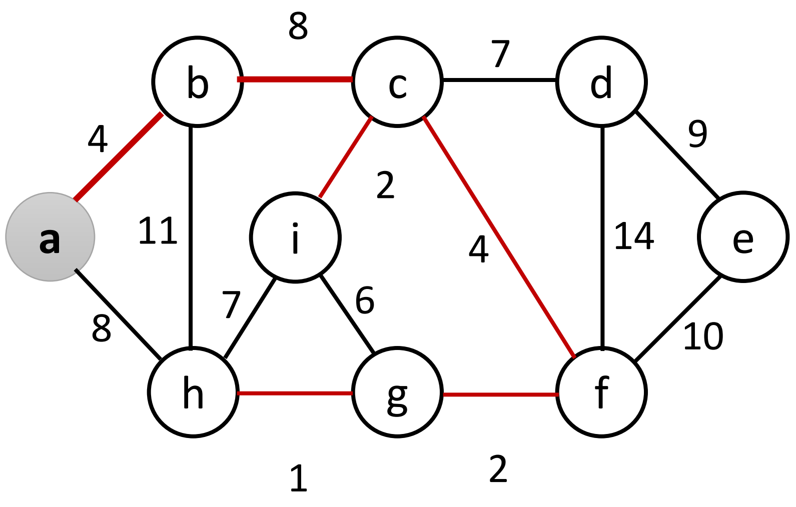
Choose the vertex u not in V’ such that
edge weight from u to a vertex in V’ is minimal}

Choose g.

V’={a,b,c, i, f, g}

E’={(a,b),(b,c), (c,i), (c,f), (f,g)}

Prim's Algorithm example



Repeat until all vertices have been chosen

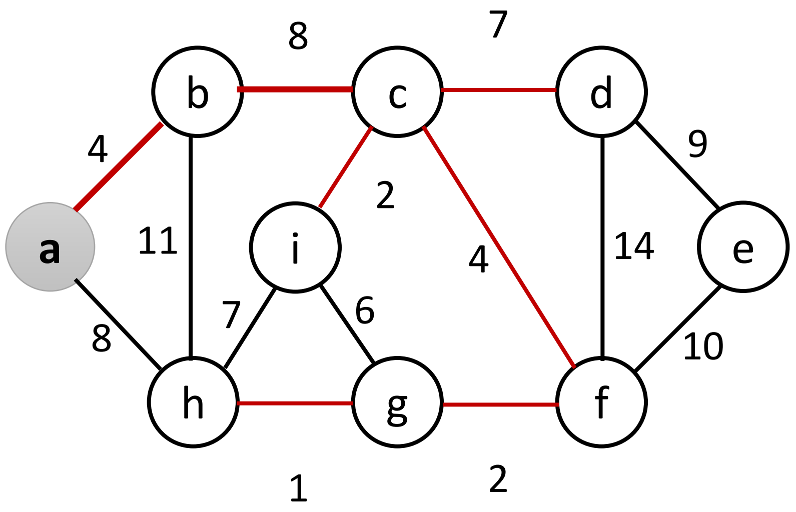
Choose the vertex u not in V' such that
edge weight from u to a vertex in V' is minimal}

Choose h.

V'={a,b,c, i, f, g, h}

E'={(a,b),(b,c), (c,i), (c,f), (f,g), (g,h)}

Prim’s Algorithm example



Repeat until all vertices have been chosen

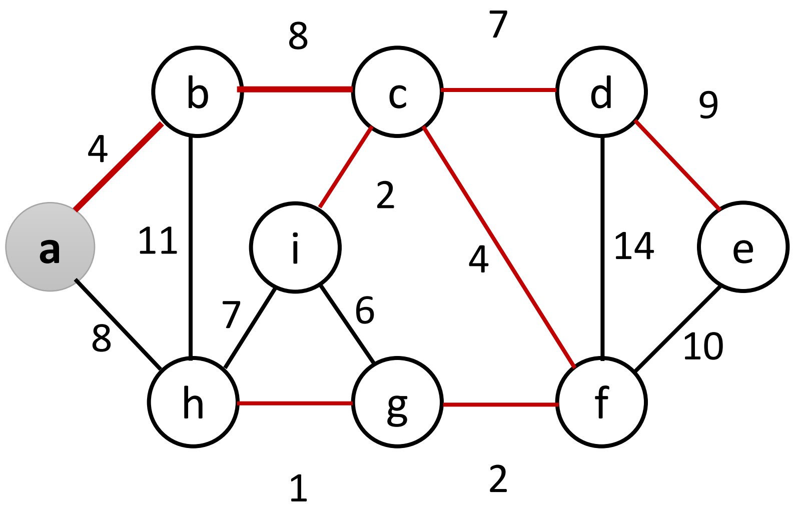
Choose the vertex u not in V’ such that
edge weight from u to a vertex in V’ is minimal}

Choose d.

V’={a,b,c, i, f, g, h, d}

E’={(a,b),(b,c), (c,i), (c,f), (f,g), (g,h), (c,d)}

Prim’s Algorithm example



Repeat until all vertices have been chosen

Choose the vertex u not in V’ such that
edge weight from u to a vertex in V’ is minimal}

Choose e.

V’={a,b,c, i, f, g, h, d, e}

E’={(a,b),(b,c), (c,i), (c,f), (f,g), (g,h), (c,d), (d,e)}

Implementing Prim's algorithm

- Assume adjacency list representation
- Initialize connection weight of each node to infinity.
- Set the predecessor of each node to NIL.
- Unmark all nodes
- Choose one node, say s (start node) and set weight(s) = 0 and prev(s) =0
- While there are unmarked nodes

  Select the unmarked node **u** with minimum weight; mark it

  For each unmarked node **w** adjacent to **u**

      if weight(u,w) < weight(w) then

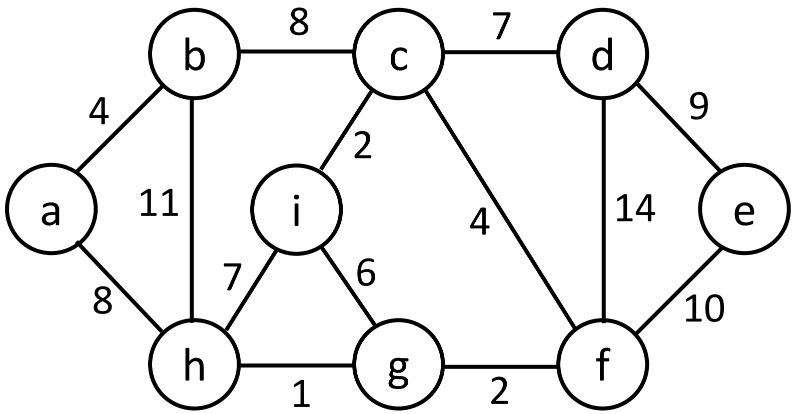
          weight(w) := weight (u,w)

          prev(w)= u

MST: Kruskal’s algorithm

- Start with each vertex being its own component
- Repeatedly merge two components into one by choosing the **light edge** that connects them
- Which components to consider at each iteration?
  - Scan the set of edges by increasing order by weight

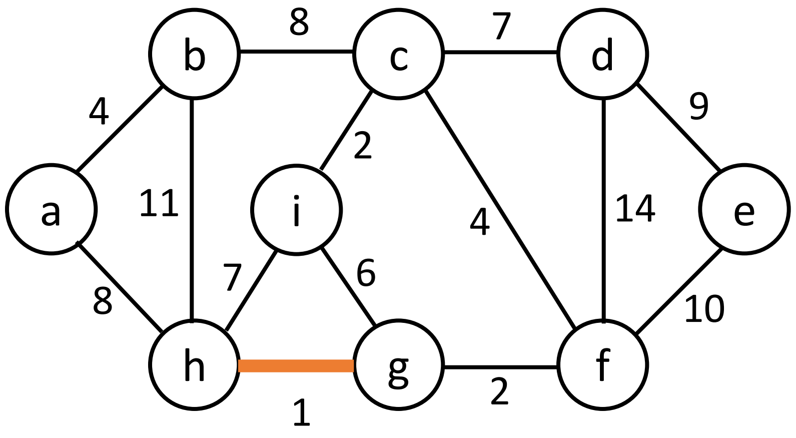
Kruskal's algorithm example



Initial Forest: {a},{b},{c},{d},{e},{f},{g},{h},{i}

Edge	Weight
hg	1
ci	2
gf	2
ab	4
cf	4
gi	6
hi	7
cd	7
bc	8
ah	8
de	9
ef	10
bh	11
df	14

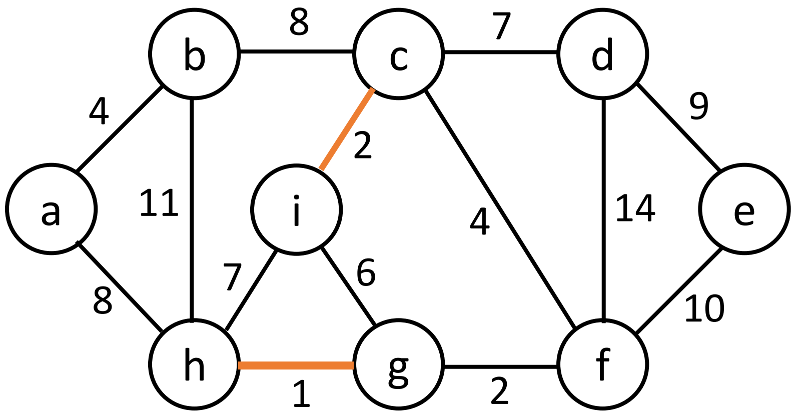
Kruskal's algorithm example



1. Add (h,g): **{g,h}**,{a},{b},{c},{d},{e},{f}, {i}

Edge	Weight
hg	**1**
ci	2
gf	2
ab	4
cf	4
gi	6
hi	7
cd	7
bc	8
ah	8
de	9
ef	10
bh	11
df	14

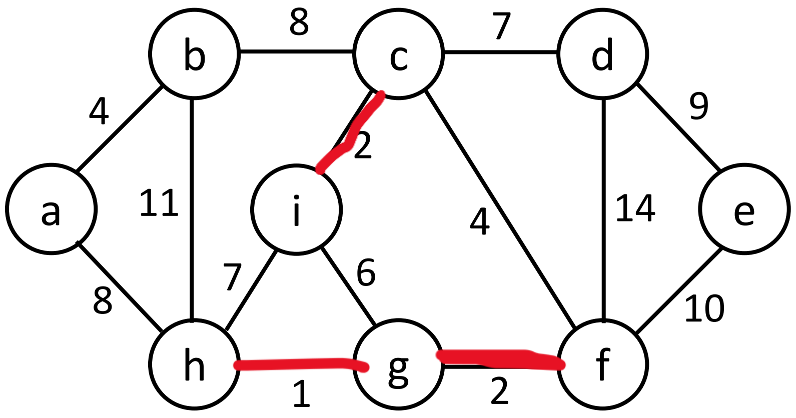
Kruskal's algorithm example



1. Add (h,g): **{g,h}**,{a},{b},{c},{d},{e},{f}, {i}
2. Add (c,i): **{g,h}** ,**{c,i}**,{a},{b}, {d},{e},{f}

Edge	Weight
hg	**1**
ci	**2**
gf	2
ab	4
cf	4
gi	6
hi	7
cd	7
bc	8
ah	8
de	9
ef	10
bh	11
df	14

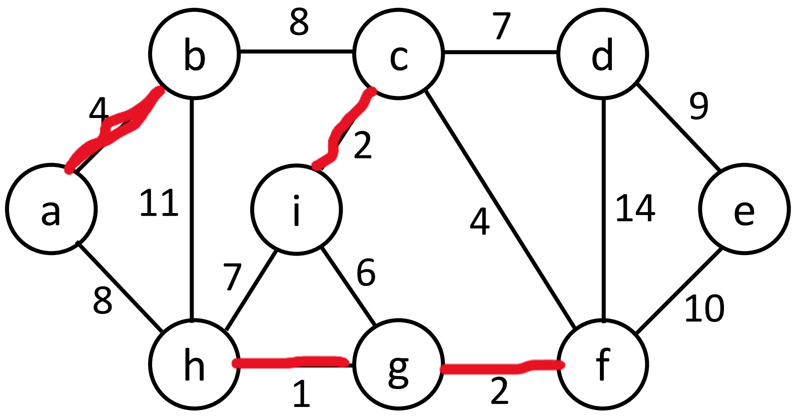
Kruskal’s algorithm example



1. Add (h,g): **{g,h}**,{a},{b},{c},{d},{e},{f}, {i}
2. Add (c,i): **{g,h}** ,**{c,i}**,{a},{b}, {d},{e},{f}
3. Add (g,f): **{g,h,f}, {c,i}**, {a},{b}, {d},{e}

Edge	Weight
hg	**1**
ci	**2**
gf	**2**
ab	4
cf	4
gi	6
hi	7
cd	7
bc	8
ah	8
de	9
ef	10
bh	11
df	14

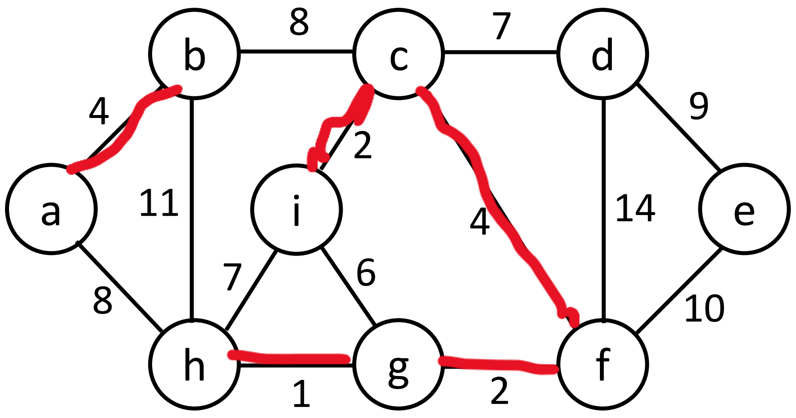
Kruskal's algorithm example



1. Add (h,g): **{g,h}**,{a},{b},{c},{d},{e},{f}, {i}
2. Add (c,i): **{g,h}** ,**{c,i}**,{a},{b}, {d},{e},{f}
3. Add (g,f): **{g,h,f}, {c,i}**, {a},{b}, {d},{e}
4. Add (a,b): **{g,h,f}, {c,i}**, **{a,b}**,{d},{e}

Edge	Weight
hg	**1**
ci	**2**
gf	**2**
ab	**4**
cf	4
gi	6
hi	7
cd	7
bc	8
ah	8
de	9
ef	10
bh	11
df	14

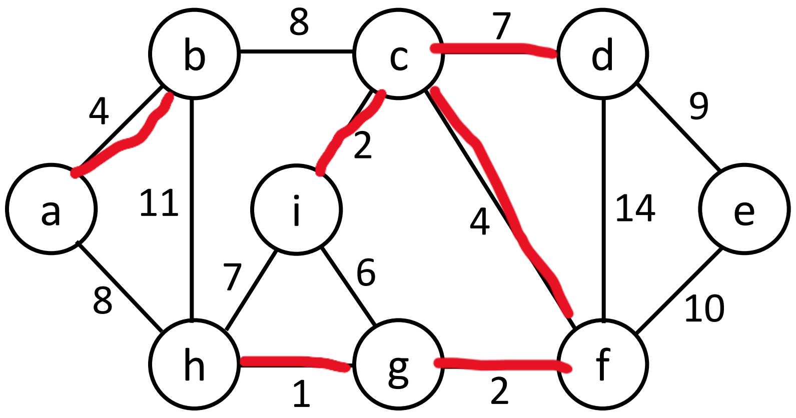
Kruskal's algorithm example



1. Add (h,g): **{g,h}**,{a},{b},{c},{d},{e},{f}, {i}
2. Add (c,i): **{g,h}** ,**{c,i}**,{a},{b}, {d},{e},{f}
3. Add (g,f): **{g,h,f}, {c,i}**, {a},{b}, {d},{e}
4. Add (a,b): **{g,h,f}, {c,i}**, **{a,b}**,{d},{e}
5. Add (c,f): **{g,h,f, c,i}, {a,b}**,{d},{e}

Edge	Weight
hg	**1**
ci	**2**
gf	**2**
ab	**4**
cf	**4**
gi	6
hi	7
cd	7
bc	8
ah	8
de	9
ef	10
bh	11
df	14

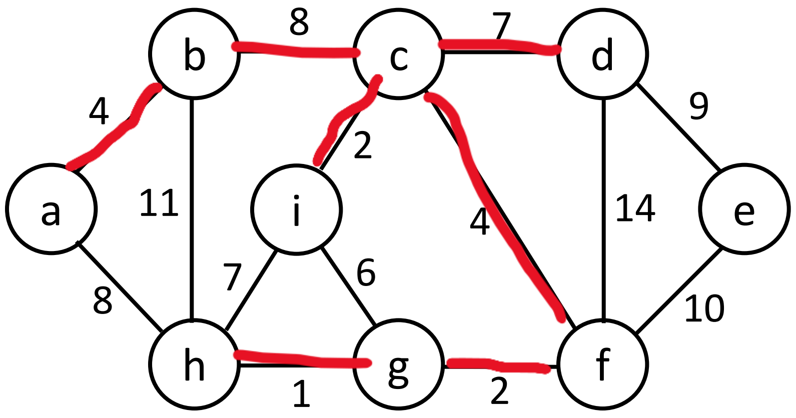
Kruskal's algorithm example



1. Add (h,g): **{g,h}**,{a},{b},{c},{d},{e},{f}, {i}
2. Add (c,i): **{g,h}** ,**{c,i}**,{a},{b}, {d},{e},{f}
3. Add (g,f): **{g,h,f}, {c,i}**, {a},{b}, {d},{e}
4. Add (a,b): **{g,h,f}, {c,i}**, **{a,b}**,{d},{e}
5. Add (c,f): **{g,h,f, c,i}, {a,b}**,{d},{e}
6. Ignore (g,i): why?
7. Ignore (h,i): why?
8. Add (c,d): **{g,h,f, c,i,d}, {a,b}**,{e}

Edge	Weight
hg	**1**
ci	**2**
gf	**2**
ab	**4**
cf	**4**
gi	**6**
hi	**7**
cd	**7**
bc	8
ah	8
de	9
ef	10
bh	11
df	14

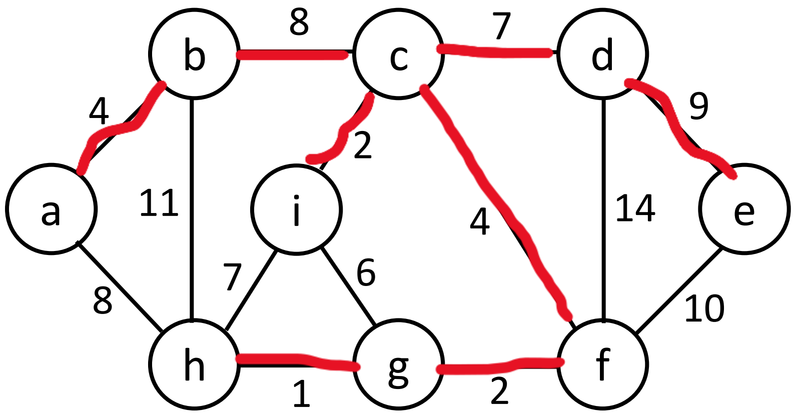
Kruskal's algorithm example



1. Add (h,g): **{g,h}**,{a},{b},{c},{d},{e},{f}, {i}
2. Add (c,i): **{g,h}** ,**{c,i}**,{a},{b}, {d},{e},{f}
3. Add (g,f): **{g,h,f}, {c,i}**, {a},{b}, {d},{e}
4. Add (a,b): **{g,h,f}, {c,i}**, **{a,b}**,{d},{e}
5. Add (c,f): **{g,h,f, c,i}, {a,b}**,{d},{e}
6. Ignore (g,i): why?
7. Ignore (h,i): why?
8. Add (c,d): **{g,h,f, c,i,d}, {a,b}**,{e}
9. Add (b,c): **{g,h,f, c,i,d, a,b},** {e}

Edge	Weight
hg	**1**
ci	**2**
gf	**2**
ab	**4**
cf	**4**
gi	6
hi	7
cd	**7**
bc	**8**
ah	8
de	9
ef	10
bh	11
df	14

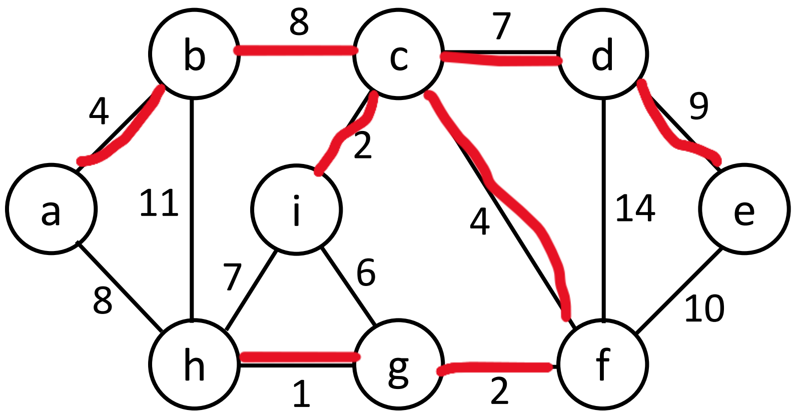
Kruskal’s algorithm example



1. Add (h,g): **{g,h}**,{a},{b},{c},{d},{e},{f}, {i}
2. Add (c,i): **{g,h}** ,**{c,i}**,{a},{b}, {d},{e},{f}
3. Add (g,f): **{g,h,f}, {c,i}**, {a},{b}, {d},{e}
4. Add (a,b): **{g,h,f}, {c,i}**, **{a,b}**,{d},{e}
5. Add (c,f): **{g,h,f, c,i}, {a,b}**,{d},{e}
6. Ignore (g,i): why?
7. Ignore (h,i): why?
8. Add (c,d): **{g,h,f, c,i,d}, {a,b}**,{e}
9. Add (b,c): **{g,h,f, c,i,d, a,b},** {e}
10. Ignore (a,h): why?
11. Add (d,e):**{g,h,f, c,i,d, a,b,e}**

Edge	Weight
hg	**1**
ci	**2**
gf	**2**
ab	**4**
cf	**4**
gi	**6**
hi	**7**
cd	**7**
bc	**8**
ah	**8**
de	9
ef	10
bh	11
df	14

Kruskal's algorithm example



1. Add (h,g): **{g,h}**,{a},{b},{c},{d},{e},{f}, {i}
2. Add (c,i): **{g,h}** ,**{c,i}**,{a},{b}, {d},{e},{f}
3. Add (g,f): **{g,h,f}, {c,i}**, {a},{b}, {d},{e}
4. Add (a,b): **{g,h,f}, {c,i}**, **{a,b}**,{d},{e}
5. Add (c,f): **{g,h,f, c,i}, {a,b}**,{d},{e}
6. Ignore (g,i): why?
7. Ignore (h,i): why?
8. Add (c,d): **{g,h,f, c,i,d}, {a,b}**,{e}
9. Add (b,c): **{g,h,f, c,i,d, a,b},** {e}
10. Ignore (a,h): why?
11. Add (d,e):**{g,h,f, c,i,d, a,b,e}**
12. Ignore (e,f): **{g,h,f, c,i,d, a,b,e}**
13. Ignore (b,h): **{g,h,f, c,i,d, a,b,e}**
14. Ignore (d,f): **{g,h,f, c,i,d, a,b,e}**

Edge	Weight
hg	1
ci	2
gf	2
ab	4
cf	4
gi	6
hi	7
cd	7
bc	8
ah	8
de	9
ef	10
bh	11
df	14

Implementing Kruskal’s algorithm

- Use:
  - adjacency list to represent the graph
  - disjoint set to represent each tree in the forest
  - binary heap for edges

MST: Kruskal's Algorithm

- Difference with Prim's algorithm:
  - Prim's algorithm grows one tree all the time
  - Kruskal's algorithm grows multiple trees (i.e., a forest) at the same time.
  - Since an MST has exactly |V| - 1 edges, after |V| - 1 merges, we would have only one component (one merged tree)

Applications of MST

- Find the cheapest connections for cities, computers, networks, etc.

- Plan road repairs in city or between towns such that traffic continues to flow.

Quiz

- Compare the performance of Prim’s and Kruskal algorithms.

Summary

- Topological sorting and its applications.
- Minimum spanning tree algorithms and applications.

Next

- Shortest-path algorithms

Graphs: Shortest Path Algorithms

Outline

- Shortest Path algorithms:
  - Dijkstra’s,
  - Floyd’s
- Applications

Shortest paths: preliminaries

- **Path**: sequence of edges connecting two vertices.
- **BFS** returns shortest path in an ***unweighted graph***.
    - **BFS** also returns shortest path if all ***weights are the same*** in a ***weighted graph***.
- In general, the shortest path in a ***weighted graph*** may pass through many intermediate vertices.
    - BFS won't work in such a case.

Shortest paths: preliminaries

Two main algorithms:

- Dijkstra's algorithm:
  - Takes as input start and destination vertices and finds the shortest path between them.
  - Other implementations find the shortest path from a ***start vertex*** and all other vertices, i.e., *a shortest path spanning tree rooted in the start vertex.*
- Floyd's algorithm:
  - Finds the shortest path between ***all pairs*** of vertices in a graph
- We assume ***positive weights*** to avoid ***looping***.

Dijkstra’s algorithm

- A greedy algorithm.
- Uses ***distance/weight/cost*** to determine shortest path from a vertex ***s***.
  - Repeatedly
    - selects the smallest distance/weight/cost,
    - extend the path one edge at a time,
    - until all vertices are included.
- Given (s,…,x,….t) is the shortest part from s to t, then (s,…,x) should be the shortest path from s to x.
- Comparison to Prim’s algorithm- similar except:
  - Instead of just considering the weight of the potential edge, it also *considers the distance from the start edge to the vertex from which the edge emanates*.

Dijkstra’s algorithm: pseudocode

- **Dijkstra(G,s,t): //shortest path from s to t**

```
path={s}
for i=1 to n
   distance[i]= ∞
for each edge (s,v)
   distance[v]=w(s,v) #initially, the distances are just weights
last=s //set last vertex to s
while(last!=t)
   select v_next,  such that v_next is the unknown vertex
   minimizing distance[v]
   for each edge (v_next,x)
      distance[x]=min(distance[x],distance[v_next]+w(v_next,x))
      //checks if a shorter path to x exists via v_next.
   last=v_next
   path=path U {v_next}
```

Dijkstra's algorithm: pseudocode

- **Dijkstra(G,s,t): //shortest path from s to t**

```
path={s}
for i=1 to n
    distance[i]= ∞
for each edge (s,v)
    distance[v]=w(s,v) #initially, the distances are just weights
last=s //set last vertex to s
while(last!=t)
    select v_next,  such that v_next is the unknown vertex
    minimizing distance[v]
    for each edge (v_next,x)
        distance[x]=min(distance[x],distance[v_next]+w(v_next,x))
    last=v_next
    path=path U {v_next}
```

In Prim's algorithm, they were always the weights

but in Dijkstra they are the
shortest distance to that vertex (so far)

Dijkstra’s algorithm: pseudocode

- Dijkstra(G,s,t): //shortest path from s to t

```
path={s}
for i=1 to n
    distance[i]= ∞
for each edge (s,v)
    distance[v]=w(s,v) #initially, the distances are just weights
last=s //set last vertex to s
while(last!=t)
    select v_next,  such that v_next is the unknown vertex
    minimizing distance[v]
    for each edge (v_next,x)
        distance[x]=min(distance[x],distance[v_next]+w(v_next,x))
    last=v_next
    path=path U {v_next}
```

Extends the path from the vertex with the shortest distance so far

Dijkstra's algorithm: pseudocode

```
Dijkstra(G,s,t): //shortest path from s to t
   path={s}
   for i=1 to n
      distance[i]= ∞
   for each edge (s,v)
      distance[v]=w(s,v) #initially, the distances are just weights
   last=s //set last vertex to s
   while(last!=t)
      select v_next,  such that v_next is the unknown vertex
      minimizing distance[v]
      for each edge (v_next,x)
         distance[x]=min(distance[x],distance[v_next]+w(v_next,x))
      last=v_next
      path=path U {v_next}
```

1. We now have a new way of reaching x …

2. … so update the (total) distance to x …

3. …but only if it is less than the current distance

Dijkstra's algorithm: implementation

```
dijkstra(graph *g, int start)
{
    node *temp;
    bool intree[MAXV+1] ;//marks status if vertex is in tree yet
    int distance[MAXV+1];//cost of adding vertex to tree
    int parent[MAXV+1]; //parent vertex
    int current_vertex;// current vertex being processed
    int candidate_vertex; //potential next vertex
    int dist=0;//cheapest cost to enlarge tree
    int weight=0 ; //tree weight
    for(int i=1;i<=nvertices;i++)
    {
        intree[i]=false;
        distance[i]=INT_MAX;
        parent[i]=-1;
    }
    distance[start]=0;
    current_vertex=start;
```

Dikstra’s algorithm: implementation

```
while(!intree[current_vertex])
{
	intree[current_vertex]=true;
	if(current_vertex!=start)
	{
		cout<<"\n\tedge("<<parent[current_vertex]<<","<<current_vertex<<") in tree\n";
		weight=weight+dist;

	}
	temp=adjLists[current_vertex].head;
	while(temp) //get all adjacent vertices
	{
		candidate_vertex=temp->dest;
		if(distance[candidate_vertex]>(distance[current_vertex]+ temp->weight))//difference to Prim’s
		{
			distance[candidate_vertex]= distance[current_vertex]+ temp->weight;//difference to Prim’s
			parent[candidate_vertex]=current_vertex;
```

Dikstra's algorithm: implementation

```
            }
            temp=temp->next;//obtain next adjacent node.
        }//end of while loop accessing the vertices
        current_vertex=1;
        dist=INT_MAX;
        //now pick node with lowest distance
        for(int i=1;i<=nvertices;i++)
        {
            if((!intree[i])&&(dist>distance[i]))
            {
                dist=distance[i];
                current_vertex=i;
            }
        }//end for
    }//end loop for intree
    return weight;
}
```

All-pairs shortest path: Floyd's algorithm

- Suitable for applications like finding the ***center*** or ***diameter*** of a graph, which requires finding shortest path between all pairs of vertices.
- If we run Dijkstra's n times (once for each start vertex), we achieve this in $O(n^3)$

All-pairs shortest path: Floyd's algorithm

- Find center of graph:
  - Minimize longest and average distance to all other vertices.
  - ***Application***: optimal location for an outlet to serve the greatest number of people.
- Find diameter of a graph:
  - Minimize longest shortest-path distance over all pairs of vertices.
  - ***Application***: communication- determine the longest possible time for a network packet to be delivered.
- Compute the shortest path between all pairs of vertices using an ***nxn distance matrix***.

Floyd's Algorithm: solution approach

- Simple solution:
  - Call Dijkstra's algorithm from each of the n possible starting vertices.
  - Takes $O(n^3)$

- Floyd-Warshall algorithm:
  - Construct a n x n shortest path distance matrix directly from n x n weight matrix.
  - Implement using adjacency matrix, instead of adjacency list data structure.

Floyd-Washall algorithm: formulation

- A ***dynamic-programmin***g algorithm.
- Considers the ***intermediate vertices*** of a shortest path.
  - ***Intermediate vertex***: An intermediate vertex of a simple path $p=<v_1,v_2,....,v_j>$ is any vertex of p other than v_1 or v_j, i.e. any vertex in the set $\{v_2,v_3,....v_{j-1}\}$.
- **Assumption**: Assuming the vertices of G are V={1,2,……,n}, let’s consider a subset {1,2,…..,k} for some k.
- For any pair of vertices (i,j) ϵ V, considering all paths from i to j, where the intermediate vertices are all drawn from {1,2,….,k}, let p be a *minimum-weight path* from among them.

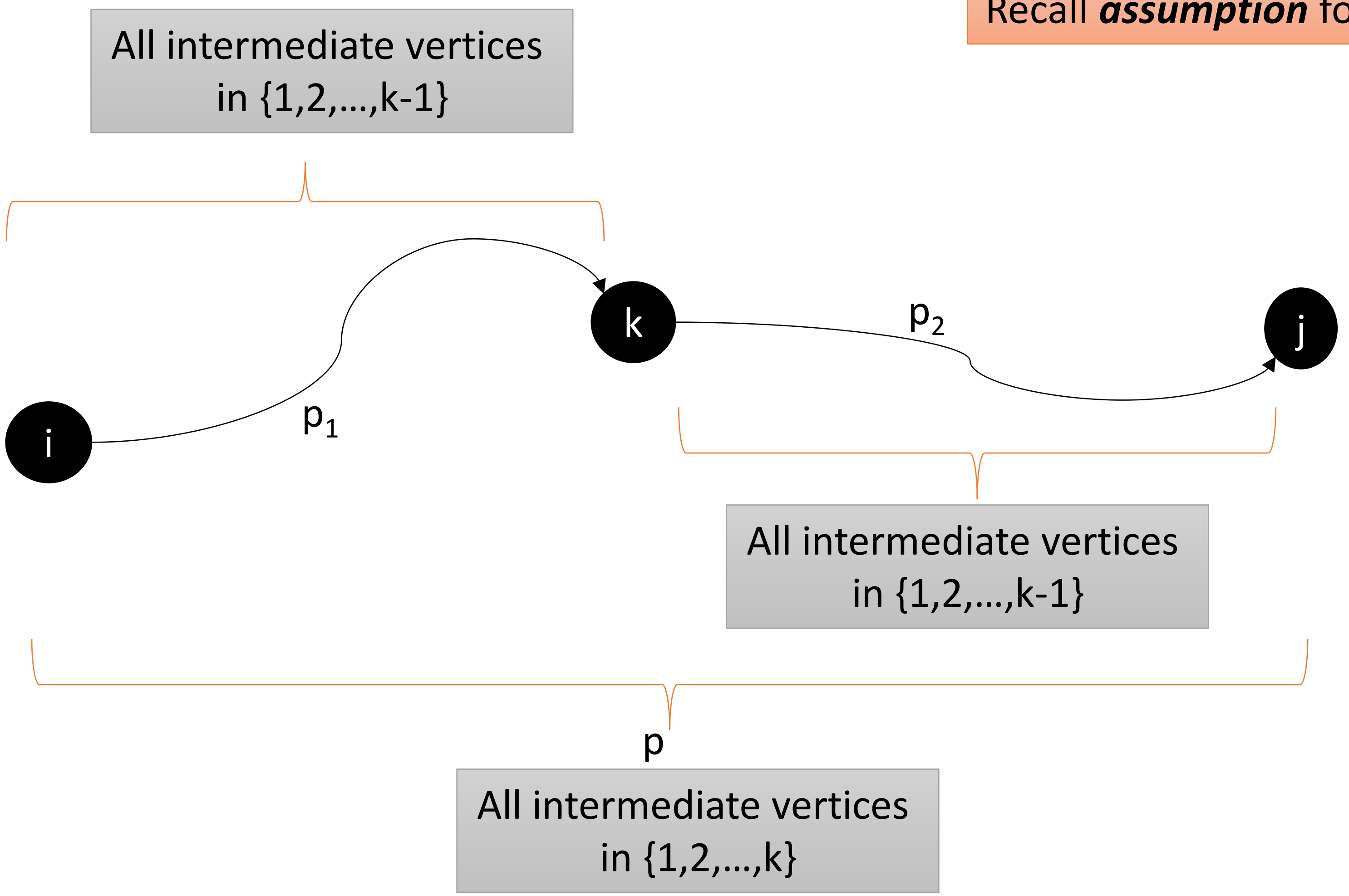
Floyd-Washall algorithm: formulation

- Exploits a relationship between path **p** and shortest paths ***from i to j*** with all intermediate vertices in the set ***{1,2,…,k-1}.***
  - If *k* is **not an** intermediate vertex of path p, then all intermediate vertices of path p are in the set ***{1,2,…,k-1}.***
    - Thus, a shortest path from vertex ***i to vertex j*** with all intermediate vertices in the set ***{1,2,…,k-1}*** is also a shortest path ***from i to j*** with all intermediate vertices in the set ***{1,2,…,k}*** , since ***{1,2,…,k-1}*** $\subseteq$ ***{1,2,…,k}***
  - If k **is an** intermediate vertex of path p, then we decompose **p** into $\mathbf{p_1}$(*from i to k*) and $\mathbf{p_2}$(*from k to j*), with both *p1 and p2 deriving their intermediate vertices* from **{1,2,…,k}-{k}**, i.e. **{1,2,…,k-1}.**

See illustration in the next slide.

Floyd-Washall algorithm: formulation

Recall ***assumption*** for the set {1,2,…k} two slides back.



Floyd-Washall algorithm: formulation

- Let $d_{ij}^{(k),}$ be the weight/cost/distance of a shortest path from vertex i to j with all intermediate vertices in {1,2,…,k}.
- For k=0,
- A recursive formulation of shortest path estimates is defined as:
  - $d_{ij}^{(k)=}w_{ij}$, if k=0 {*path with at most one edge; no intermediate vertices*}

  - $d_{ij}^{(k)}=\min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)}+d_{kj}^{(k-1)})$, if k≥1 {*path has intermediate vertices*}.
- Given that for any path, all intermediate vertices are in the set {1,2,…,n}, the matrix $D^{(n)}=(d_{ij}^{(n)})$, gives all shortest pairs.

Floyd-Washall algorithm: pseudocode

Floyd-Warshall(W):#W: nxn weight matrix

set n: number of vertices in W

initialize distance matrix $D^{(0)=}W$ #initial distance matrix

for k=1 **to** n

let $D^{(k)}=(d_{ij}^{(k)})$ be a new **n x n matrix** *#We will have matrices*

#$D^{(1)}$, $D^{(2)}$….$D^{(n)}$. The final matrix $D^{(n)}$ is returned.

for i=1 **to** n

for j=1 **to** n

$d_{ij}^{(k)}= \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)}+d_{kj}^{(k-1)})$

return $D^{(n)}$ #at this point ***k=n***, the final matrix.

Floyd’s algorithm: implementation

```
#define MAXV 100

struct adjacency_matrix
{
int weight[MAXV+1][MAXV+1]; //for adjacency or weight information
int nvertices; //number of vertices in graph
};
```

Floyd’s algorithm: implementation

```
void floyd(adjacency_matrix *g){
      int i,j;//counters
      int k; //intermediate vertex counter
      int through_k;//distance through vertex k
      for(k=1;k<=g->nvertices;k++)
      {
            for(i=1;i<=g->nvertices;i++)
            {
                  for(j=1;j<=g->nvertices;j++)
                  {
                        through_k=g->weight[i][k]+g->weight[k][j];
                        if(through_k<g->weight[i][j])
                        {
                                    g->weight[i][j]=through_k;
                        }
                  }
            }
      }
}
```

Comments on Performance

- Dijkstra's:
  - O(n^2) with simple data structures.
  - Constructs actual shortest path between any given pair of vertices.
- Floyd's: O(n^3).
  - No better than ***n*** calls to Dijkstra's but performs better in practice.
  - Does not construct actual shortest path between any given pair of vertices.

Acknowledgement

Adapted from material by Prof. David Vernon

Augmented by material from:

The Algorithm Design Manual 2nd Edition: by Steven Skiena

Introduction to Algorithms, 3rd Edition, Thomas H. Cormen et al. (2009)