Module 5
Graph Algorithms

Dr. Natarajan Meghanathan
Professor of Computer Science
Jackson State University
Jackson, MS 39217
E-mail: natarajan.meghanathan@jsums.edu

Module Topics

- 5.1 Traversal (DFS, BFS)
  - Brute Force
- 5.2 Topological Sorting of a DAG
  - Decrease and Conquer
- 5.3 Single-Source Shortest Path Algorithms (Dijkstra and Bellman-Ford)
  - Greedy
- 5.4 Minimum Spanning Trees (Prim's, Kruskal's)
  - Greedy
- 5.5 All Pairs Shortest Path Algorithm (Floyd's)
  - Dynamic Programming

5.1 Graph Traversal Algorithms

Depth First Search (DFS)

- Visits graph's vertices (also called nodes) by always moving away from last visited vertex to unvisited one, backtracks if there is no adjacent unvisited vertex.

- Break any tie to visit an adjacent vertex, by visiting the vertex with the lowest ID or the lowest alphabet (label).

- Uses a stack
  – a vertex is pushed onto the stack when it's visited for the first time
  –a vertex is popped off the stack when it becomes a dead end, i.e., when there is no adjacent unvisited vertex

- "Redraws" graph in tree-like fashion (with tree edges and back edges for undirected graph):
  – Whenever a new unvisited vertex is reached for the first time, it is attached as a child to the vertex from which it is being reached. Such an edge is called a tree edge.
  – While exploring the neighbors of a vertex, it the algorithm encounters an edge leading to a previously visited vertex other than its immediate predecessor (i.e., its parent in the tree), such an edge is called a back edge.
  – The leaf nodes have no children; the root node and other intermediate nodes have one more child.

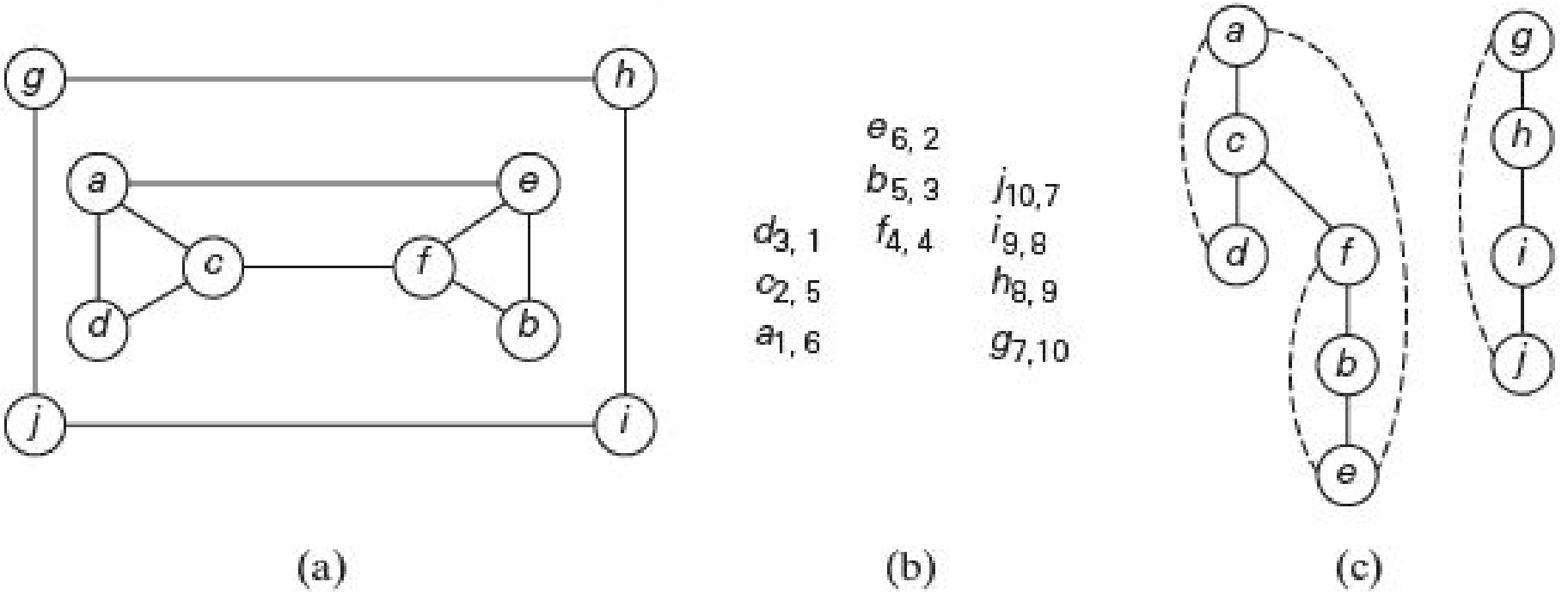
Pseudo Code of DFS

ALGORITHM *DFS*(G)

```
//Implements a depth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//        in the order they are first encountered by the DFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        dfs(v)
```

```
dfs(v)
//visits recursively all the unvisited vertices connected to vertex v
//by a path and numbers them in the order they are encountered
//via global variable count
count ← count + 1;   mark v with count
for each vertex w in V adjacent to v do
    if w is marked with 0
        dfs(w)
```

Example 1: DFS



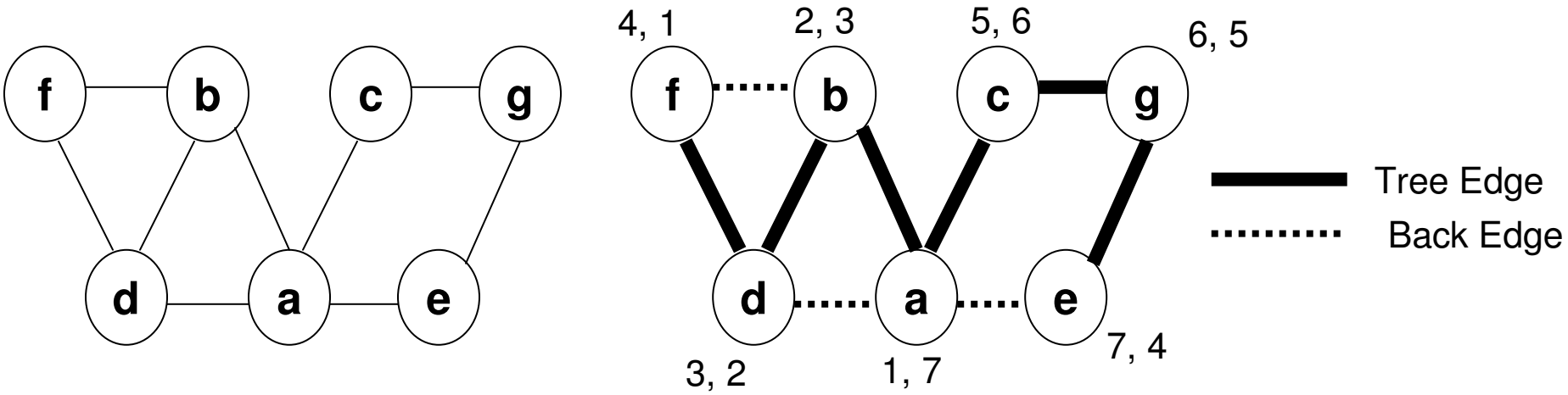
Example of a DFS traversal. (a) Graph. (b) Traversal's stack (the first subscript number indicates the order in which a vertex is visited, i.e., pushed onto the stack; the second one indicates the order in which it becomes a dead-end, i.e., popped off the stack). (c) DFS forest with the tree and back edges shown with solid and dashed lines, respectively.

Source: Figure 3.10: Levitin, 3rd Edition: Introduction to the Design and Analysis of Algorithms, 2012.

DFS

- DFS can be implemented with graphs represented as:
  - –adjacency matrices: $\Theta(V^2)$; adjacency lists: $\Theta(|V|+|E|)$
- Yields two distinct ordering of vertices:
  - –order in which vertices are first encountered (pushed onto stack)
  - –order in which vertices become dead-ends (popped off stack)
- Applications:
  - –checking connectivity, finding connected components
    - The set of vertices that we can visit through DFS, starting from a particular vertex in the set constitute a connected component.
    - If a graph has only one connected component, we need to run DFS only once and it returns a tree; otherwise, the graph has more than one connected component and we determine a forest – comprising of trees for each component.
  - –checking for cycles (a DFS run on an undirected graph returns a back edge)
  - –finding articulation points and bi-connected components
    - An articulation point of a connected component is a vertex that when removed disconnects the component.
    - A graph is said to have bi-connected components if none of its components have an articulation point.

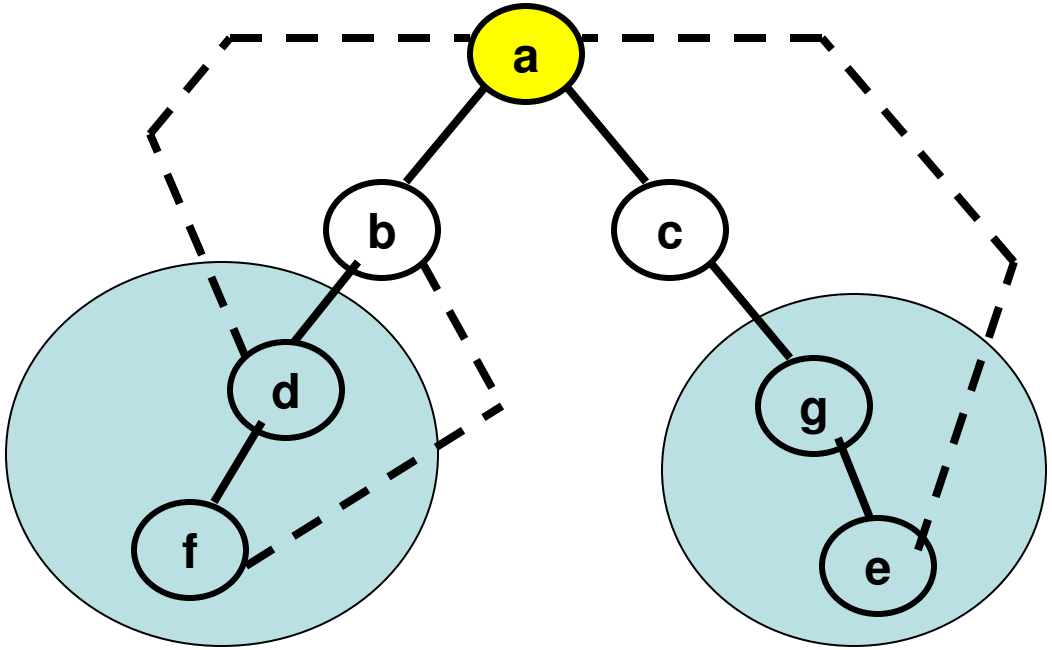
Example 2: DFS



- Notes on Articulation Point
  - The root of a DFS tree is an articulation point if it has more than one child connected through a tree edge. (In the above DFS tree, the root node ‘a’ is an articulation point)
  - The leaf nodes of a DFS tree are not articulation points.
  - Any other internal vertex *v* in the DFS tree, if it has one or more sub trees rooted at a child (at least one child node) of *v* that does NOT have an edge which climbs ’higher ’ than *v* (through a back edge), then *v* is an articulation point.

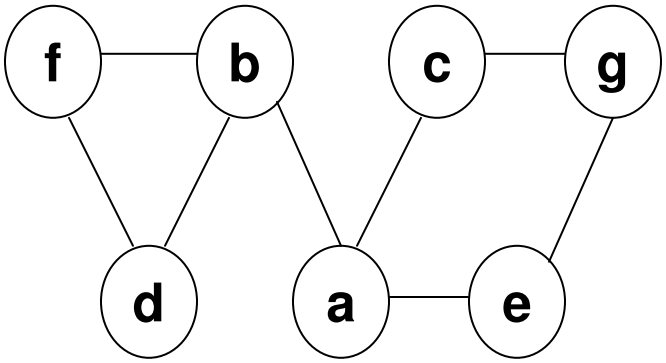
DFS: Articulation Points

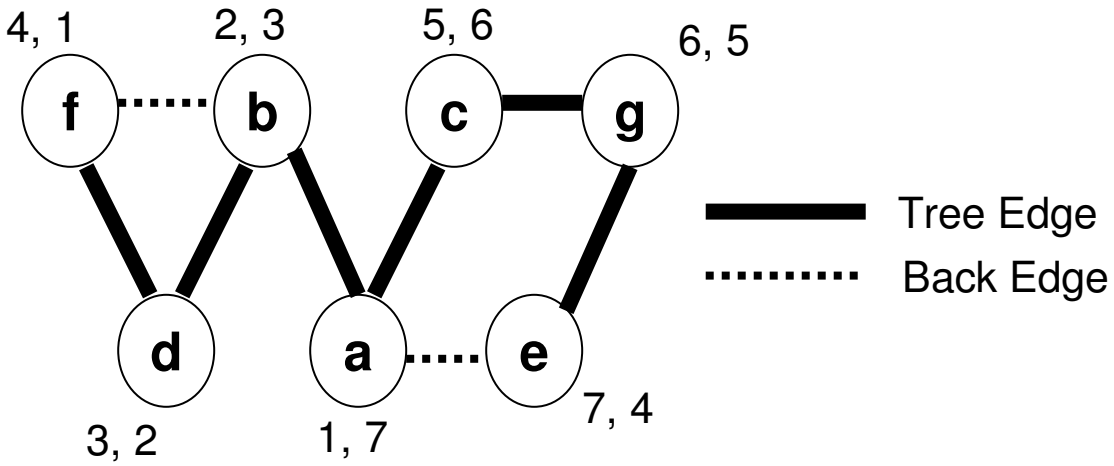
Based on Example 2

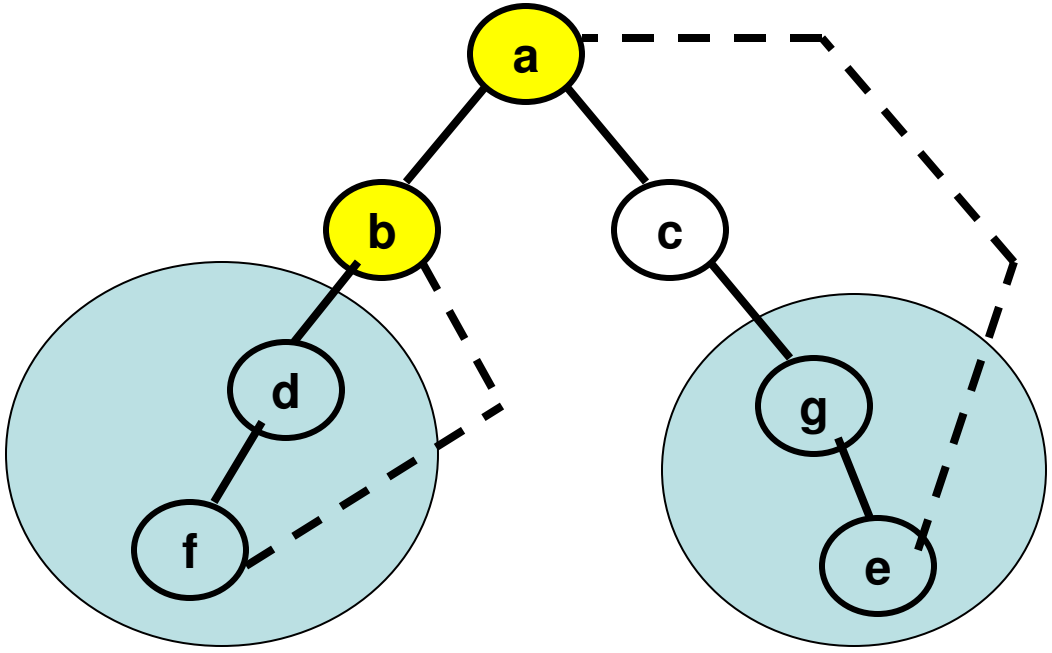


- In the above graph, vertex ‘a’ is the only articulation point.
- Vertices ‘e’ and ‘f’ are leaf nodes.
- Vertices ‘b’ and ‘c’ are candidates for articulation points. But, they cannot become articulation point, because there is a back edge from the only sub tree rooted at their child nodes (‘d’ and ‘g’ respectively) that have a back edge to ‘a’.
- By the same argument, vertices ‘d’ and ‘g’ are not articulation points, because they have only child node (f and e respectively); each of these child nodes are connected to a higher level vertex (b and a respectively) through a back edge.

Example 3: DFS and Articulation Points

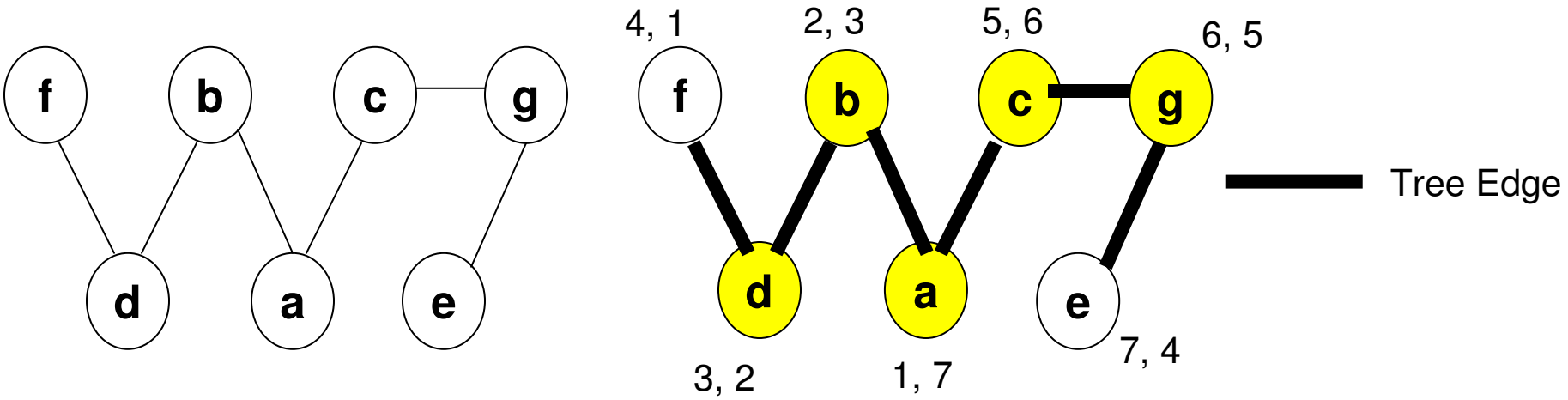






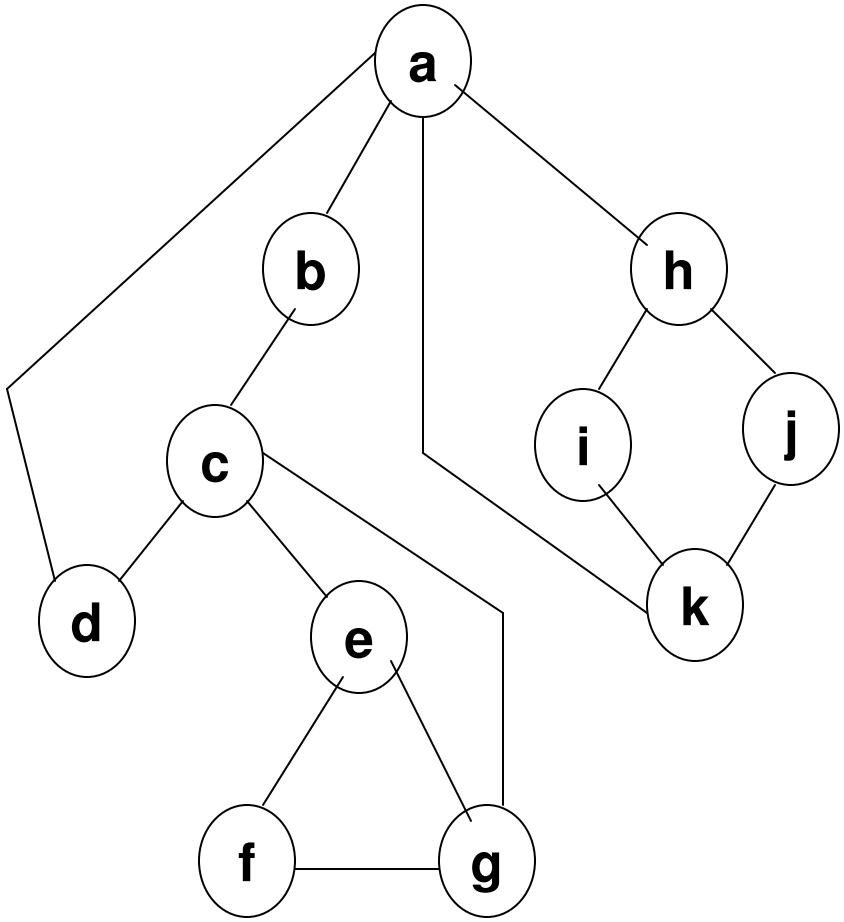
- In the above new graph (different from the previous example: note edge a – e and b – f are added back; but a – d is missing):
  - Vertices ‘a’ and ‘b’ are articulation points
  - Vertex ‘c’ is not an articulation point

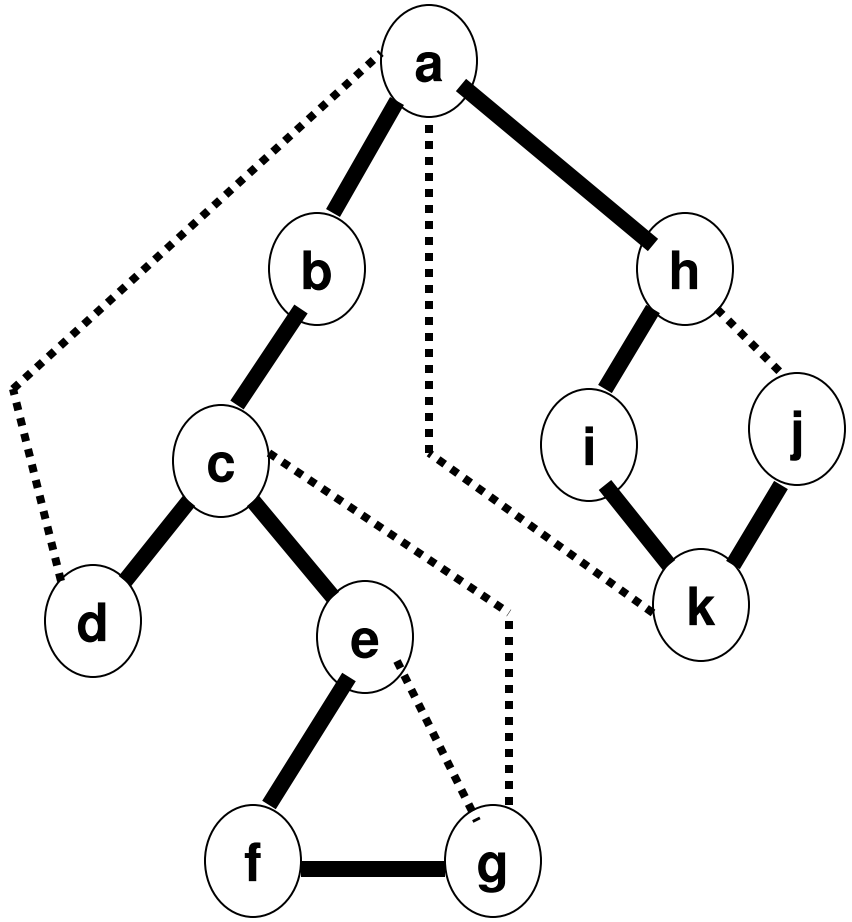
Example 4: DFS and Articulation Points



- In the above new graph (different from the previous example: note edges b – f, a – d and a – e are missing), vertices ‘a’, ‘b’, ‘c’, ‘d’ and ‘g’ are articulation points, because:
  - Vertex ‘a’ is the root node of the DFS tree and it has more than one child node
  - Vertex ‘b’ is an intermediate node; it has one sub tree rooted at its child node (d) that does not have any node, including ‘d’, to climb higher than ‘b’. So, vertex ‘b’ is an articulation point.
  - Vertex ‘c’ is also an articulation point, by the same argument as above – this time, applied to the sub tree rooted at child node ‘g’.
  - Vertices ‘d’ and ‘g’ are articulation points; because, they have one child node (‘f’ and ‘e’ respectively) that are not connected to any other vertex higher than ‘d’ and ‘g’ respectively.

Example 5: DFS and Articulation Points





DFS TREE

Identification of the Articulation Points of the Graph in Example 5

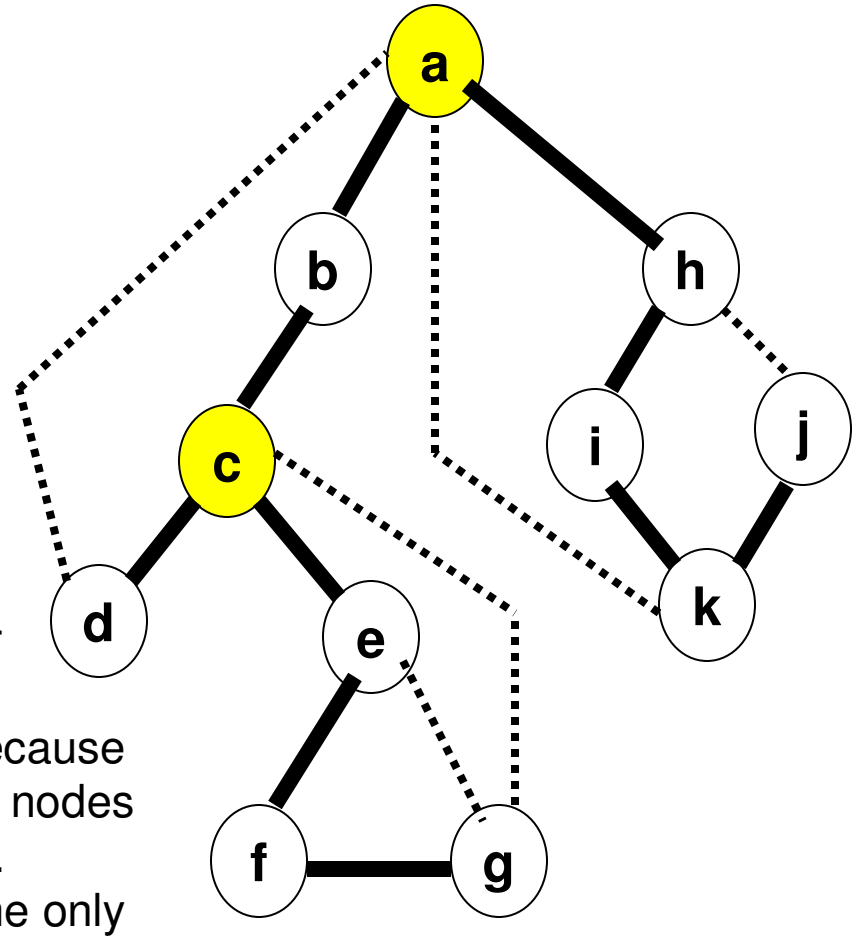
1) Root Vertex 'a' has more than one child; so, it is an articulation point.

2) Vertices '**d**', '**g**' and '**j**' are leaf nodes

3) Vertex '**b**' is not an articulation point because the only sub tree rooted at its child node 'c' has a back edge to a vertex higher than 'b' (in this case to the root vertex 'a')

4) **Vertex 'c' is an articulation point.** One of its child vertex 'd' does not have any sub tree rooted at it. The other vertex 'e' has a sub tree rooted at it and this sub tree has no back edge higher up than 'c'.

5) By argument (4), it follows that vertex '**e**' is not an articulation point because the sub tree rooted at its child node 'f' has a back edge higher up than 'e' (to vertex 'c');

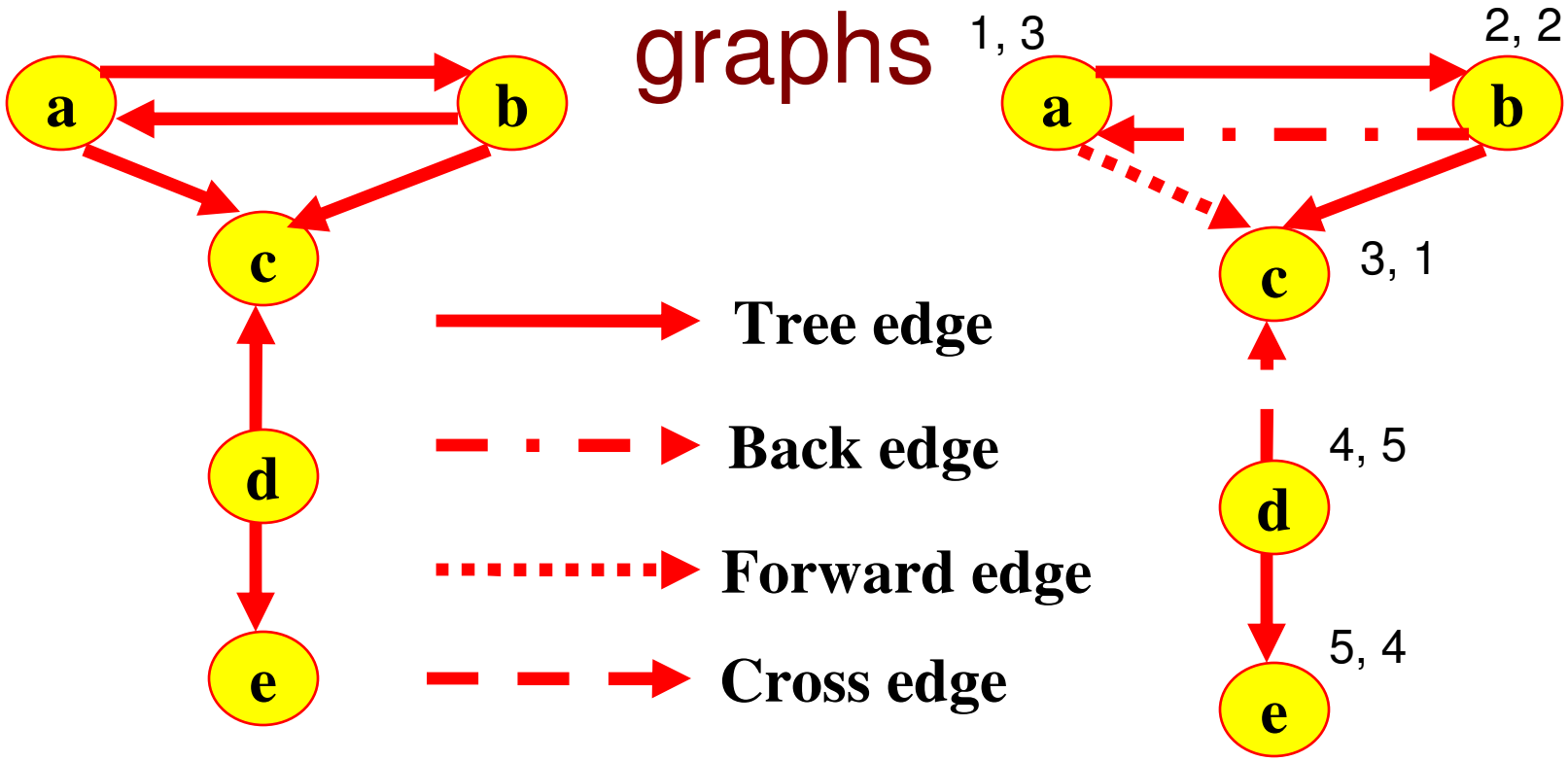


6) Vertices 'f' and 'k' are not articulation points because they have only one child node each and the child nodes are connected to a vertex higher above 'f' and 'k'.

7) Vertex 'i' is not an articulation point because the only sub tree rooted at its child has a back edge higher up (to vertices 'a' and 'h').

8) Vertex 'h' is not an articulation point because the only sub tree rooted at 'h' has a back edge higher up (to the root vertex 'a').

DFS: Edge Terminology for directed graphs

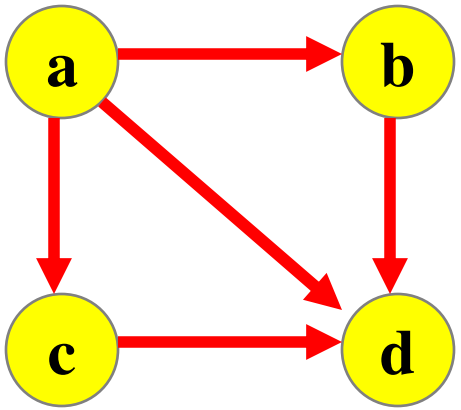


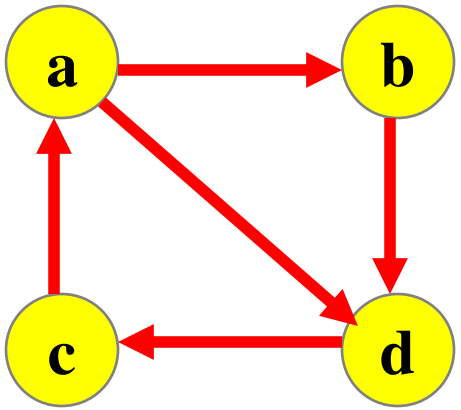
Tree edge – an edge from a parent node to a child node in the tree
Back edge – an edge from a vertex to its ancestor node in the tree
Forward edge – an edge from an ancestor node to its descendant node in the tree. The two nodes do not have a parent-child relationship. The back and forward edges are in a single component (the DFS tree).
Cross edge – an edge between two different components of the DFS Forest. So, basically an edge other than a tree edge, back edge and forward edge

Directed Acyclic Graphs (DAG)

- A directed graph is a graph with directed edges between its vertices (e.g., u → v).
- A DAG is a directed graph (digraph) without cycles.
  - A DAG is encountered for many applications that involve pre-requisite restricted tasks (e.g., course scheduling)

a DAG

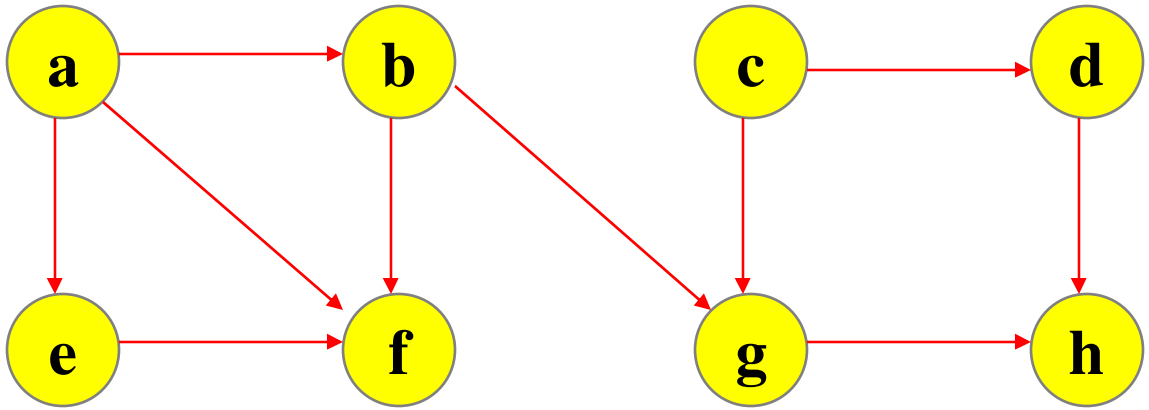




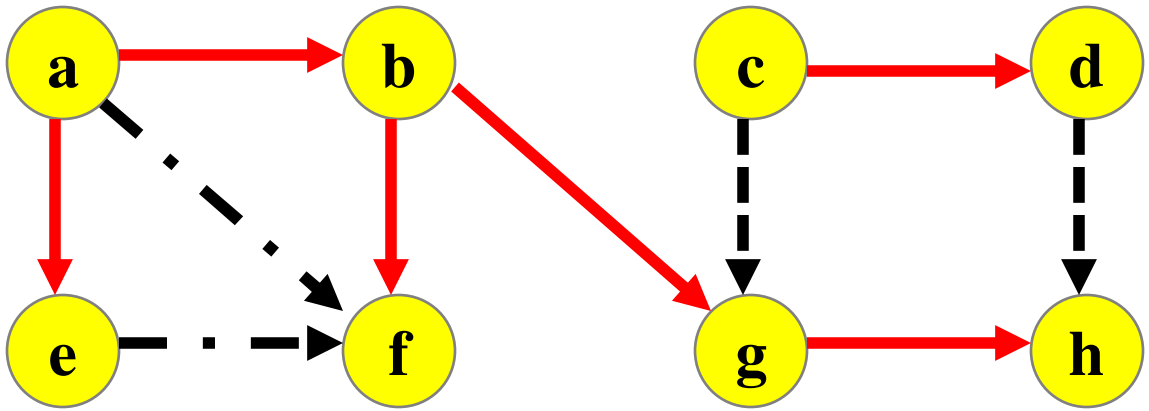
not a DAG

To test whether a directed graph is a DAG, run DFS on the directed graph. If a back edge is not encountered, then the directed graph is a DAG.

DFS on a DAG



$h_{5,2}$
$g_{4,3}$
$f_{3,1}$
$b_{2,4}$ $e_{6,5}$ $d_{8,7}$
$a_{1,6}$ $c_{7,8}$



Forward edge

Cross edge

Order in which the Vertices are popped of from the stack

f h g b e a d c

Reverse the order

Topological Sort

c d a e b g h f

Topological Sort

- Topological sort is an ordering of the vertices of a directed acyclic graph (DAG) – a directed graph (a.k.a. digraph) without cycles.
  - This implies if there is an edge u→ v in the digraph, then *u* should be listed ahead of *v* in the topological sort: … u … v …
  - Being a DAG is the necessary and sufficient condition to be able to do a topological sorting for a digraph.
  - **Proof for Necessary Condition:** If a digraph is not a DAG and lets say it has a topological sorting. Consider a cycle (in the digraph) comprising of vertices u1, u2, u3, …, uk, u1. In other words, there is an edge from uk to u1 and there is a directed path to uk from u1. So, it is not possible to decide whether u1 should be ahead of uk or after uk in the topological sorting of the vertices of the digraph. Hence, there cannot be a topological sorting of the vertices of a digraph, if the digraph has even one cycle. **To be able to topologically sort the vertices of a digraph, the digraph has to first of all be a DAG. [Necessary Condition].** We will next prove that this is also the sufficient condition.

Topological Sort
Proof for Sufficient Condition

- After running DFS on the digraph (also a DAG), the topological sorting is the listing of the vertices of the DAG in the reverse order according to which they are removed from the stack.
  - Consider an edge u → v in the digraph (DAG).
  - If there exists, an ordering that lists v ahead of u, then it implies that u was popped out from the stack ahead of v. That is, vertex v has been already added to the stack and we were to able to visit vertex u by exploring a path leading from v to u. This means the edge u → v has to be a **back edge**. This implies, the digraph has a cycle and is not a DAG. We had earlier proved that if a digraph has a cycle, we cannot generate a topological sort of its vertices.

  - For an edge u->v, if v is listed ahead of u ==> the graph is not a DAG (Note that a ==> b, then !b ==> !a)
  - If the graph is a DAG ==> u should be listed ahead of v for every edge u → v.
  - Hence, it is sufficient for a directed to be DAG to generate a topological sort for it.

Breadth First Search (BFS)

- BFS is a graph traversal algorithm (like DFS); but, BFS proceeds in a concentric breadth-wise manner (not depth wise) by first visiting all the vertices that are adjacent to a starting vertex, then all unvisited vertices that are two edges apart from it, and so on.
  - The above traversal strategy of BFS makes it ideal for determining minimum-edge (i.e., minimum-hop paths) on graphs.
- If the underling graph is connected, then all the vertices of the graph should have been visited when BFS is started from a randomly chosen vertex.
  - If there still remains unvisited vertices, the graph is not connected and the algorithm has to restarted on an arbitrary vertex of another connected component of the graph.
- BFS is typically implemented using a FIFO-queue (not a LIFO-stack like that of DFS).
  - The queue is initialized with the traversal's starting vertex, which is marked as visited. On each iteration, BFS identifies all unvisited vertices that are adjacent to the front vertex, marks them as visited, and adds them to the queue; after that, the front vertex is removed from the queue.
- When a vertex is visited for the first time, the corresponding edge that facilitated this visit is called the *tree edge*. When a vertex that is already visited is re-visited through a different edge, the corresponding edge is called a *cross edge*.

Pseudo Code of BFS

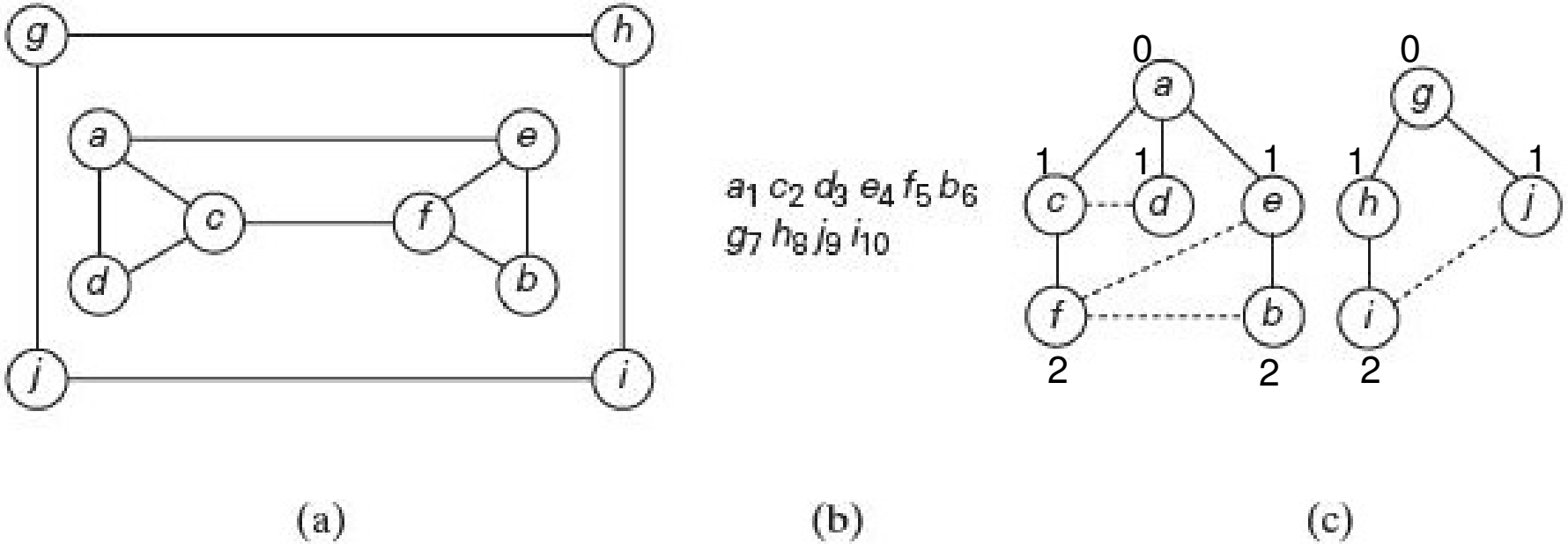
ALGORITHM *BFS*(G)

```
//Implements a breadth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//        in the order they are visited by the BFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
    if v is marked with 0
        bfs(v)

bfs(v)
//visits all the unvisited vertices connected to vertex v
//by a path and numbers them in the order they are visited
//via global variable count
count ← count + 1;   mark v with count and initialize a queue with v
while the queue is not empty do
    for each vertex w in V adjacent to the front vertex do
        if w is marked with 0
            count ← count + 1;   mark w with count
            add w to the queue
    remove the front vertex from the queue
```

BFS can be implemented with graphs represented as:
adjacency matrices: $\Theta(V^2)$; adjacency lists: $\Theta(|V|+|E|)$

Example for BFS

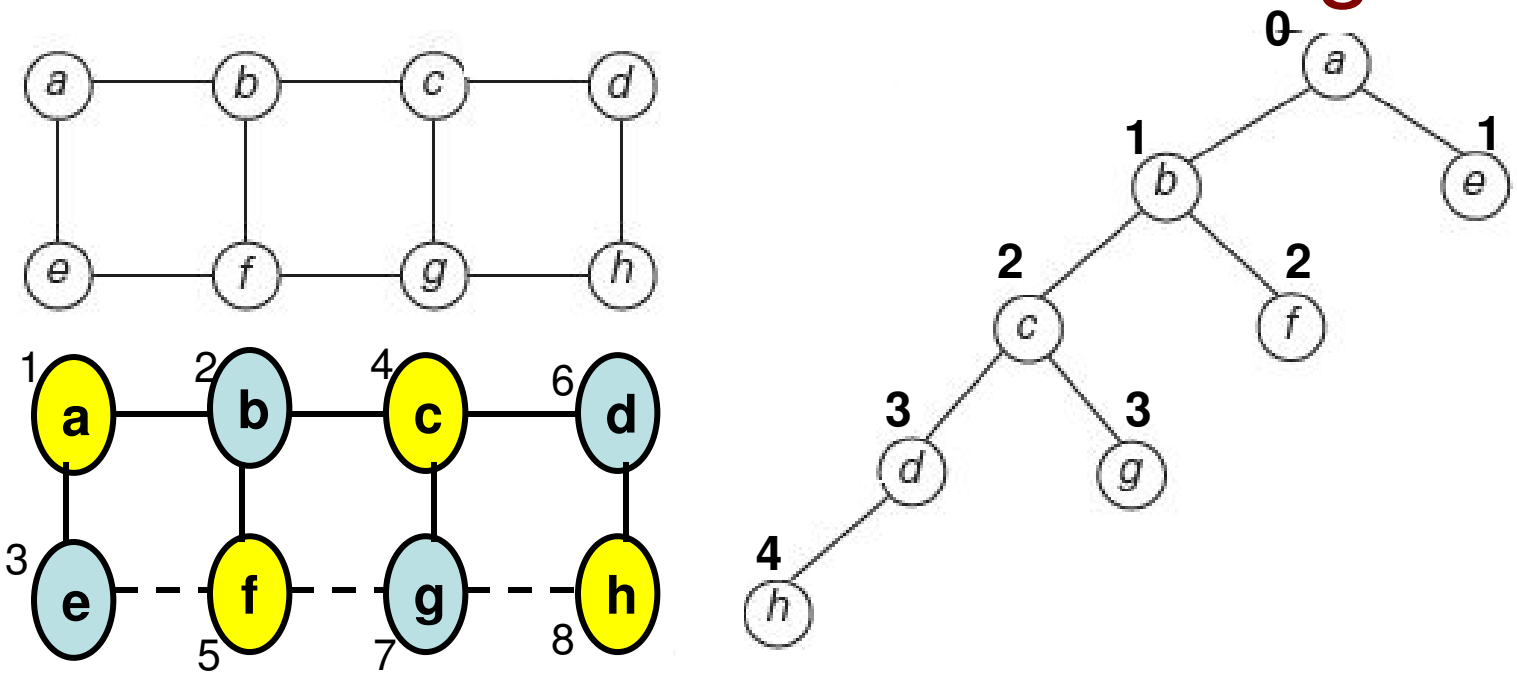


(a) (b) (c)

(a) Graph. (b) Traversal queue, with the numbers indicating the order in which the vertices are visited, i.e., added to (and removed from) the queue. (c) BFS forest with the tree and cross edges shown with solid and dotted lines, respectively.

Source: Figure 3.11: Levitin, 3rd Edition: Introduction to the Design and Analysis of Algorithms, 2012.

Use of BFS to find Minimum Edge Paths



Note: DFS cannot be used to find minimum edge paths, because DFS is not guaranteed to visit all the one-hop neighbors of a vertex, before visiting its two-hop neighbors and so on.

For example, if DFS is executed starting from vertex ‘a’ on the above graph, then vertex ‘e’ would be visited through the path a – b – c – d – h – g – f – e and not through the direct path a – e, available in the graph.

Source: Figure 3.12: Levitin, 3rd Edition: Introduction to the Design and Analysis of Algorithms, 2012.

Comparison of DFS and BFS

	DFS	BFS
Data structure	a stack	a queue
Number of vertex orderings	two orderings	one ordering
Edge types (undirected graphs)	tree and back edges	tree and cross edges
Applications	connectivity, acyclicity, articulation points	connectivity, acyclicity, minimum-edge paths
Efficiency for adjacency matrix	$\Theta(\|V^2\|)$	$\Theta(\|V^2\|)$
Efficiency for adjacency lists	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$

With the levels of a tree, referenced starting from the root node,
A back edge in a DFS tree could connect vertices at different levels; whereas, a cross edge in a BFS tree always connects vertices that are either at the same level or at adjacent levels.

There is always only a unique ordering of the vertices, according to BFS, in the order they are visited (added and removed from the queue in the same order).
On the other hand, with DFS – vertices could be ordered in the order they are added to the Stack, typically different from the order in which they are removed from the stack.

Source: Table 3.1: Levitin, 3rd Edition: Introduction to the Design and Analysis of Algorithms, 2012.

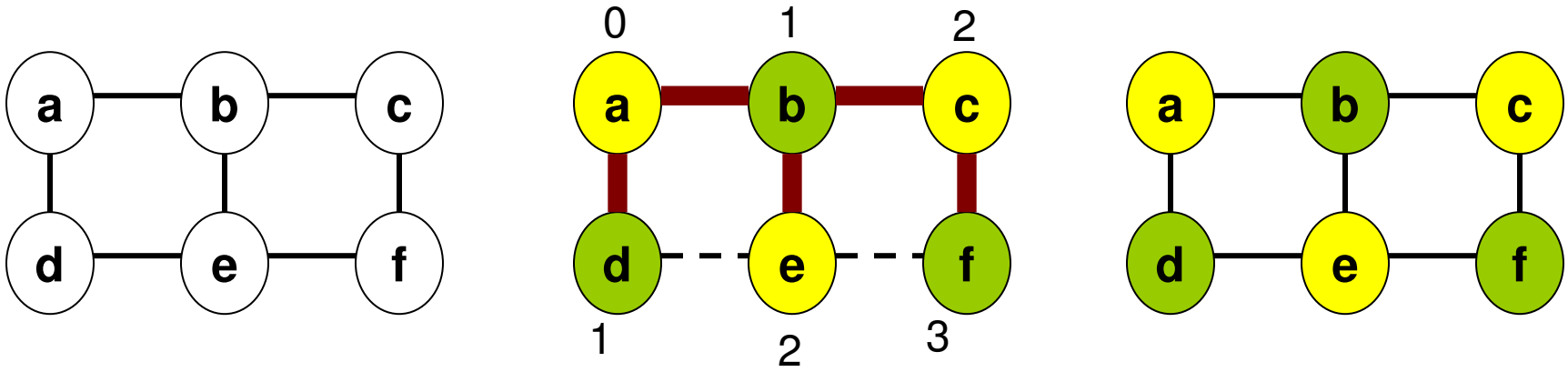
Bi-Partite (2-Colorable) Graphs

- A graph is said to be bi-partite or 2-colorable if the vertices of the graph can be colored in two colors such that every edge has its vertices in different colors.
- In other words, we can partition the set of vertices of a graph into two disjoint sets such that there is no edge between vertices in the same set. All the edges in the graph are between vertices from the two sets.
- We can check for the 2-colorable property of a graph by running a DFS or BFS
  - With BFS, if there are no cross-edges between vertices at the same level, then the graph is 2-colorable.
  - With DFS, if there are no back edges between vertices that are both at odd levels or both at even levels, then the graph is 2-colorable.

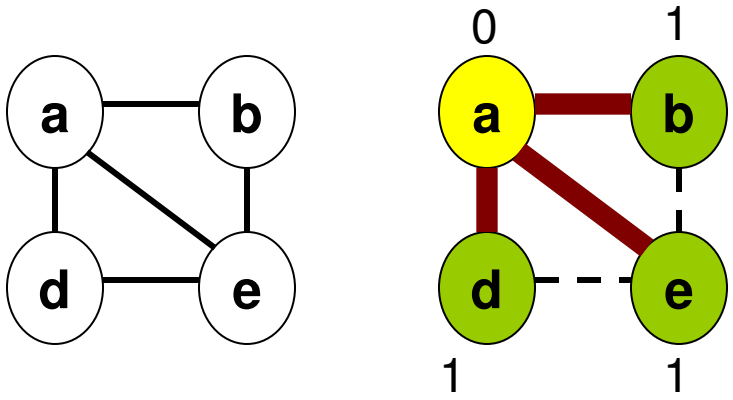
- We will use BFS as the algorithm to check for the 2-colorability of a graph.
  - The level of the root is 0 (consider 0 to be even).
  - The level of a child node is 1 more than the level of the parent node from which it was visited through a tree edge.
  - If the level of a node is even, then color the vertex in yellow.
  - If the level of a node is odd, then color the vertex in green.

Bi-Partite (2-Colorable) Graphs

Example for a 2-Colorable Graph

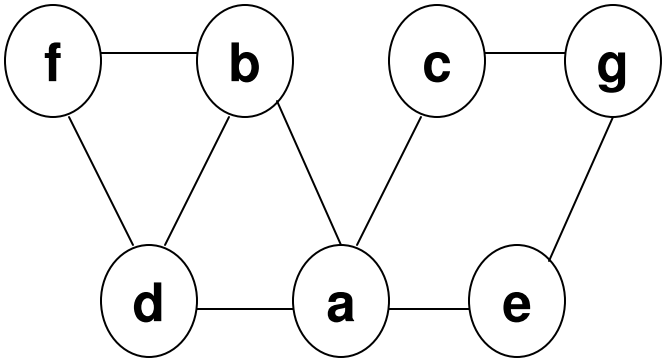


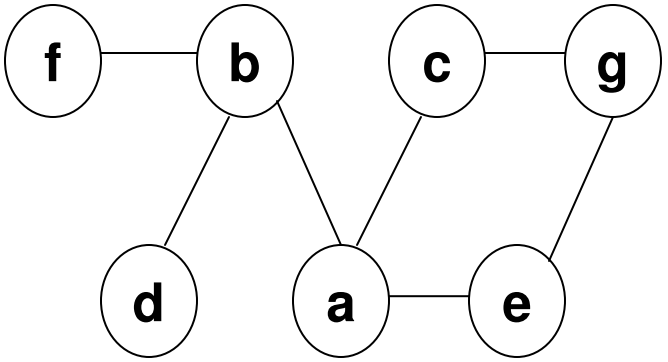
Example for a Graph that is Not 2-Colorable

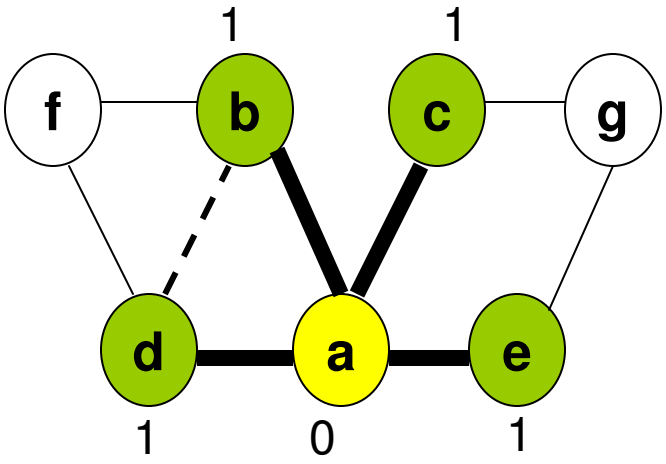


We encounter cross edges between vertices b and e; d and e – all the three vertices are in the same level.

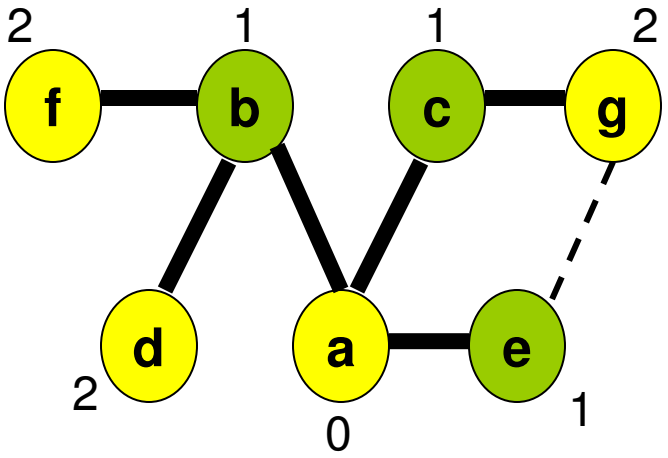
Examples: 2-Colorability of Graphs







b – d is a cross edge between Vertices at the same level. So, the graph is not 2-colorable



The above graph is 2-Colorable as there are no cross edges between vertices at the same level

Dijkstra’s Shortest Path Algorithm

Shortest Path (Min. Wt. Path) Problem

- Path p of length k from a vertex s to a vertex d is a sequence $(v_0, v_1, v_2, \ldots, v_k)$ of vertices such that $v_0 = s$ and $v_k = d$ and $(v_{i-1}, v_i) \in E$, for $i = 1, 2, \ldots, k$

- Weight of a path $p = (v_0, v_1, v_2, \ldots, v_k)$ is $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$

- The weight of a shortest path from s to d is given by

  $\delta(s, d) = \min \{w(p): s \xrightarrow{p} d$ if there is a path from s to $d\}$

  $= \infty$ otherwise

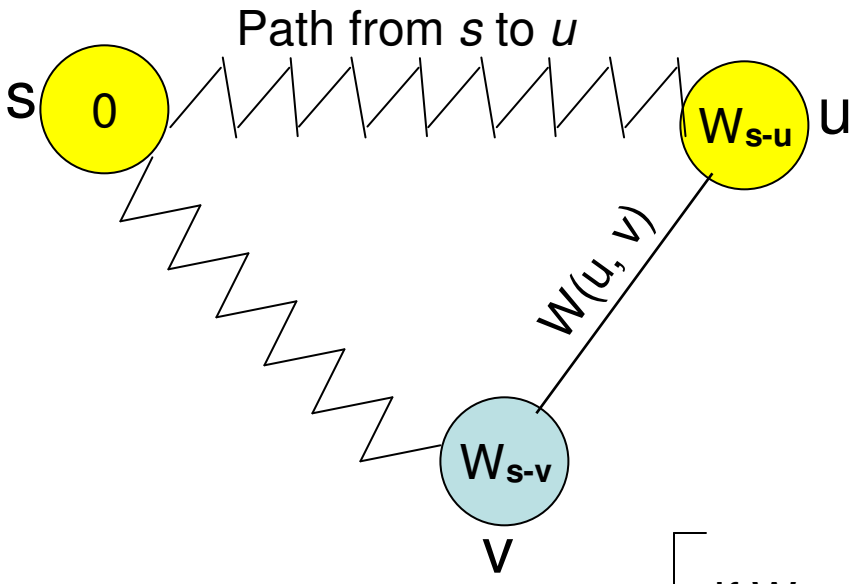
Examples of shortest path-finding algorithms:

- Dijkstra algorithm – $\Theta(|E|*log|V|)$
- Bellman-Ford algorithm – $\Theta(|E|*|V|)$

Dijkstra Algorithm

- **Assumption:** $w(u, v) >= 0$ for each edge $(u, v) \in E$ (i.e., the edge weights are positive)
- **Objective:** Given $G = (V, E, w)$, find the shortest weight path between a given source s and destination d
- **Principle:** Greedy strategy
- Maintain a minimum weight path estimate $d[v]$ from s to each other vertex v.
- At each step, pick the vertex that has the smallest minimum weight path estimate
- **Output:** After running this algorithm for $|V|$ iterations, we get the shortest weight path from s to all other vertices in G
- Note: Dijkstra algorithm does not work for graphs with edges (other than those leading from the source) with negative weights.

Principle of Dijkstra Algorithm



Principle in a nutshell

During the beginning of each iteration we will pick a vertex *u* that has the minimum weight path to *s*. We will then explore the neighbors of *u* for which we have not yet found a minimum weight path. We will try to see if by going through *u*, we can reduce the weight of path from *s* to *v*, where *v* is a neighbor of *u*.

Relaxation Condition

If $W_{s-v} > W_{s-u} + W(u, v)$ then
    $W_{s-v} = W_{s-u} + W(u, v)$
    Predecessor (v) = u
else
    Retain the current path from *s* to *v*

Note: Sub-path of a shortest path is also a shortest path. For example, if *s* – *a* – *c* – *f* – *g* – *d* is the minimum weight path from *s* to *d*, then *c* – *f* – *g* – *d* and *a* – *c* – *f* – *g* are the minimum weight paths from *c* to *d* and *a* to *g* respectively.

Dijkstra Algorithm

```
Begin Algorithm Dijkstra (G, s)
1    For each vertex v Є V
2        d [v] ← ∞ // an estimate of the min-weight path from s to v
3    End For
4    d [s] ← 0
5    S ← Φ  // set of nodes for which we know the min-weight path from s
6    Q ←  V // set of nodes for which we know estimate of min-weight path from s
7    While Q ≠ Φ
8        u ← EXTRACT-MIN(Q)
9        S ← S U {u}
10       For each vertex v such that (u, v) Є E
11           If v ∈ Q and d [v] > d [u] + w (u, v) then
12               d [v] ← d [u] + w (u, v)
13               Predecessor (v) = u
13           End If
14       End For
15   End While
16 End Dijkstra
```

Dijkstra Algorithm: Time Complexity

Begin Algorithm *Dijkstra* (*G*, *s*)

1 **For** each vertex $v \in V$

2 $d[v] \leftarrow \infty$ // an estimate of the min-weight path from *s* to *v*

3 **End For**

4 $d[s] \leftarrow 0$

5 $S \leftarrow \Phi$ // set of nodes for which we know the min-weight path from *s*

6 $Q \leftarrow V$ // set of nodes for which we know estimate of min-weight path from *s*

7 **While** $Q \neq \Phi$

8 $u \leftarrow$ EXTRACT-MIN(Q)

9 $S \leftarrow S \cup \{u\}$

10 **For** each vertex v such that $(u, v) \in E$

11 **If** $v \in Q$ and $d[v] > d[u] + w(u, v)$ then

12 $d[v] \leftarrow d[u] + w(u, v)$

13 Predecessor (v) = u

13 **End If**

14 **End For**

15 **End While**

16 **End** *Dijkstra*

Θ(V) time

Θ(V) time to Construct a Min-heap

done |V| times = Θ(V) time

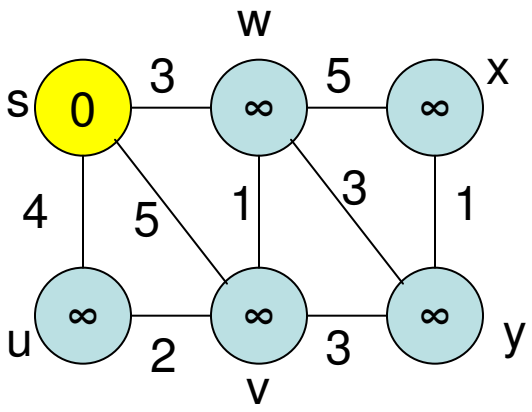
Each extraction takes Θ(logV) time

done Θ(E) times totally

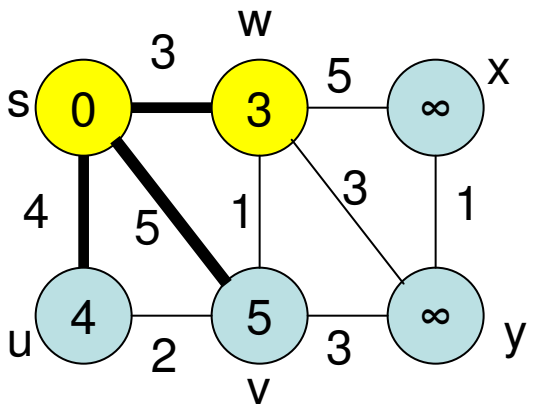
It takes Θ(logV) time when done once

Overall Complexity: Θ(V) + Θ(V) + Θ(VlogV) + O(ElogV)
Since |V| = O(|E|), the VlogV term is dominated by the ElogV term. Hence, overall complexity = O(|E|*log|V|)

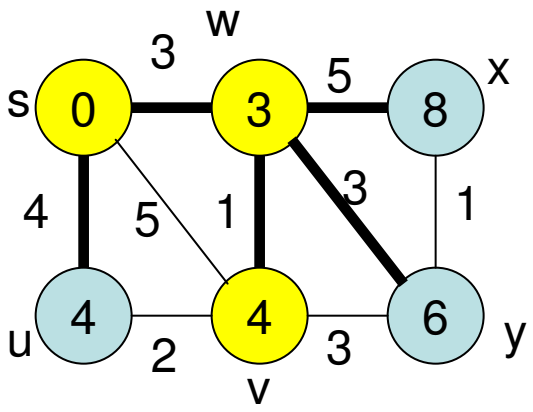
Dijkstra Algorithm (Example 1)



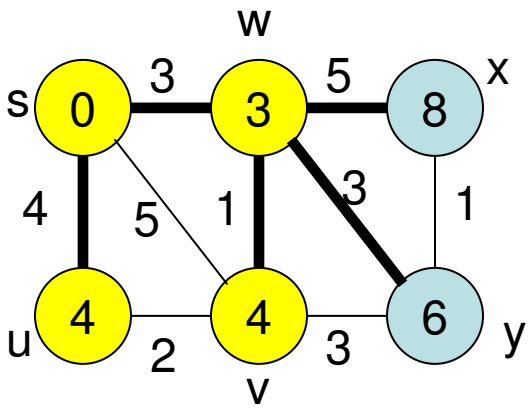
Given Graph, Initialization



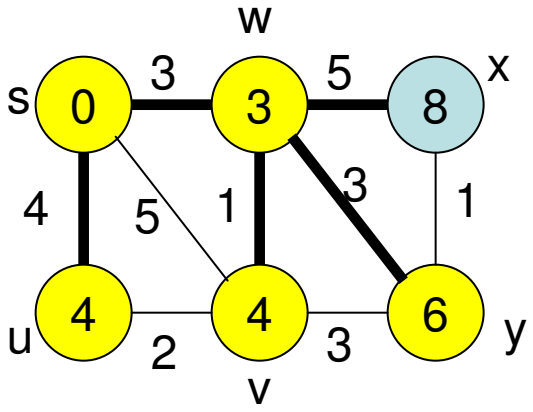
Iteration 1



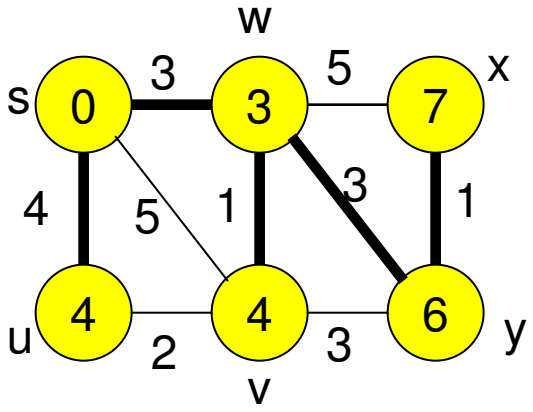
Iteration 2



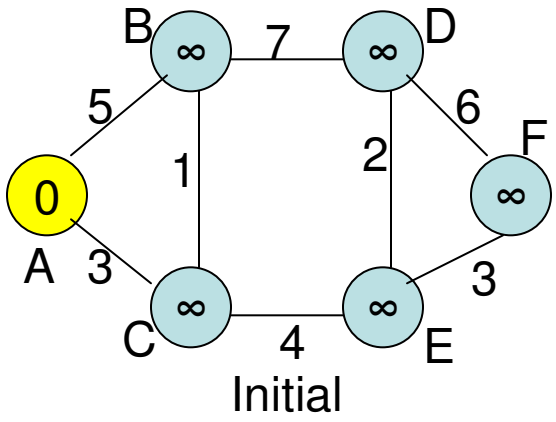
Iteration 3



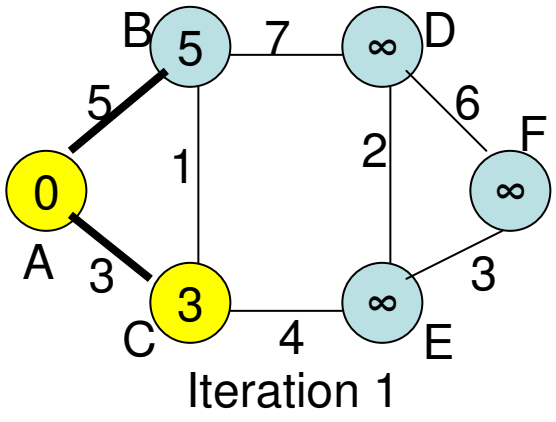
Iteration 4



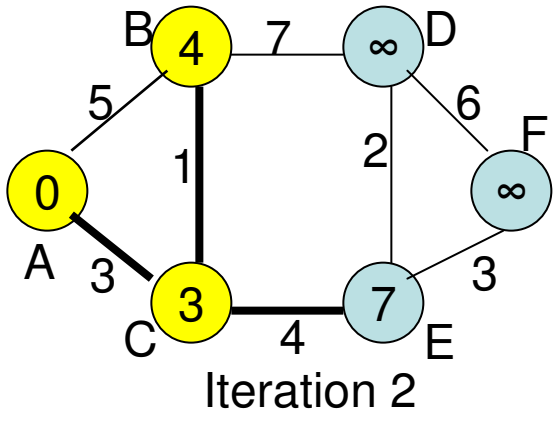
Iteration 5



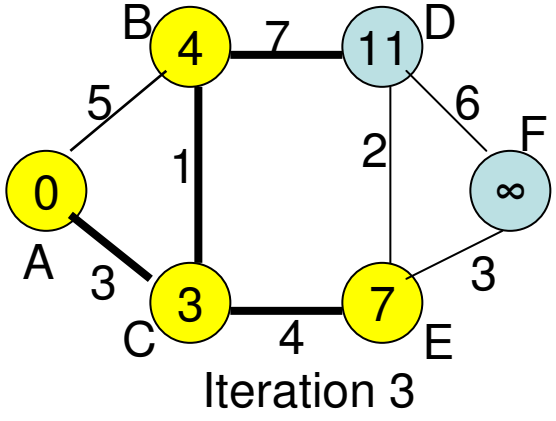
Initial



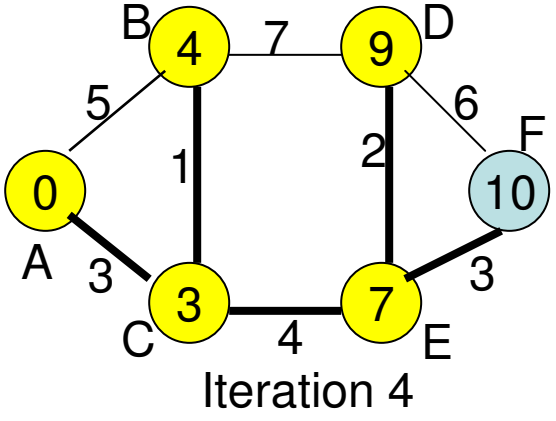
Iteration 1



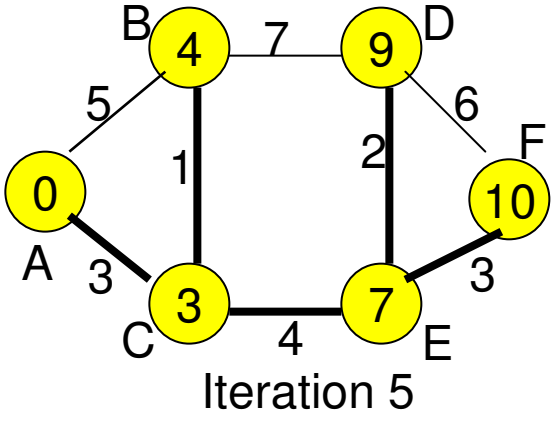
Iteration 2



Iteration 3

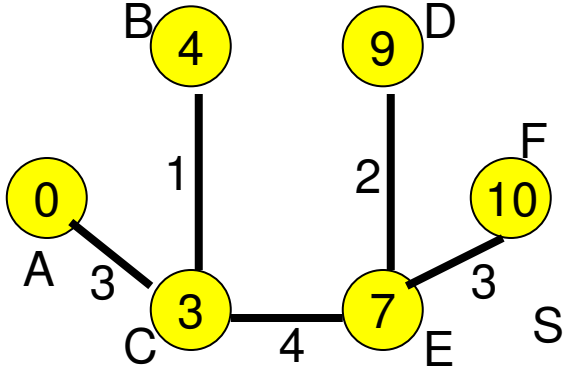


Iteration 4



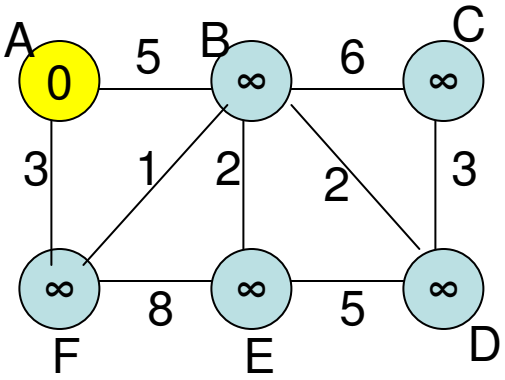
Iteration 5

Dijkstra Algorithm Example 2

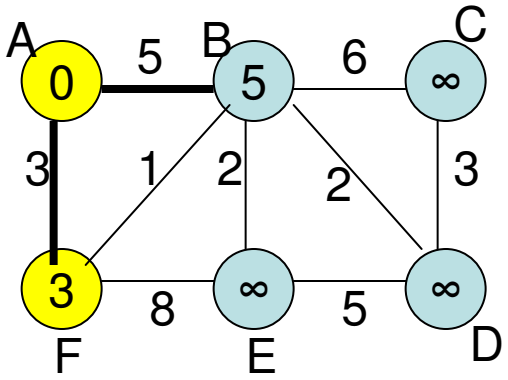


Shortest Path Tree

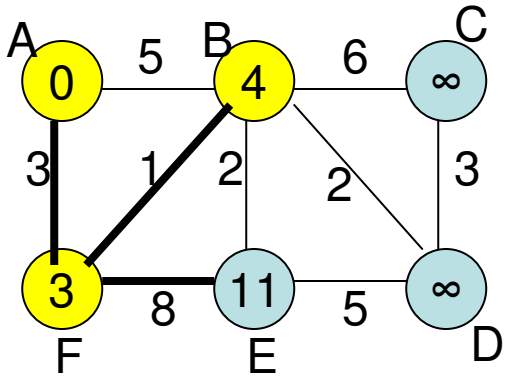
Dijkstra Algorithm
Example 3



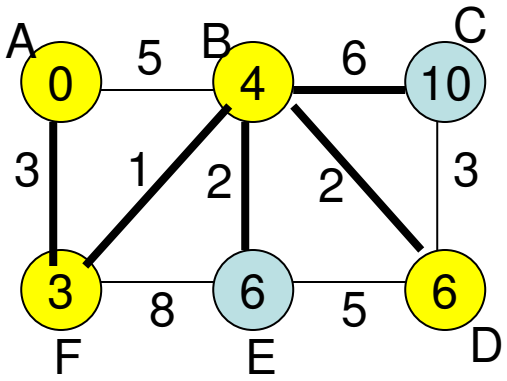
Initial



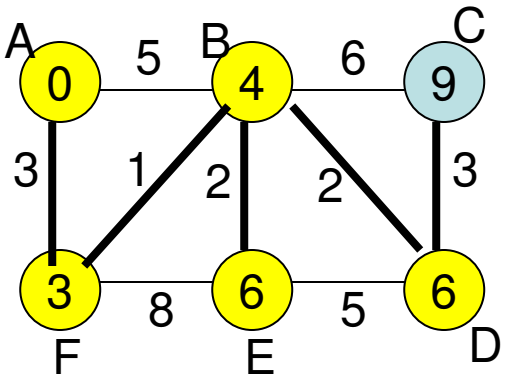
Iteration 1



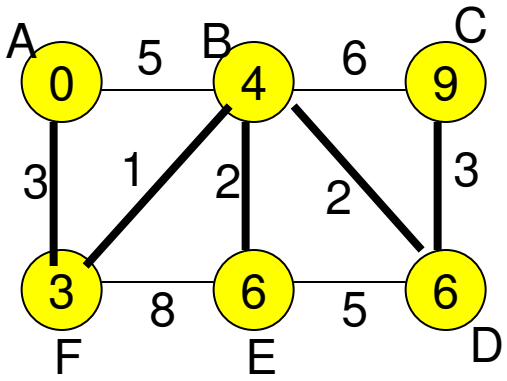
Iteration 2



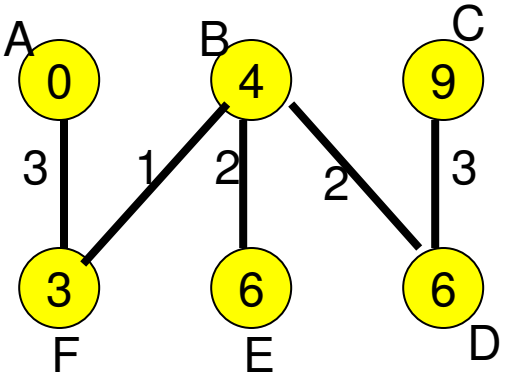
Iteration 3



Iteration 4



Iteration 5



Shortest Path Tree

Theorems on Shortest Paths and Dijsktra Algorithm

- **Theorem 1:** Sub path of a shortest path is also shortest.
- **Proof:** Lets say there is a shortest path from s to d through the vertices s – a – b – c – d.
- Then, the shortest path from a to c is also a – b – c.
- If there is a path of lower weight than the weight of the path from a – b – c, then we could have gone from s to d through this alternate path from a to c of lower weight than a – b – c.
- However, if we do that, then the weight of the path s – a – b – c – d is not the lowest and there exists an alternate path of lower weight.
- This contradicts our assumption that s – a – b – c – d is the shortest (lowest weight) path.

- **Theorem 2:** When a vertex v is picked for relaxation/optimization, every intermediate vertex on the s…v shortest path is already optimized.
- **Proof:** Let there be a path from s to v that includes a vertex *x* (i.e., s...x...v) for which we have not yet found the shortest path. From Theorem 1, weight(s...x) < weight(s...v). Also, the x...v path has to have edges of positive weight. Then, the Dijkstra's algorithm would have picked up *x* ahead of *v*. So, the fact that we picked *v* as the vertex with the lowest weight among the remaining vertices (yet to be optimized) implies that every intermediate vertex on the *s...v* is already optimized.

Theorems on Shortest Paths and Dijsktra Algorithm

- **Theorem 3:** The weights of the vertices that are optimized are in the non-decreasing (i.e., typically increasing) order.
- **Proof:** We want to prove that if a vertex u is optimized in an earlier iteration (say iteration i), then the weight of the vertex v optimized at a later iteration (say iteration j) is always larger than that of vertex u.
- Whenever a vertex u is optimized, we relax its neighbors and explore whether the weight of the shortest paths from the source to these neighbors could be decreased by going through u. As the edge weights are positive, even if we reduce the weights of the shortest paths to the neighbors, the weights of these paths are still going to be larger than the weight of the shortest path to u.

- **Theorem 4:** When a vertex v is picked for relaxation, we have optimized the vertex (i.e., found the shortest path for the vertex from a source vertex s).
- **Proof:** Let P be the path from source s to vertex v based on whose weight we decide to relax the vertex. We want to prove P is the optimal path of minimum weight from s to v. We will prove this by contradiction.

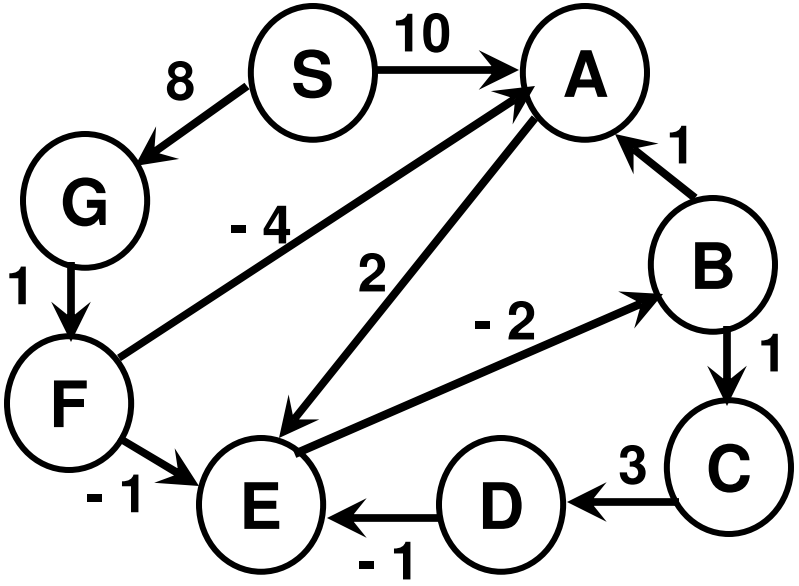
Theorems on Shortest Paths and Dijsktra Algorithm

- Let P’ be a path from s to v such that w(P’) < w(P)
- If that is the case, (as edge weights are positive), the weight of every intermediate vertex on the path P’ from s to v should be lower than the weight of vertex v on the path P.
- This implies that the weight of vertex v on the path P’ should be lower than the weight of vertex v on the path P and vertex v would have been picked for relaxation based on path P’ and not path P.
- The fact that vertex v was picked for relaxation based on path P and not P’ indicates that there exists at least one intermediate vertex on the path P’ that has not been yet relaxed. However, the weight of this intermediate vertex (even after relaxation) is going to be larger than that of vertex v on the path P as the vertices are only relaxed in the non-decreasing order of their weights. Hence, the weight of path P’ is going to be only larger than that of path P.

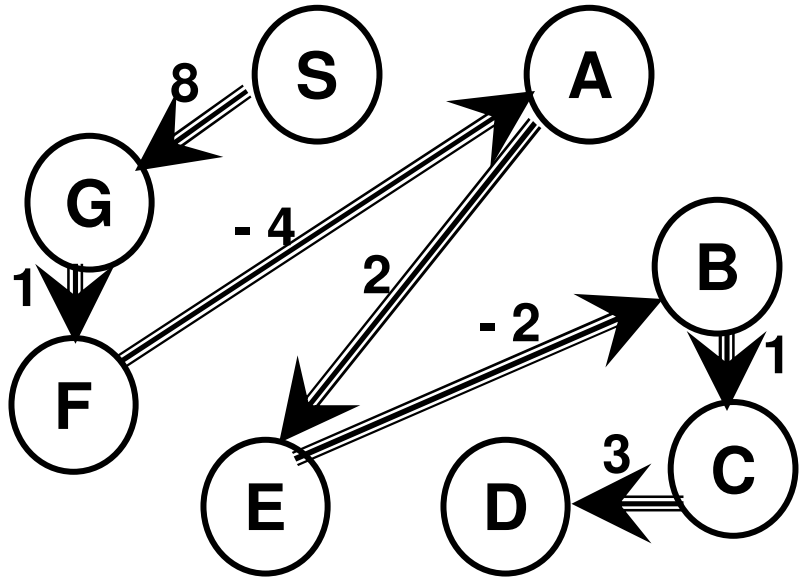
Bellman-Ford Algorithm

- Used to find single source shortest path trees on any graph (including those with negative weight edges).
- Starts with an estimate of 0 for the source and ∞ as estimates for the shortest path weights from the source to every other vertex; we proceed in iterations:
  - In each iteration, we relax every vertex (instead of just one vertex, as in Dijkstra) and try to improve the estimates of the shortest path weights to the neighbors
  - We do not target to optimize any particular vertex in an iteration; but since there cannot be more than V – 1 intermediate edges on path from the source to any vertex, we proceed for a total of $V - 1$ iterations for a graph of V vertices.
  - The time complexity is Θ(VE)
  - Optimization: In a particular iteration, if the estimates of the shortest path weights does not change for even one vertex, then we could stop!

Bellman-Ford Algorithm Example 1

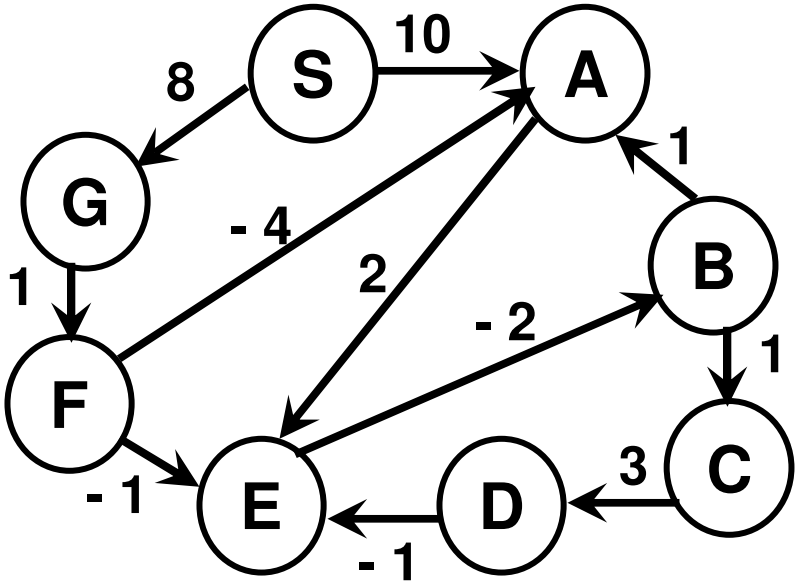


		Iterations						
Vertex	Initial	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8



		Iterations						
Vertex	Initial	1	2	3	4	5	6	7
S	-	-	-	-	-	-	-	-
A	-	S	S	F	F	F	F	F
B	-	-	-	E	E	E	E	E
C	-	-	-	-	B	B	B	B
D	-	-	-	-	-	C	C	C
E	-	-	A	F	A	A	A	A
F	-	-	G	G	G	G	G	G
G	-	S	S	S	S	S	S	S

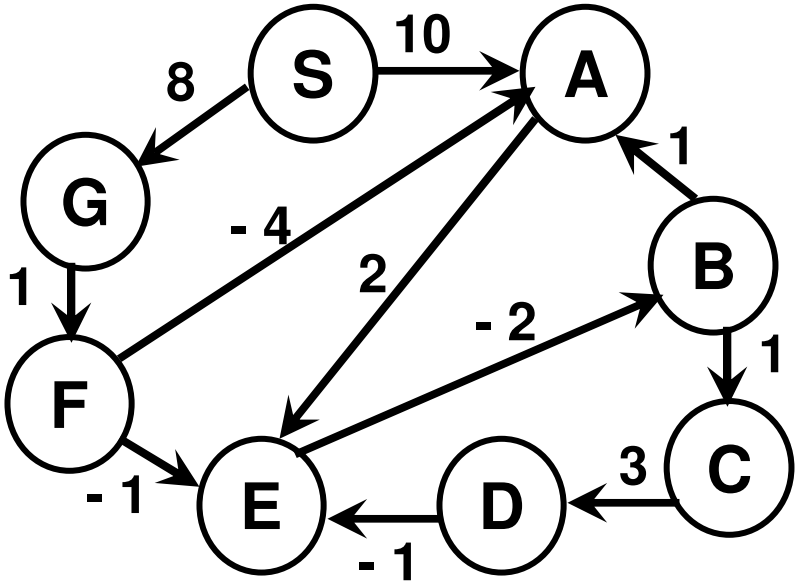
Example 1 Details



Vertex	Initial	Iterations
S	0	
A	∞	
B	∞	
C	∞	
D	∞	
E	∞	
F	∞	
G	∞	

Vertex	Initial	Iterations
S	-	
A	-	
B	-	
C	-	
D	-	
E	-	
F	-	
G	-	

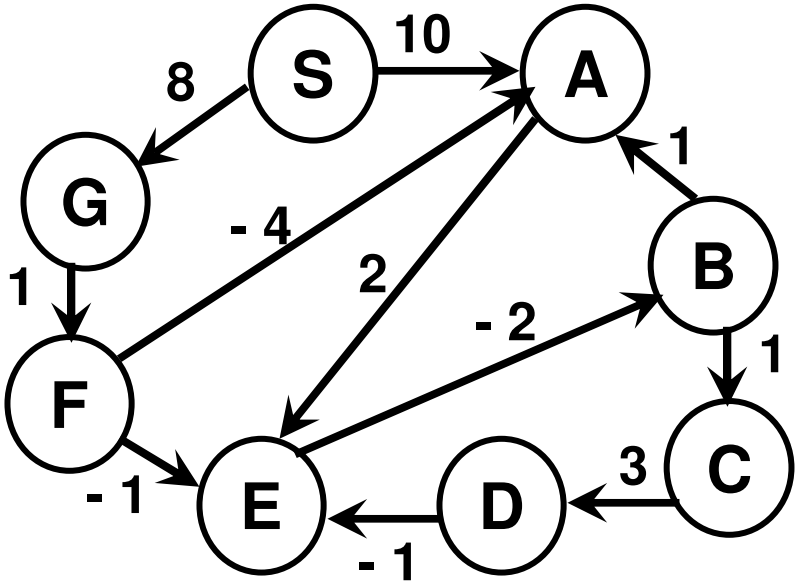
Example 1 Details



Vertex	Initial	Iterations: 1
S	0	0
A	∞	10
B	∞	∞
C	∞	∞
D	∞	∞
E	∞	∞
F	∞	∞
G	∞	8

Vertex	Initial	Iterations: 1
S	-	-
A	-	S
B	-	-
C	-	-
D	-	-
E	-	-
F	-	-
G	-	S

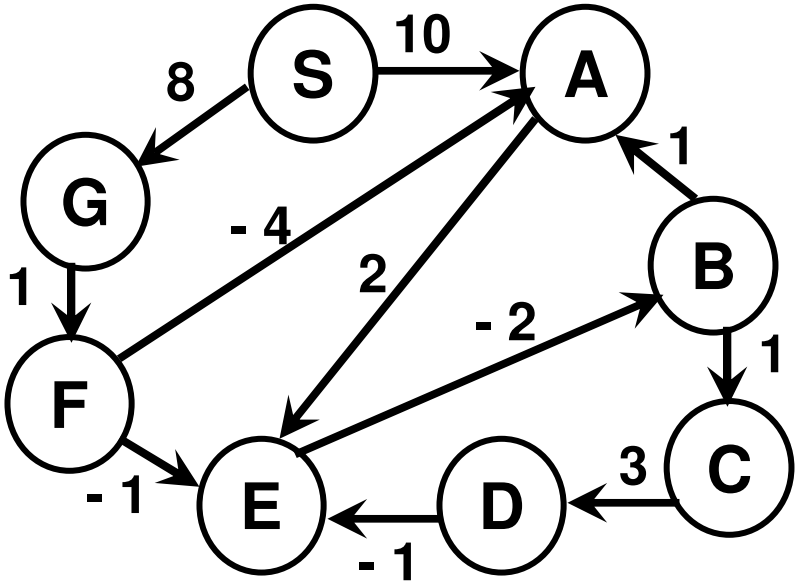
Example 1 Details



		Iterations	
Vertex		1	2
S		0	0
A		10	10
B		∞	∞
C		∞	∞
D		∞	∞
E		∞	12
F		∞	9
G		8	8

		Iterations	
Vertex		1	2
S		-	-
A		S	S
B		-	-
C		-	-
D		-	-
E		-	A
F		-	G
G		S	S

Example 1 Details



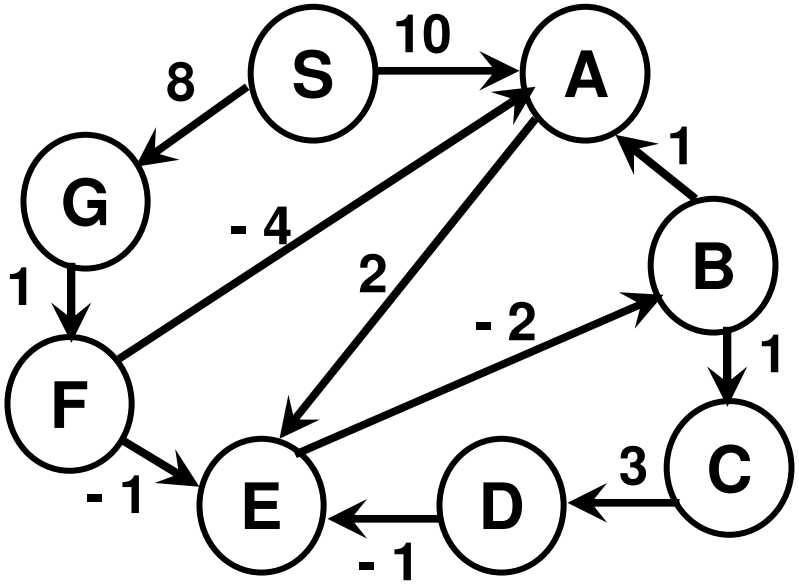
Vertex	2	3
S	0	0
A	10	5
B	∞	10
C	∞	∞
D	∞	∞
E	12	8
F	9	9
G	8	8

Iterations

Vertex	2	3
S	-	-
A	S	F
B	-	E
C	-	-
D	-	-
E	A	F
F	G	G
G	S	S

Iterations

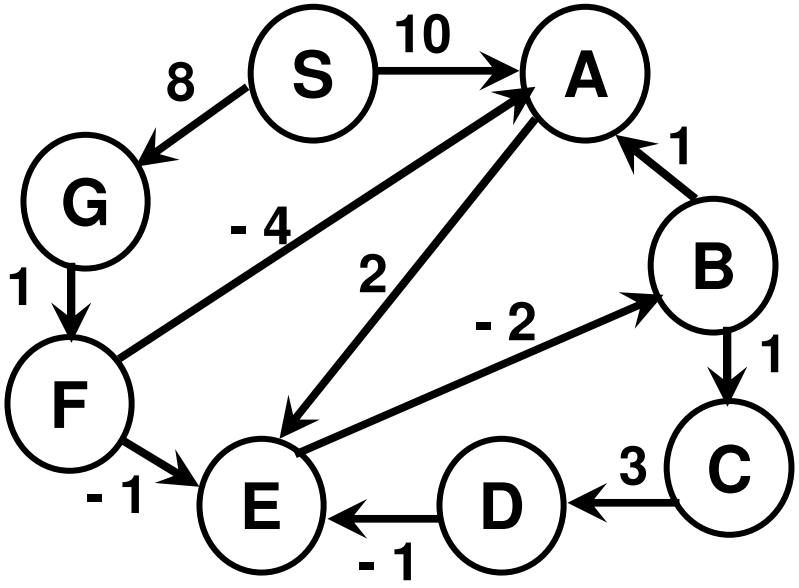
Example 1 Details



	Iterations	
Vertex	3	4
S	0	0
A	5	5
B	10	6
C	∞	11
D	∞	∞
E	8	7
F	9	9
G	8	8

	Iterations	
Vertex	3	4
S	-	-
A	F	F
B	E	E
C	-	B
D	-	-
E	F	A
F	G	G
G	S	S

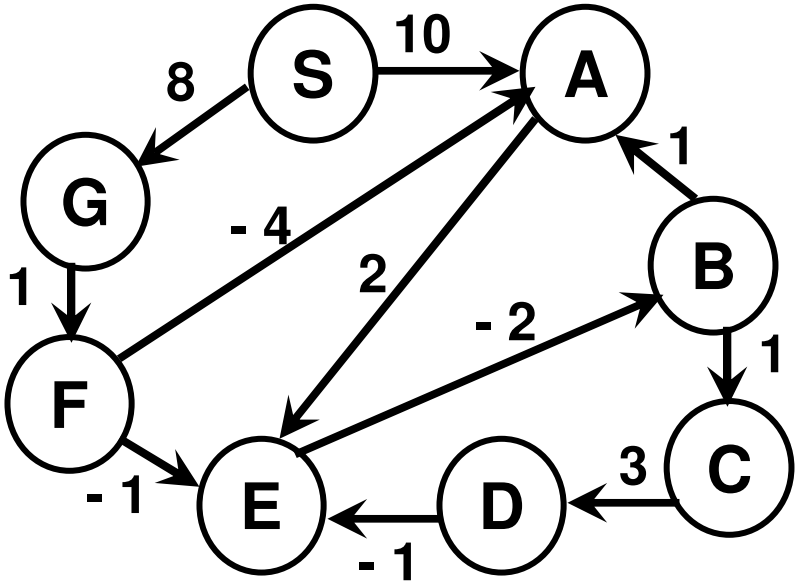
Example 1 Details



	Iterations	
Vertex	4	5
S	0	0
A	5	5
B	6	5
C	11	7
D	∞	14
E	7	7
F	9	9
G	8	8

	Iterations	
Vertex	4	5
S	-	-
A	F	F
B	E	E
C	B	B
D	-	C
E	A	A
F	G	G
G	S	S

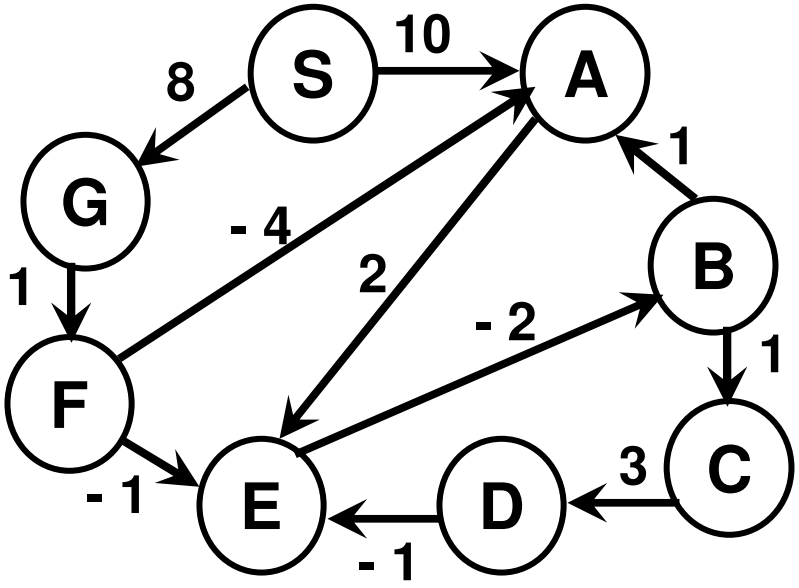
Example 1 Details



	Iterations	
Vertex	5	6
S	0	0
A	5	5
B	5	5
C	7	6
D	14	10
E	7	7
F	9	9
G	8	8

	Iterations	
Vertex	5	6
S	-	-
A	F	F
B	E	E
C	B	B
D	C	C
E	A	A
F	G	G
G	S	S

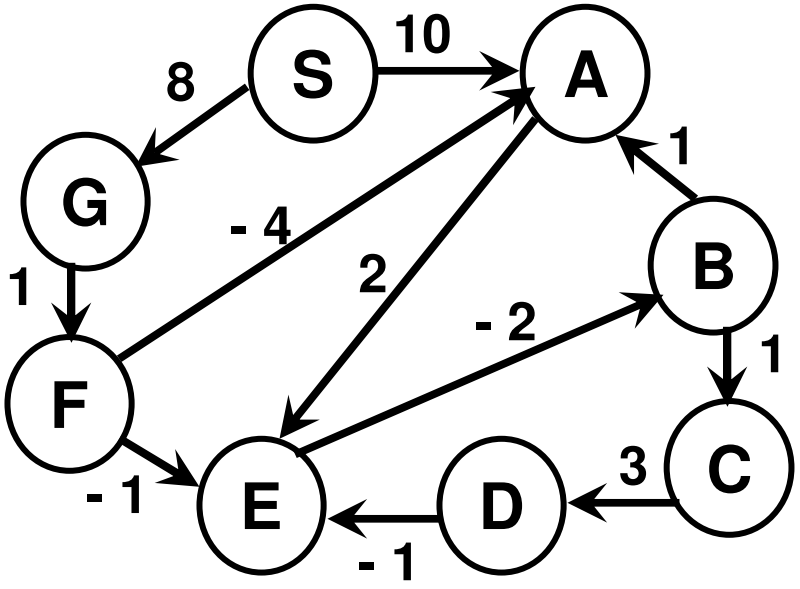
Example 1 Details



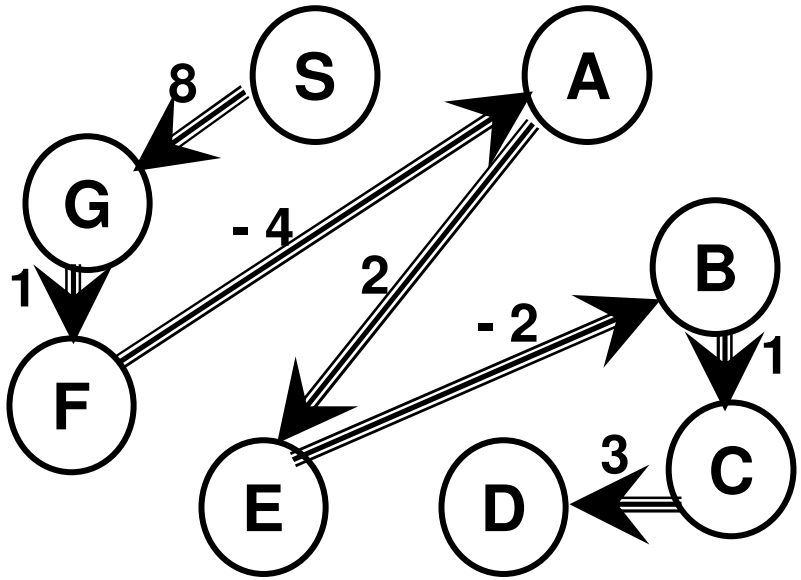
		Iterations	
Vertex		6	7
S		0	0
A		5	5
B		5	5
C		6	6
D		10	9
E		7	7
F		9	9
G		8	8

		Iterations	
Vertex		6	7
S		-	-
A		F	F
B		E	E
C		B	B
D		C	C
E		A	A
F		G	G
G		S	S

Example 1 Details

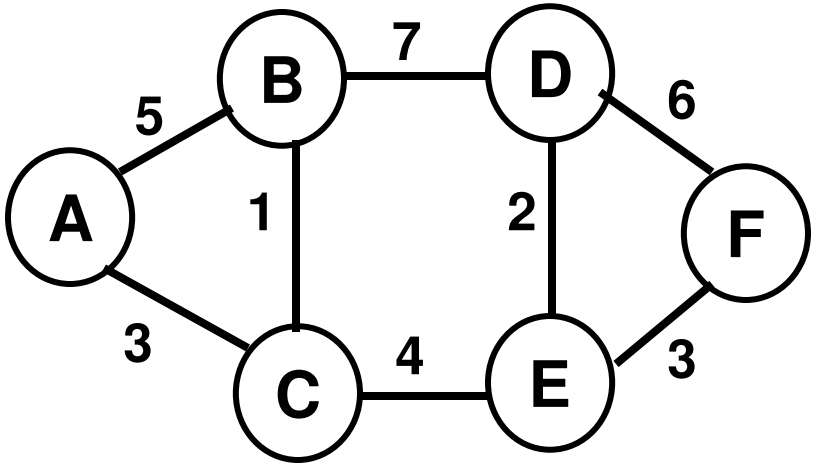


		Iterations						
Vertex	Initial	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

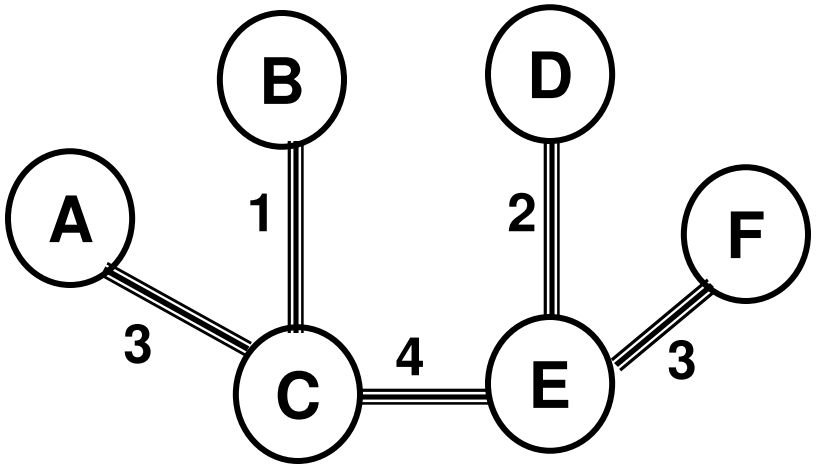


		Iterations						
Vertex	Initial	1	2	3	4	5	6	7
S	-	-	-	-	-	-	-	-
A	-	S	S	F	F	F	F	F
B	-	-	-	E	E	E	E	E
C	-	-	-	-	B	B	B	B
D	-	-	-	-	-	C	C	C
E	-	-	A	F	A	A	A	A
F	-	-	G	G	G	G	G	G
G	-	S	S	S	S	S	S	S

Bellman-Ford Algorithm Example 2



		Iterations					
Vertex	Initial	1	2	3	4	5	6
A	0	0	0	0	0	S	
B	∞	5	4	4	4	T	
C	∞	3	3	3	3	O	
D	∞	∞	12	9	9	P	
E	∞	∞	7	7	7	!	
F	∞	∞	∞	10	10	!	



		Iterations					
Vertex	Initial	1	2	3	4	5	6
A	-	-	-	-	-	S	
B	-	A	C	C	C	T	
C	-	A	A	A	A	O	
D	-	-	B	E	E	P	
E	-	-	C	C	C	!	
F	-	-	-	E	E	!	

Minimum Spanning Trees

Minimum Spanning Tree Problem

- Given a weighted graph, we want to determine a tree that spans all the vertices in the tree and the sum of the weights of all the edges in such a spanning tree should be minimum.
- Two algorithms:
  - Prim algorithm: $\Theta(|E|*log|V|)$,
  - Kruskal Algorithm: $\Theta(|E|*log\ |E| + |V|*log|V|))$

- Prim algorithm is just a variation of Dijkstra algorithm with the relaxation condition being

If v∈ Q and d [v] > w (u, v) then
    d [v] ← w (u, v)
    Predecessor (v) = u
End If

On each iteration, the algorithm expands the current tree in a greedy manner by attaching to it the nearest vertex not in the tree. By the 'nearest vertex', we mean a vertex not in the tree connected to a vertex in the tree by an edge of the smallest weight.

- Kruskal algorithm: Consider edges in the increasing order of their weights and include an edge in the tree, if and only if, by including the edge in the tree, we do not create a cycle!!
- Note: Shortest Path trees need not always have minimum weight and minimum spanning trees need not always be shortest path trees.

Kruskal Algorithm

```
Begin Algorithm Kruskal (G = (V, E))
    A ← Φ // Initialize the set of edges to null set

    for each vertex v_i ϵ V do
        Component (v_i) ← i

    Sort the edges of E in the non-decreasing (increasing) order of weights

    for each edge (v_i, v_j) ϵ E, in order by non-decreasing weight do
        if (Component (v_i) ≠ Component (v_j) ) then
            A ← A U (v_i, v_j)

            if Component(v_i) < Component(v_i)  then
              for each vertex v_k in the same component as of v_j do
                Component(v_k) ← Component(v_i)
            else
              for each vertex v_k in the same component as of v_i do
                Component(v_k) ← Component(v_j )
            end if
        end if
    end for
    return A
End Algorithm Kruskal
```

Kruskal Algorithm: Time Complexity

Begin Algorithm *Kruskal* ($G = (V, E)$)

$A \leftarrow \Phi$ // Initialize the set of edges to null set

for each vertex $v_i \in V$ do
  Component (v_i) $\leftarrow i$

$\Theta(V)$ time

Components are kept track of using a Disjoint-set data structure

Sort the edges of E in the non-decreasing (increasing) order of weights

$\Theta(E \log E)$ time

for each edge $(v_i, v_j) \in E$, in order by non-decreasing weight do

  if (Component (v_i) ≠ Component (v_j)) then

    A ← A U (v_i, v_j)

Can be done in $\Theta(\log V)$ time

    if Component(v_i) < Component(v_i) then
      for each vertex v_k in the same component as of v_j do
        Component(v_k) ← Component(v_i)
    else
      for each vertex v_k in the same component as of v_i do
        Component(v_k) ← Component(v_j)
    end if

Takes $\Theta(\log V)$ time per merger

  end if

V-1 mergers. So, $\Theta(V \log V)$

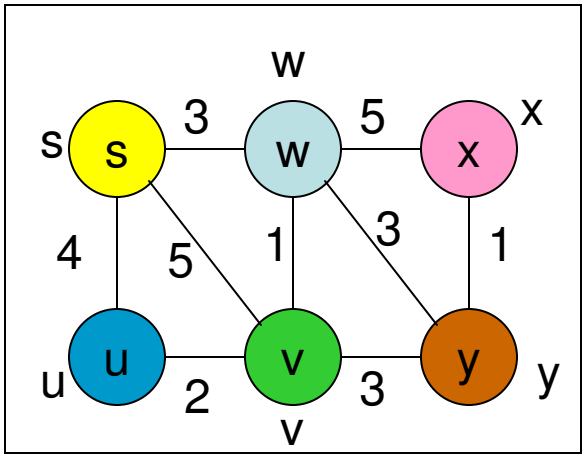
  end for

  return *A*

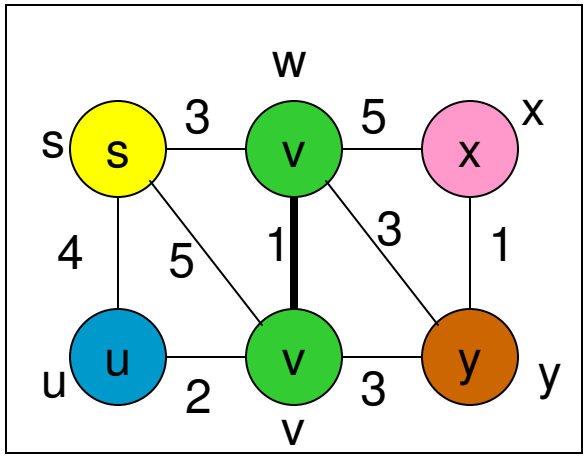
End Algorithm *Kruskal*

Overall time complexity:
$\Theta(V) + \Theta(E \log E) + \Theta(V \log V) = \Theta(V \log V + E \log E)$

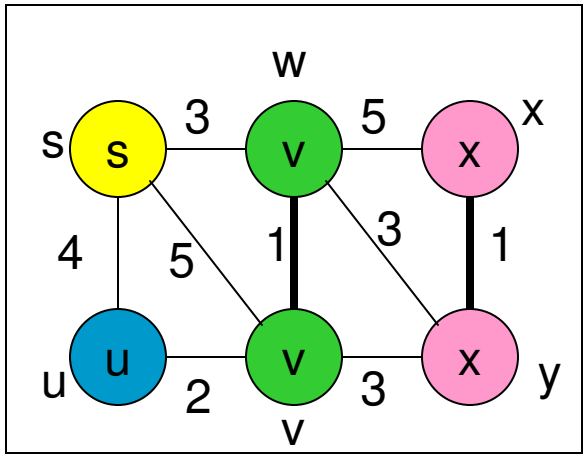
Kruskal Algorithm (Example 1)



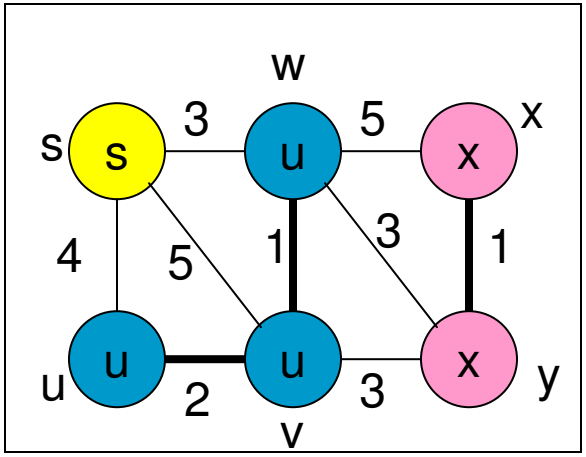
(a)



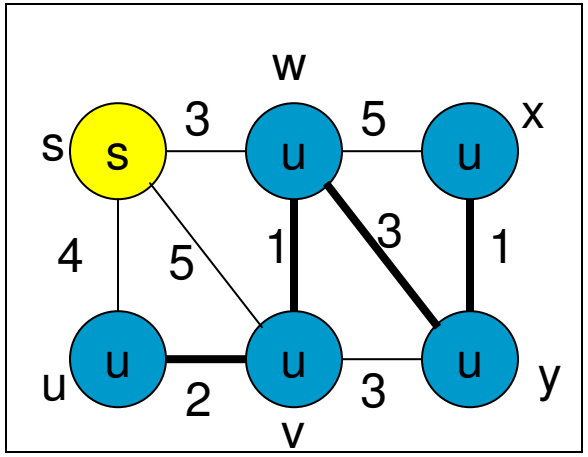
(b)



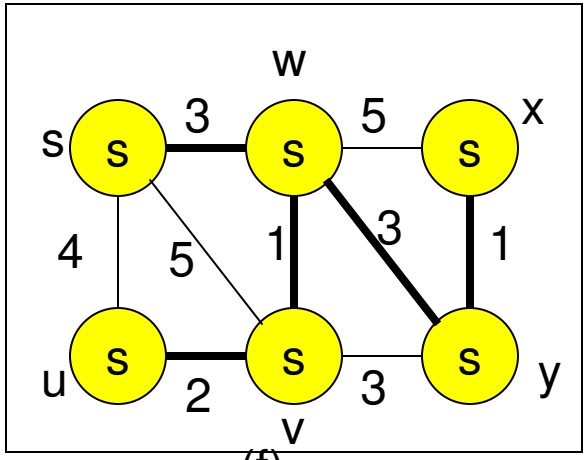
(c)



(d)

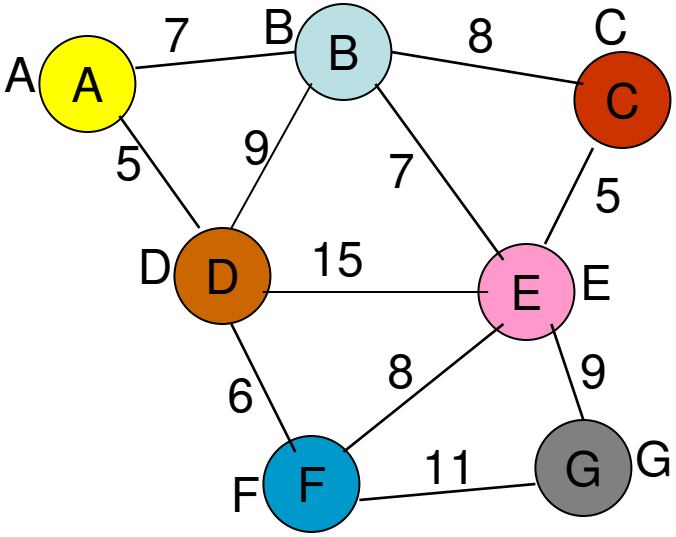


(e)

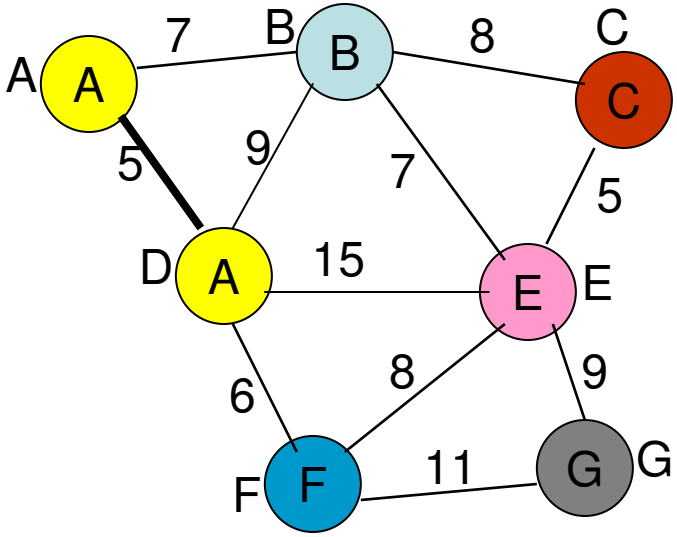


(f)

Weight of the minimum spanning tree = 10
Weight of the shortest path tree = 12

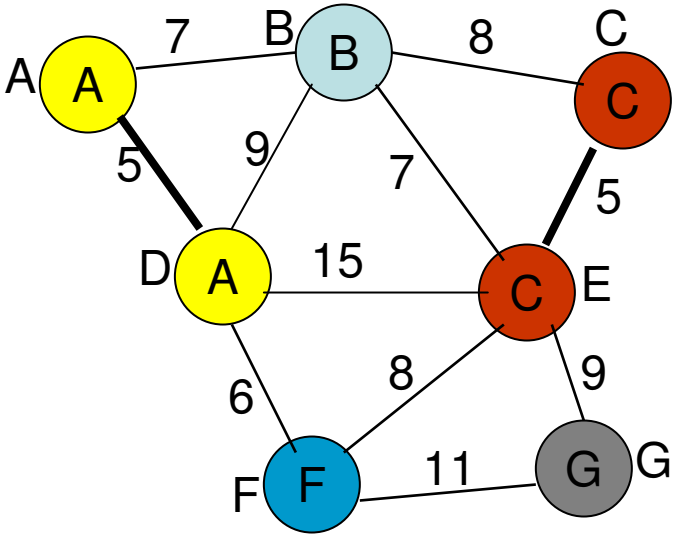


Initial

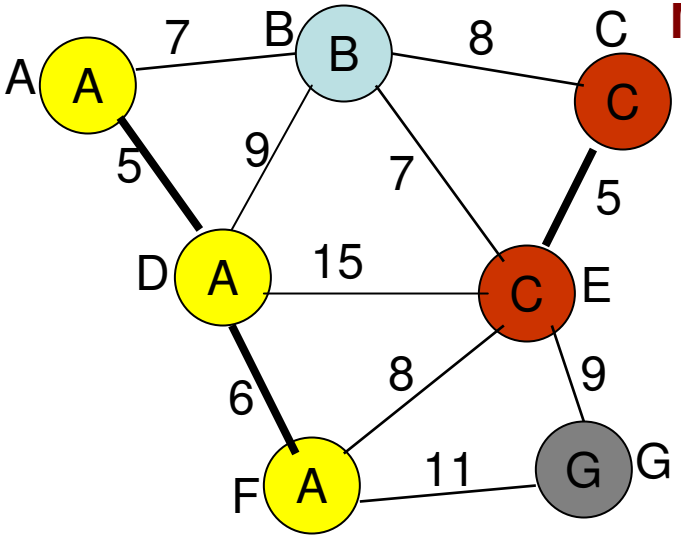


Iteration 1

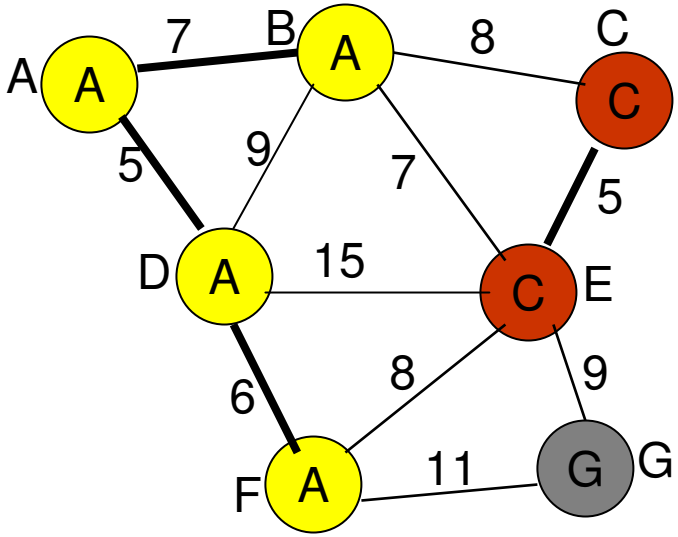
Kruskal Algorithm Example 2 for Minimum Spanning Tree



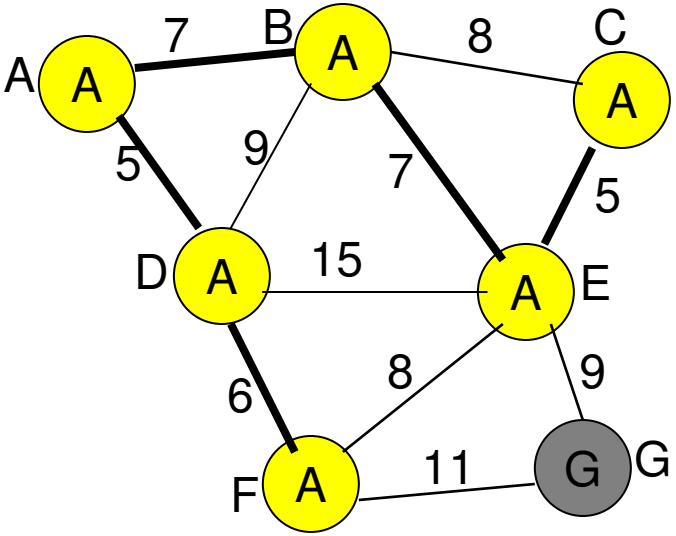
Iteration 2



Iteration 3

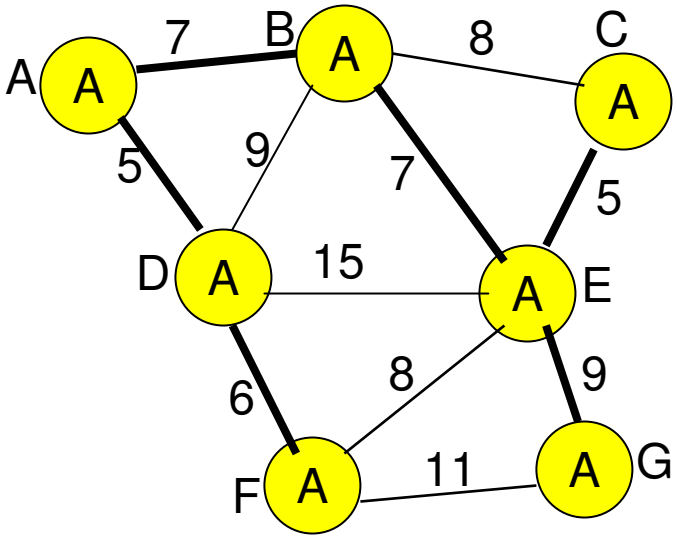


Iteration 4

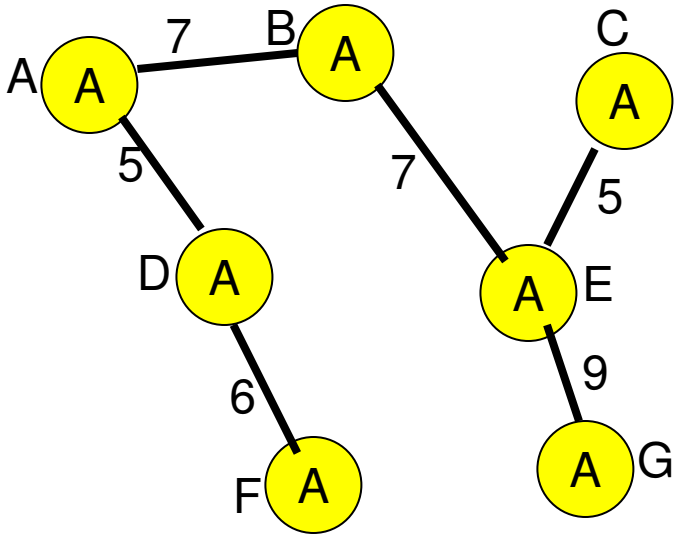


Iteration 5

Kruskal Algorithm Example 2



Iteration 6



Minimum Spanning Tree

Proof of Correctness: Kruskal's Algorithm

- Let *T* be the spanning tree generated by Kruskal's algorithm for a graph *G*. Let *T'* be a minimum spanning tree for *G*. We need to show that both *T* and *T'* have the same weight.
- Assume that wt(*T'*) < wt(*T*).
- Hence, there should be an edge *e* in *T* that is not in *T'.* Because, if every edge of *T* is in *T'*, then *T* = *T'* and wt(*T*) = wt(*T'*).
- Pick up the edge $e \in T$ and $e \notin T'$. Include *e* in *T'.* This would create a cycle among the edges in *T'.* At least one edge in this cycle would not be part of *T*; because if all the edges in this cycle are in *T*, then *T* would have a cycle.
- Pick up the edge *e'* that is in this cycle and not in *T*.
- Note that wt(*e'*) $\not<$ wt(*e*); because, if this was the case then the Kruskal's algorithm would have picked *e'* ahead of *e*. So, wt(*e'*) ≥ wt(*e*). This implies that we could remove *e'* from the cycle and include edge *e* as part of *T'* without increasing the weight of the spanning tree.
- We could repeat the above procedure for all edges that are in *T* and not in *T'* ; and eventually transform *T'* to *T*, without increasing the cost of the spanning tree.
- Hence, *T* is a minimum spanning tree.

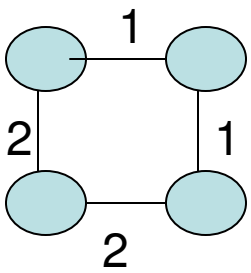
Properties of Minimum Spanning Tree

- **Property 1:** If an edge (i, j) is part of a minimum spanning tree T of a weighted graph of V vertices, its two end vertices are part of an IJ-cut and (i, j) is the minimum weight edge in the IJ-cut-set.

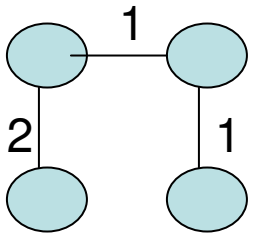
- **Proof:** We will prove this by contradiction. Assume an edge (i, j) exists in a minimum spanning tree T. Let there be an edge (i', j') of an IJ-cut such that vertices i and i' are in I and vertices j and j' are in J, and that weight(i', j') < weight(i, j). If that is the case, we can remove (i, j) from the minimum spanning tree T and restore its connectivity and spanning nature by adding (i', j') instead. By doing this, we will only lower the weight of T contradicting the assumption that T is a minimum spanning tree to start with. Hence, every edge (i, j) of a minimum spanning tree has to be the minimum weight edge in an IJ-cut such that i is in I and j is in J, and $I \cup J = V$ and $I \cap J = \phi$.

Properties of Minimum Spanning Tree

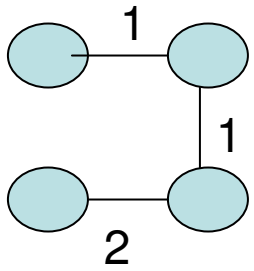
- **<u>Property 2:</u>** If a graph does not have unique edge weights, there could be more than one minimum spanning tree for the graph.
- **Proof (by Example)**



Graph



One Min. Spanning Tree



Another Min. Spanning Tree

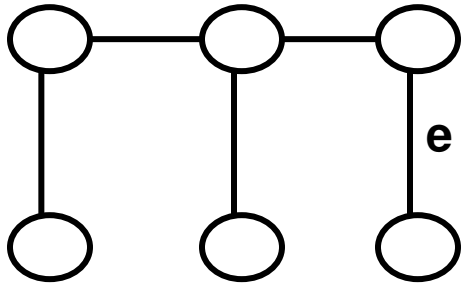
- **<u>Property 3:</u>** If all the edges in a weighted graph have unique weights, then there can be only one minimum spanning tree of the graph.
- **Proof:** Consider a graph G whose edges are of distinct weight. Assume there are two different spanning trees *T* and T', both are of minimum weight; but have at least one edge difference. That is, there should be at least one edge *e* in T and e is not in T'. Add the edge e in T' to create a cycle. This cycle should involve at least one edge e' that is not in T; because if all the edges in the cycle are in T, then T is not a tree.

Properties of Minimum Spanning Tree

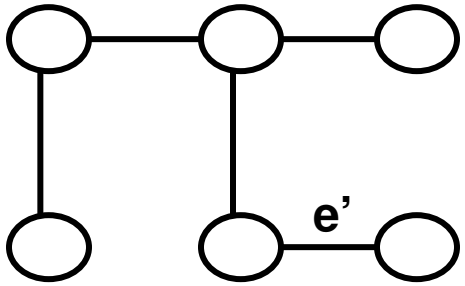
- **<u>Property 3:</u>** If all the edges in a weighted graph have unique weights, then there can be only one minimum spanning tree of the graph.
- **Proof (continued..):** Thus, the end vertices of each of the two edges, e and e', should belong to two disjoint sets of vertices that if put together will be the set of vertices in the graph.
- Since all the edges in the graph are of unique weight, the weight(e') < weight(e) for T' to be a min. spanning tree. However, if that is the case, the weight of T can be reduced by removing e and including e', lowering the weight of T further. This contradicts our assumption that T is a min. spanning tree.
- Hence, weight(e) $\not>$ weight(e'). That is, the weight of edge *e* cannot be greater than the weight of edge *e*' for T to be a min. spanning tree. Hence, weight(e) ≤ weight(e') for T to be a min. spanning tree. Since, all edge weights are distinct, weight(e) < weight(e') for T to be a min. spanning tree.
- However, from the previous argument, we have that weight(e') < weight(e) for T' to be a min. spanning tree.
- Thus, even though the graph has unique edge weights, it is not possible to say which of the two edges (e and e') are of greater weight, if the graph has two minimum spanning trees.
- Thus, a graph with unique edge weights has to have only one minimum spanning tree.

Property 3

Assume that both T and T' are MSTs, but different MSTs to start with.

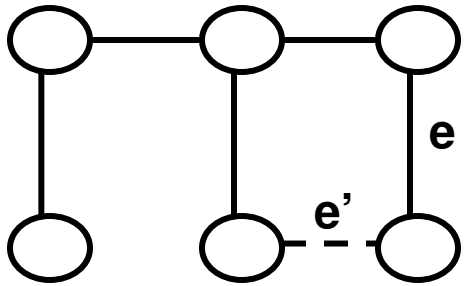


T

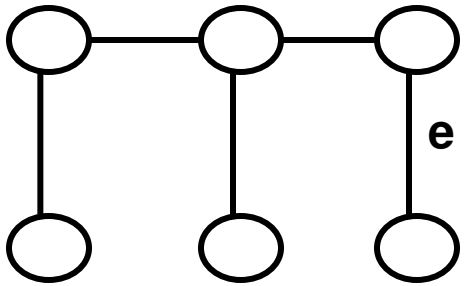


T'

W(e) < W(e') => T' is not a MST
W(e) > W(e') => T is not a MST
Hence, for both T and T' to be different MSTs ➔ W(e) = W(e').
But the graph has unique edge weights.
W(e) ≠ W(e) ➔ Both T and T' have to be the same.



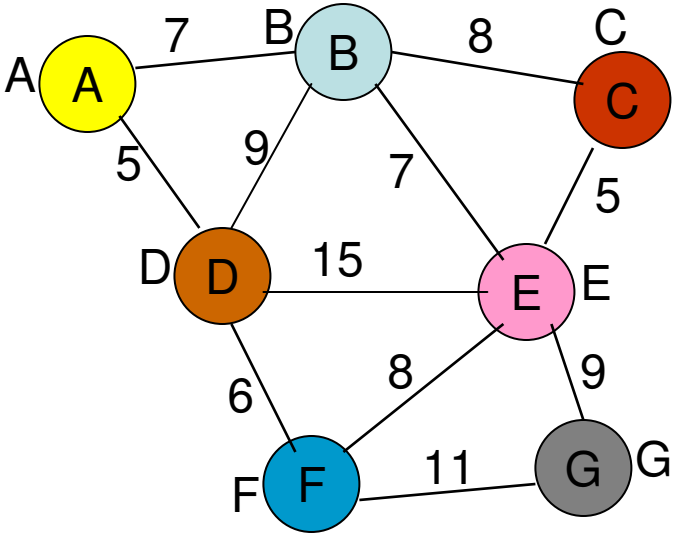
T



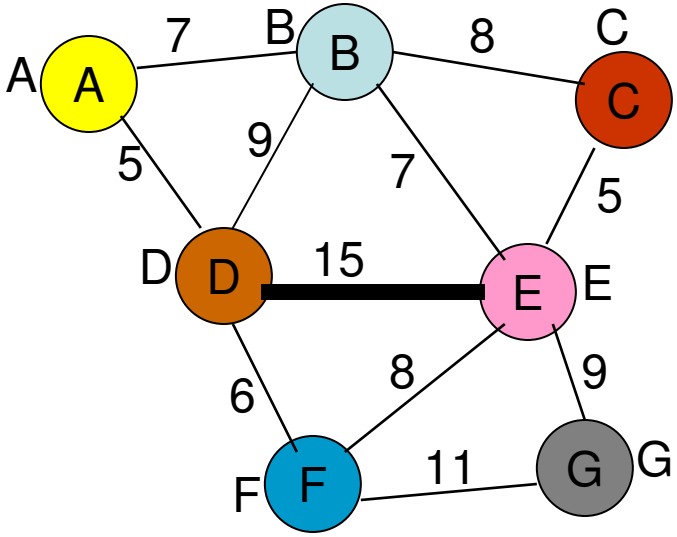
Modified T'

Maximum Spanning Tree

- A Maximum Spanning Tree is a spanning tree such that the sum of its edge weights is the maximum.
- We can find a Maximum Spanning Tree through any one of the following ways:
  - Run Kruskal's algorithm by selecting edges in the decreasing order of edge weights (i.e., edge with the largest weight is chosen first) as long as the end vertices of an edge are in two different components
  - Run Prim's algorithm by selecting the edge with the largest weight crossing from the optimal set to the fringe set
  - Given a weighted graph, set all the edge weights to be negative, run a minimum spanning tree algorithm on the negative weight graph, then turn all the edge weights to positive on the minimum spanning tree to get a maximum spanning tree.

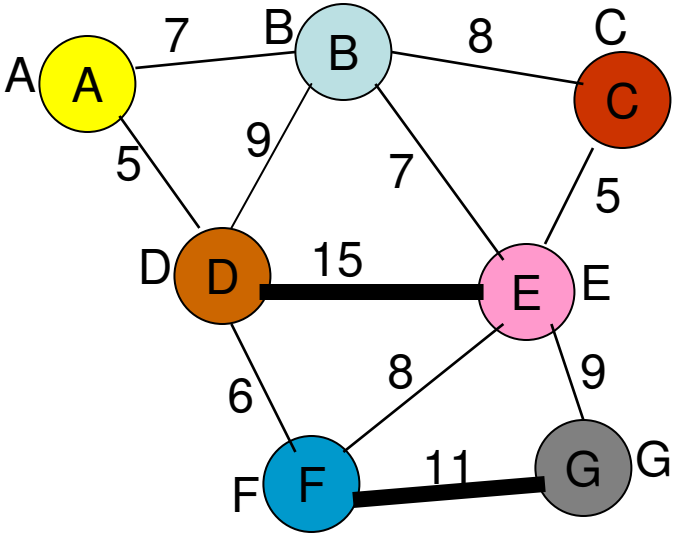


Initial

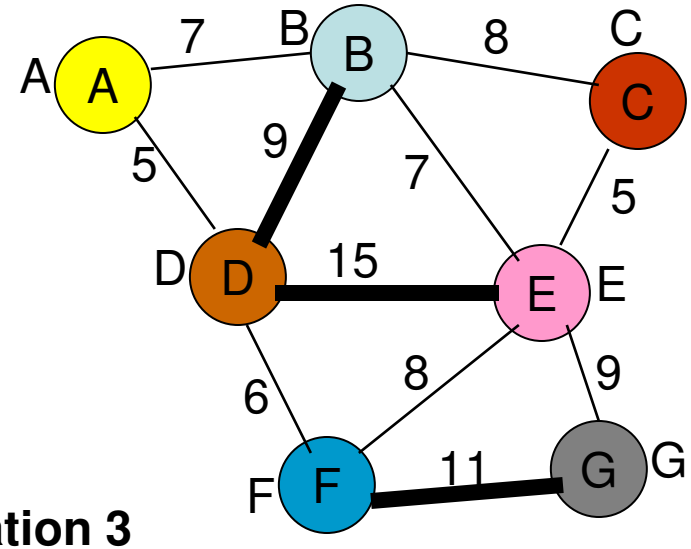


Iteration 1

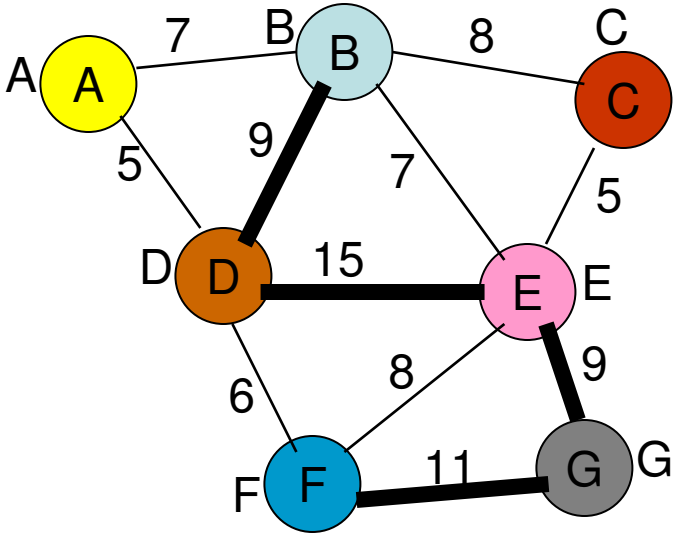
Kruskal Algorithm Example 2 for Max. Spanning Tree



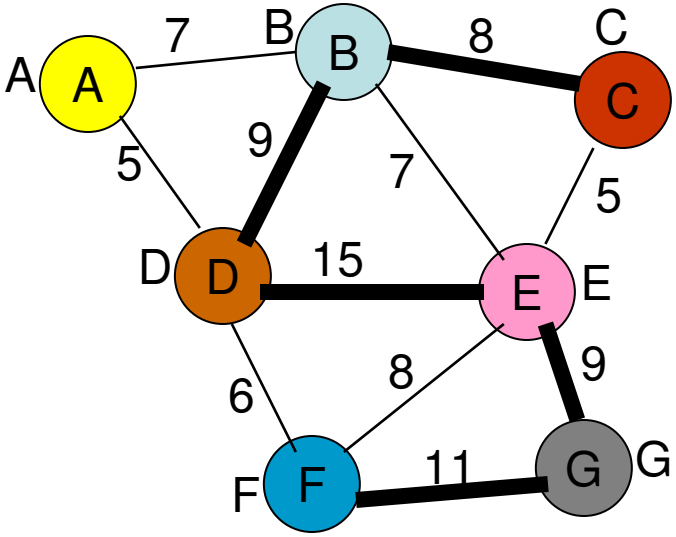
Iteration 2



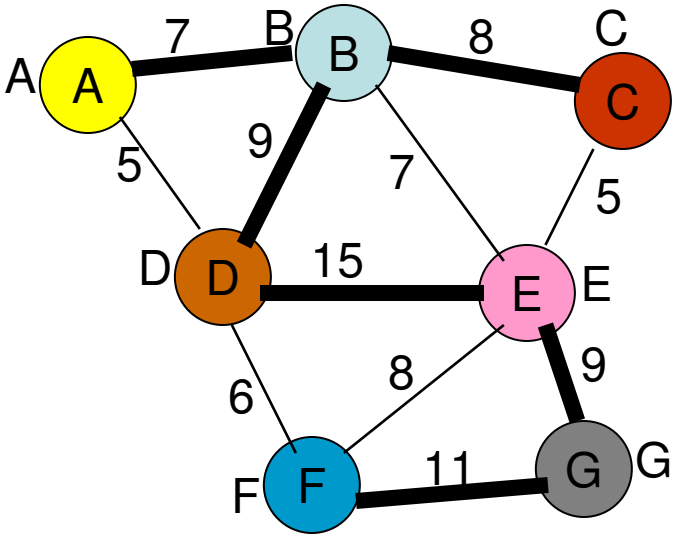
Iteration 3



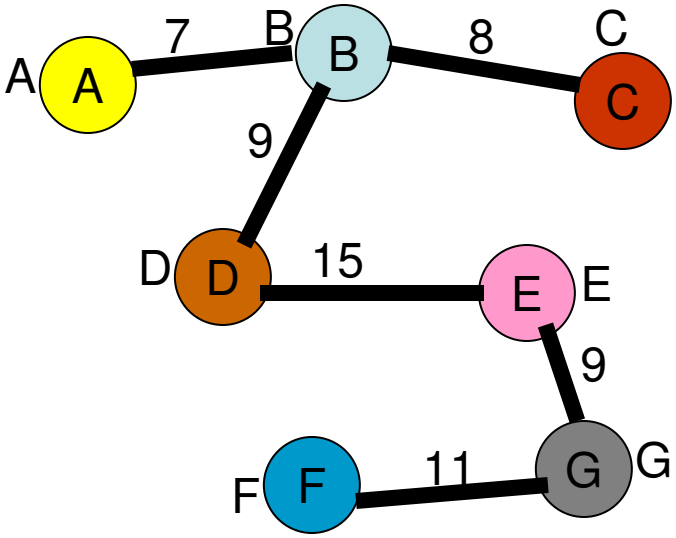
Iteration 4



Iteration 5



Iteration 6



Maximum Spanning Tree

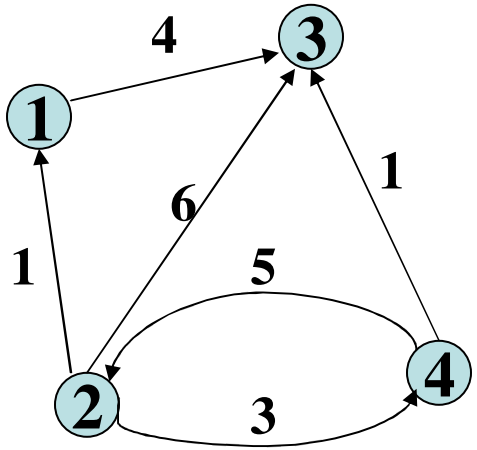
All Pairs Shortest Paths Problem

Floyd's Algorithm: All pairs shortest paths

Problem: **In a weighted (di)graph, find shortest paths between every pair of vertices**

idea: construct solution through series of matrices $D^{(0)}$, …, $D^{(n)}$ using increasing subsets of the vertices allowed as intermediate

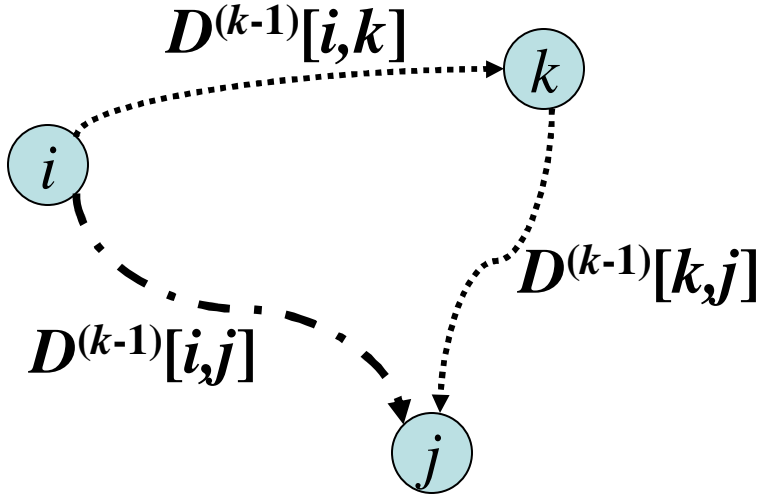
Example:



Floyd's Algorithm (matrix generation)

On the *k*-th iteration, the algorithm determines shortest paths between every pair of vertices *i, j* that use only vertices among 1,…,*k* as intermediate

$$D^{(k)}[i,j] = \min \{D^{(k-1)}[i,j],\ D^{(k-1)}[i,k] + D^{(k-1)}[k,j]\}$$

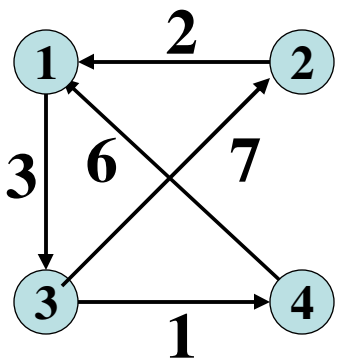


Predecessor Matrix

$$\pi_{ij}^{(0)} = \text{N/A if i} = \text{j or } w_{ij} = \infty$$
$$= \text{i if i} \neq \text{j and } w_{ij} < \infty$$

$$\pi_{ij}^{(k)} = \pi_{ij}^{(k-1)} \quad \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$$
$$= \pi_{kj}^{(k-1)} \quad \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$$

Floyd's Algorithm (example)



$D^{(0)}$

	v1	v2	v3	v4
v1	0	∞	3	∞
v2	2	0	∞	∞
v3	∞	7	0	1
v4	6	∞	∞	0

$\Pi^{(0)}$

	v1	v2	v3	v4
v1	N/A	N/A	v1	N/A
v2	v2	N/A	N/A	N/A
v3	N/A	v3	N/A	v3
v4	v4	N/A	N/A	N/A

$D^{(1)}$

	v1	v2	v3	v4
v1	0	∞	3	∞
v2	2	0	5	∞
v3	∞	7	0	1
v4	6	∞	9	0

$\Pi^{(1)}$

	v1	v2	v3	v4
v1	N/A	N/A	v1	N/A
v2	v2	N/A	v1	N/A
v3	N/A	v3	N/A	v3
v4	v4	N/A	v1	N/A

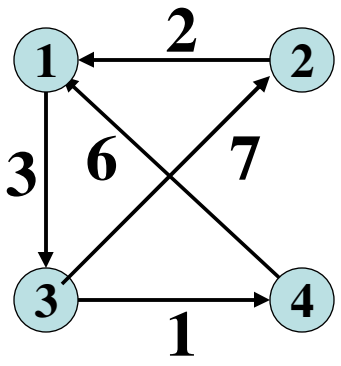
$D^{(2)}$

	v1	v2	v3	v4
v1	0	∞	3	∞
v2	2	0	5	∞
v3	9	7	0	1
v4	6	∞	9	0

$\Pi^{(2)}$

	v1	v2	v3	v4
v1	N/A	N/A	v1	N/A
v2	v2	N/A	v1	N/A
v3	v2	v3	N/A	v3
v4	v4	N/A	v1	N/A

Floyd's Algorithm (example)



$D^{(3)}$

	v1	v2	v3	v4
v1	0	10	3	4
v2	2	0	5	6
v3	9	7	0	1
v4	6	16	9	0

$\Pi^{(3)}$

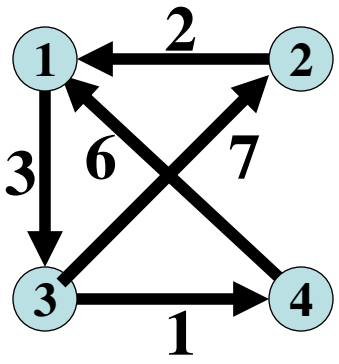
	v1	v2	v3	v4
v1	N/A	v3	v1	v3
v2	v2	N/A	v1	v3
v3	v2	v3	N/A	v3
v4	v4	v3	v1	N/A

$D^{(4)}$

	v1	v2	v3	v4
v1	0	10	3	4
v2	2	0	5	6
v3	7	7	0	1
v4	6	16	9	0

$\Pi^{(4)}$

	v1	v2	v3	v4
v1	N/A	v3	v1	v3
v2	v2	N/A	v1	v3
v3	v4	v3	N/A	v3
v4	v4	v3	v1	N/A



Deducing path for v2 to v4

$\pi(v2\text{-}v4) = \pi(v2\text{-}v3)$ --> v3 --> v4

$= \pi(v2\text{-}v1)$ --> v1 --> v3 --> v4

= v2 --> v1 --> v3 --> v4

Deducing path for v4 to v2

$\pi(v4\text{-}v2) = \pi(v4\text{-}v3)$ --> v3 --> v2

$= \pi(v4\text{-}v1)$ --> v1 --> v3 --> v2

= v4 --> v1 --> v3 --> v2

Floyd's Algorithm (pseudocode and analysis)

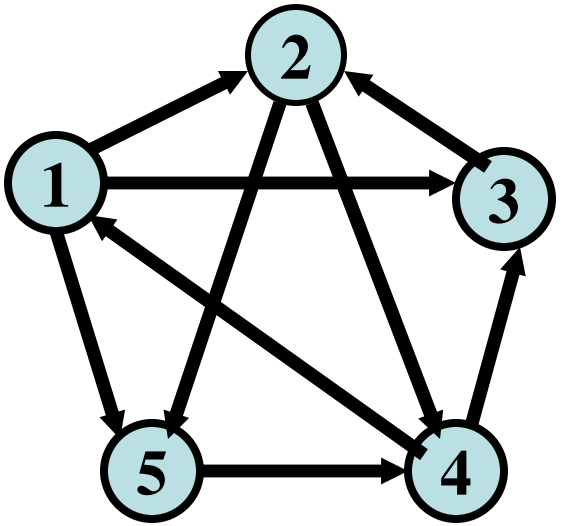
ALGORITHM *Floyd*$(W[1..n, 1..n])$

```
//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix W of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
D ← W //is not necessary if W can be overwritten
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
return D
```

Time efficiency: $\Theta(n^3)$

Space efficiency: $\Theta(n^2)$

Floyd's Algorithm (example)



	Weight Matrix				
	v1	v2	v3	v4	v5
v1	0	3	8	∞	-4
v2	∞	0	∞	1	7
v3	∞	4	0	∞	∞
v4	2	∞	-5	0	∞
v5	∞	∞	∞	6	0

$D^{(0)}$

	v1	v2	v3	v4	v5
v1	0	3	8	∞	-4
v2	∞	0	∞	1	7
v3	∞	4	0	∞	∞
v4	2	∞	-5	0	∞
v5	∞	∞	∞	6	0

$\Pi^{(0)}$

	v1	v2	v3	v4	v5
v1	N/A	v1	v1	N/A	v1
v2	N/A	N/A	N/A	v2	v2
v3	N/A	v3	N/A	N/A	N/A
v4	v4	∞	v4	N/A	N/A
v5	N/A	N/A	N/A	v5	N/A

$D^{(1)}$

	v1	v2	v3	v4	v5
v1	0	3	8	∞	-4
v2	∞	0	∞	1	7
v3	∞	4	0	∞	∞
v4	2	5	-5	0	-2
v5	∞	∞	∞	6	0

$\Pi^{(1)}$

	v1	v2	v3	v4	v5
v1	N/A	v1	v1	N/A	v1
v2	N/A	N/A	N/A	v2	v2
v3	N/A	v3	N/A	N/A	N/A
v4	v4	v1	v4	N/A	v1
v5	N/A	N/A	N/A	v5	N/A

$D^{(2)}$

	v1	v2	v3	v4	v5
v1	0	3	8	4	-4
v2	∞	0	∞	1	7
v3	∞	4	0	5	11
v4	2	5	-5	0	-2
v5	∞	∞	∞	6	0

$\Pi^{(2)}$

	v1	v2	v3	v4	v5
v1	N/A	v1	v1	v2	v1
v2	N/A	N/A	N/A	v2	v2
v3	N/A	v3	N/A	v2	v2
v4	v4	v1	v4	N/A	v1
v5	N/A	N/A	N/A	v5	N/A

Floyd's Algorithm (example)

$D^{(3)}$

	v1	v2	v3	v4	v5
v1	0	3	8	4	-4
v2	∞	0	∞	1	7
v3	∞	4	0	5	11
v4	2	-1	-5	0	-2
v5	∞	∞	∞	6	0

$\Pi^{(3)}$

	v1	v2	v3	v4	v5
v1	N/A	v1	v1	v2	v1
v2	N/A	N/A	N/A	v2	v2
v3	N/A	v3	N/A	v2	v2
v4	v4	v3	v4	N/A	v1
v5	N/A	N/A	N/A	v5	N/A

Deducing path from v3 to v1

π(v3-v1) = π(v3-v4) --> v4 --> v1

= π(v3-v2) --> v2 --> v4 --> v1

= v3 --> v2 --> v4 --> v1

$D^{(4)}$

	v1	v2	v3	v4	v5
v1	0	3	-1	4	-4
v2	3	0	-4	1	-1
v3	7	4	0	5	3
v4	2	-1	-5	0	-2
v5	8	5	1	6	0

$\Pi^{(4)}$

	v1	v2	v3	v4	v5
v1	N/A	v1	v4	v2	v1
v2	v4	N/A	v4	v2	v1
v3	v4	v3	N/A	v2	v1
v4	v4	v3	v4	N/A	v1
v5	v4	v3	v4	v5	N/A

Deducing path from v1 to v3

π(v1-v3) = π(v1-v4) --> v4 --> v3

π(v1-v5) --> v5 --> v4 --> v3

v1 --> v5 --> v4 --> v3

$D^{(5)}$

	v1	v2	v3	v4	v5
v1	0	1	-3	2	-4
v2	3	0	-4	1	-1
v3	7	4	0	5	3
v4	2	-1	-5	0	-2
v5	8	5	1	6	0

$\Pi^{(5)}$

	v1	v2	v3	v4	v5
v1	N/A	v3	v4	v5	v1
v2	v4	N/A	v4	v2	v1
v3	v4	v3	N/A	v2	v1
v4	v4	v3	v4	N/A	v1
v5	v4	v3	v4	v5	N/A