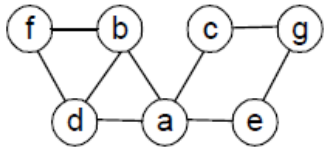
CSC 323 Algorithm Design and Analysis
Module 5: Graph Algorithms
5.1: DFS and BFS Traversal Algorithms

Instructor: Dr. Natarajan Meghanathan
Sample Questions and Solutions

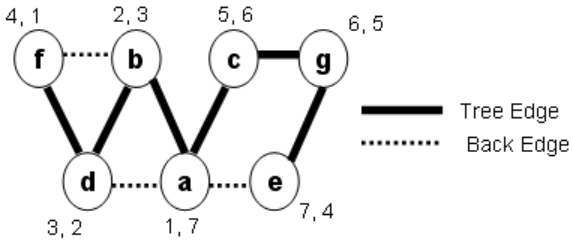
1) Consider the following graph:



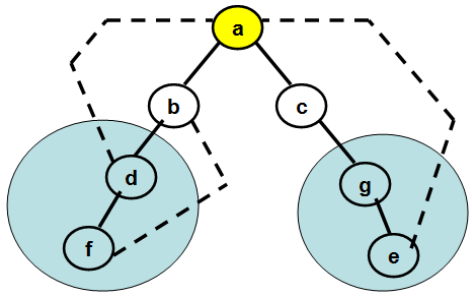
a) Compute the DFS tree and draw the tree edges and back edges
b) Write the order in which the vertices were reached for the first (i.e. pushed into the stack)
c) Write the order in which the vertices became dead ends (i.e. popped from the stack)
d) Determine the articulation points of the graph

Solution:

a)



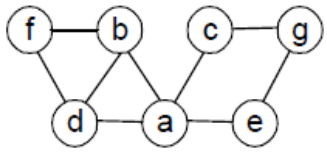
b) Order being pushed into the stack: a, b, d, f, c, g, e
c) Order being popped off the stack: f, d, b, e, g, c, a
d)

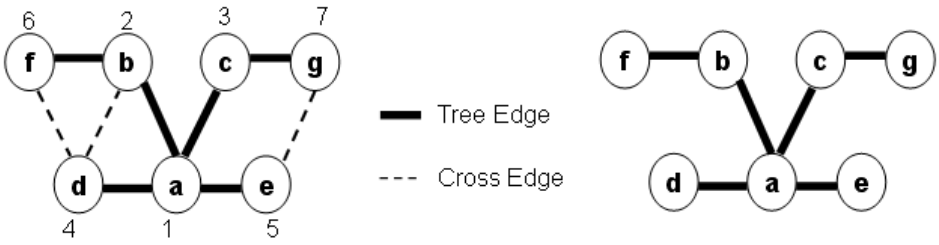


- In the above graph, vertex 'a' is the only articulation point.
- Vertices 'e' and 'f' are leaf nodes.
- Vertices 'b' and 'c' are candidates for articulation points. But, they cannot become articulation point, because there is a back edge from the only sub tree rooted at their child nodes ('d' and 'g' 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.

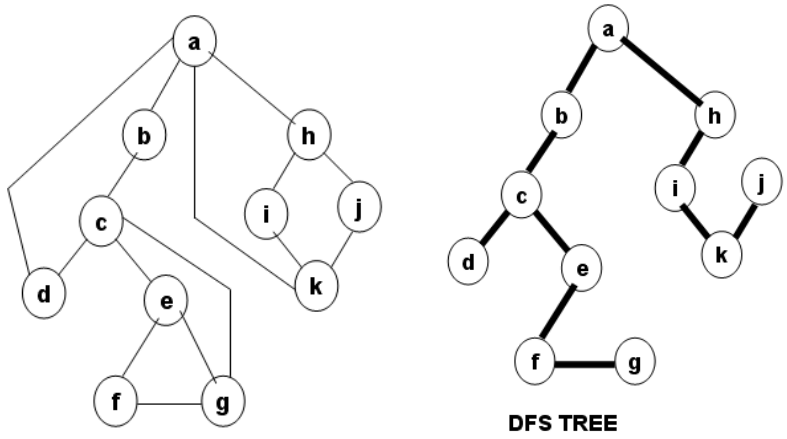
2) How will you check for a graph's acyclicity with DFS and BFS?
With DFS, if we encounter a back edge, the graph has cycles; With BFS, if we encounter a cross edge, the graph has cycles.

3) Find the minimum edge paths from vertex 'a' to every other vertex in the graph below:



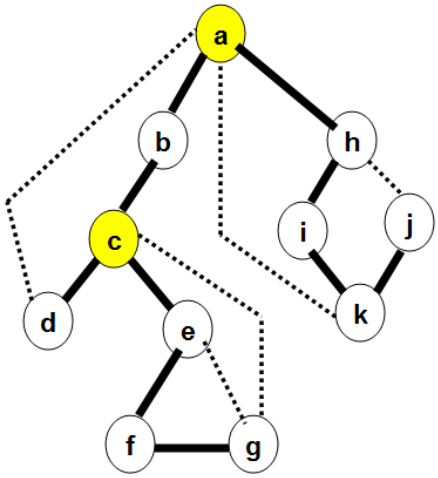


4) Find the articulation points of the following graph and justify your answer.



Solution

DFS Tree



Articulation Points

- Vertex **a** is an articulation point as it has more than one child node (nodes b and h) connected with a tree edge
- Vertex **c** is an articulation point because the only child node that has a rooted sub tree is e and there is no back edge in this sub tree that goes past or above c in the DFS tree.

Not Articulation points

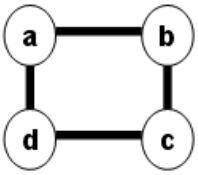
- Vertices d, g and j are leaf nodes
- Vertices f and k are not articulation points as they have only one child node, each, and the sub tree rooted at these child nodes are connected to higher-level vertices.
- Vertices b, e, h and i are not articulation points as the sub tree rooted at their respective only child nodes c, f, i and k have back edges to vertices that are higher above these vertices.

5) Prove that BFS yields the minimum edge paths from the source vertex to every other vertex in a connected graph.

Note that with BFS, we do not explore the neighbors of a vertex that is at a longer distance from the source vertex, before exploring the neighbors of a vertex that is at a shorter distance from the source vertex. Hence, in BFS starting with a source vertex *s*, when a vertex *v* is visited for the first time, through a tree edge *u* – *v*, the distance (# edges) from *s* to *v* is one edge more than the distance from *s* to *u*. If there was a shorter path from *s* to *v*, then *v* should have been visited before *u* was visited (i.e. *v* is closer to *s* than *u*) or at the same time when *u* was visited (i.e. *v* is at the same distance to *s* as *u*). However, since *v* is

visited (for the first time) from *u*, the number of edges of *s* to *v* has to be one more than the number of edges from *s* to *u*.

6) Prove (through an example) that DFS is not always guaranteed to find the shortest path (minimum edge tree).



A DFS on the graph would yield a tree (that is basically a chain): a – b – c – d, whereas by running BFS from vertex **a**, we could find the shortest path tree with edges a – b, b – c, and a – d.

11) How would you find whether a graph is 2-colorable (also called bi-partite) or not.

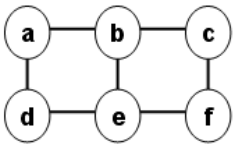
Note: A graph is said to be bi-partite or 2-colorable if the vertices of the graph can be colored in two colors such that every edge has its vertices in different colors. In other words, we can partition the set of vertices of a graph into two disjoint sets such that there is no edge from a vertex in one set to a vertex in the other set.

We can check for the 2-colorable property of a graph by running a DFS or BFS

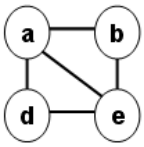
- With BFS, if there are no cross-edges between vertices at the same level, then the graph is 2-colorable.
- With DFS, if there are no back edges between vertices that are both at odd levels or both at even levels, then the graph is 2-colorable.

7) Find whether the following graphs are 2-colorable or not.

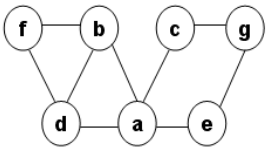
a)



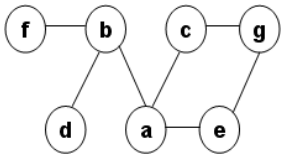
b)



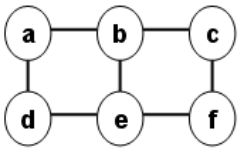
c)

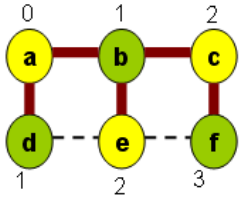


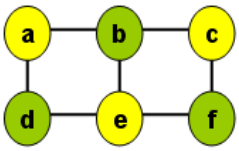
d)



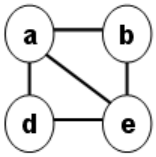
a) Yes, the graph is 2-colorable

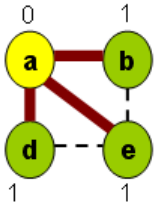






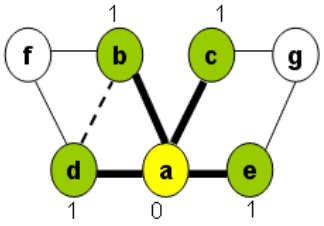
b) No, the graph is not 2-colorable





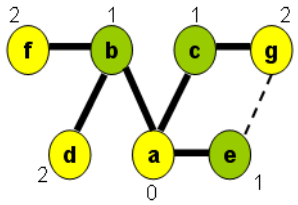
We encounter cross edges between vertices b and e; d and e – all the three vertices are in the same level.

c) No, the graph is not 2-colorable



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

d) Yes, the graph is 2-colorable



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

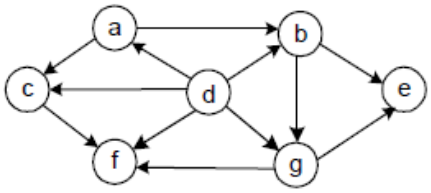
8) When do you say a graph is bi-partite? How would you determine it?

A graph is said to be bi-partite, if we can partition its vertex set into two disjoint sets such that there is no edge between vertices in the same set; all the edges in the graph are across the two sets. A graph is said to be bi-partite, if it is 2-colorable. We test for the 2-colorability of a graph by running Breadth First Search (BFS): We should not encounter a cross edge between vertices at the same level (distance from the root).

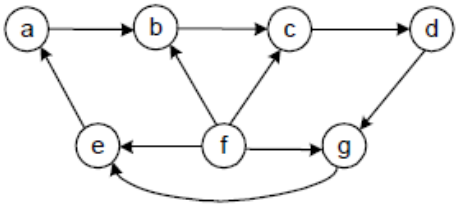
CSC 323 Algorithm Design and Analysis
Module 5: Graph Algorithms
5.2: Directed Acyclic Graphs and Topological Sorting

Instructor: Dr. Natarajan Meghanathan
Sample Questions and Solutions

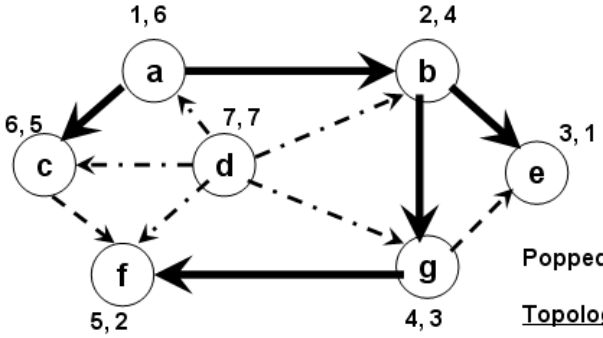
1) Apply the DFS-algorithm to solve the topological sorting problem for the following directed graphs:



(a)



(b)

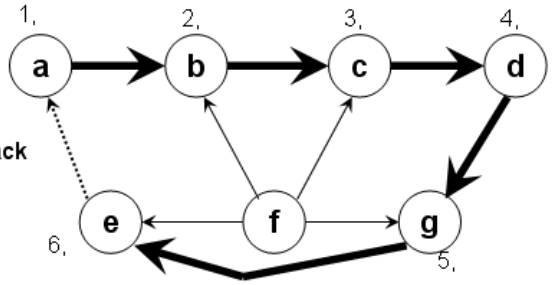


Forward edge

Cross edge

Popped order: e, f, g, b, c, a, d

Topological Sort: d, a, c, b, g, f, e



Back edge

Encountered a back edge. So, no Topological sort is possible

2) Prove that being a DAG is the necessary and sufficient condition to be able to do a topological sorting of 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.

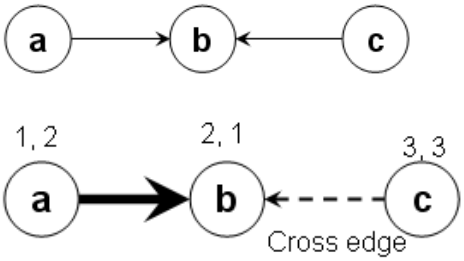
- To test whether a directed graph is a DAG, run DFS on the directed graph, if we encounter a back edge, then the digraph is not a DAG; otherwise, the digraph is a DAG.

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).
  - We may have many other topologically sorted ordering of the vertices in the digraph (DAG); but, no such ordering could list v ahead of u. 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.
  - So, if a DAG has an edge u → v, a DFS traversal of the DAG and a listing of the vertices in the reverse order according to which they are popped out from the stack guarantees that u is listed ahead of v in the list. This observation holds good for every edge in the DAG.

3) Can we use the order in which the vertices are pushed onto the DFS stack (instead of the order they are popped off it) to solve the topological sorting problem?

The answer is no. We show this with a counterexample.

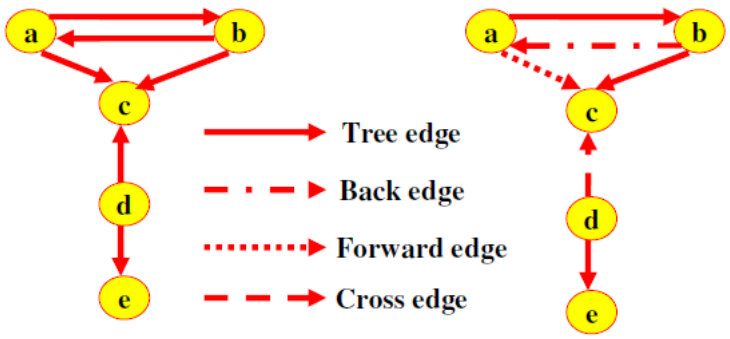


Pushed order: a, b, c. This is not a topological sort as there is an edge from c → b; so vertex c should appear before vertex b in the topological sort.

Reverse of Popped off order (Topological sort order): c, a, b

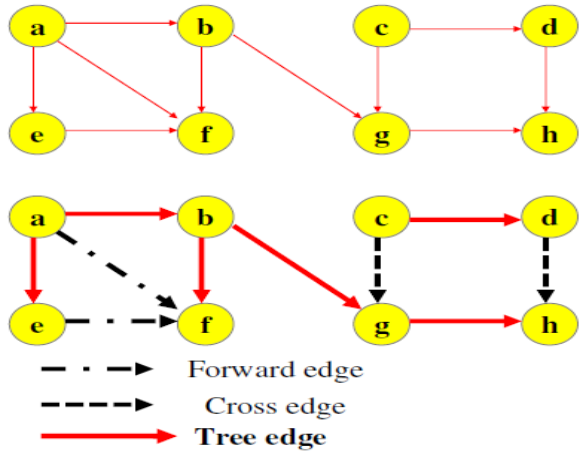
4) Run the DFS algorithm on the following directed graphs and identify the tree edges, back edges, forward edges and cross edges. State whether the directed graph is a DAG or not. Justify.

(a)



The directed graph is not a DAG since it has a back edge (from b to a).

(b)



The above directed graph is a DAG since there is no back edge in its DFS traversal.

CSC 323 Algorithm Design and Analysis
Module 5: Graph Algorithms
5.3: Single-Source Shortest Path Dijkstra Algorithm
Instructor: Dr. Natarajan Meghanathan
Sample Questions and Solutions

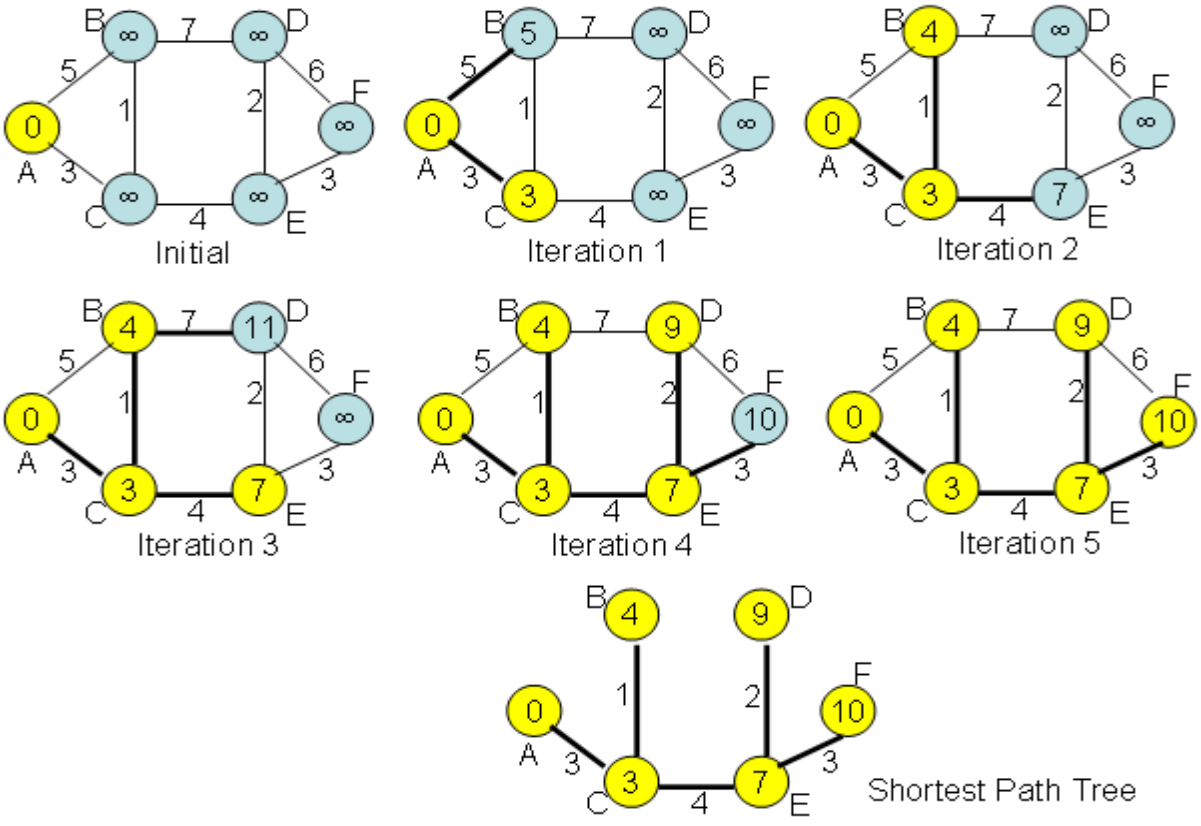
1) Prove that the sub-path of a shortest path is also a shortest path.

- **Proof:** Let us 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.

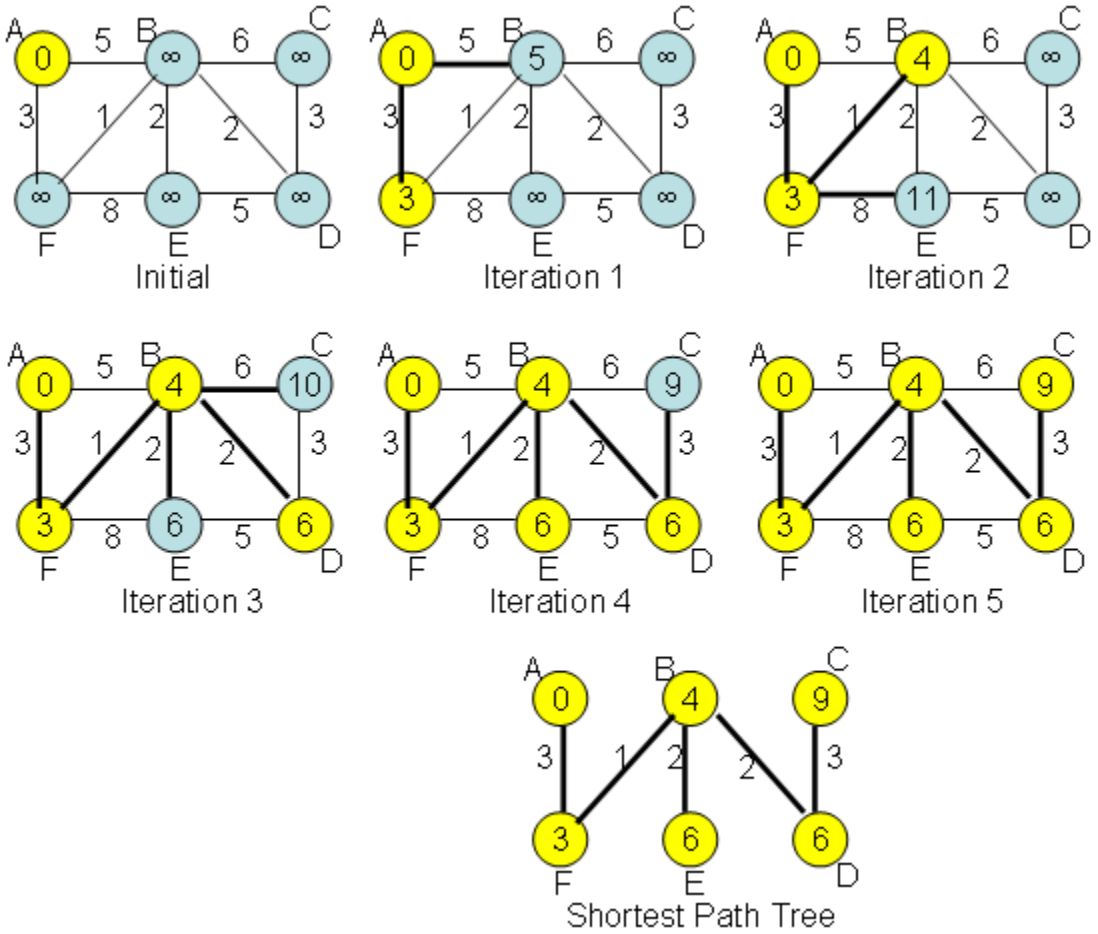
2) Prove the following statement: When a vertex *v* is picked for relaxation/optimization, every intermediate vertex on the *s*…*v* shortest path is already optimized.

- **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.

3) Determine the shortest paths (minimum weight paths) from the source vertex A to every other vertex on the following graphs:
(a)



(b)



4) **Prove the correctness of the Dijkstra's algorithm.**

- Let P be the so-called shortest path from *s* to *v* that the Dijkstra algorithm finds. We need to prove that P is indeed the shortest *s*…*v* path. Assume the contradiction that there exists a path P' from *s* to *v* such that the weight(P') < weight(P).
- With regards to the hypothetical path P' (such that P ≠ P' and weight(P') < weight(P)), there are two scenarios:
- Scenario 1: If all vertices in P' are already optimized, the weight of the shortest path to these vertices should be even less than that of weight (P') < weight (P). Dijkstra algorithm would have then used the relaxation steps for optimizing these vertices to also optimize the weight of the shortest path to vertex *v*, and weight (P) = weight (P').
- Scenario 2: There has to be an intermediate vertex *y* on the path P' that has not yet been relaxed by Dijkstra's algorithm. However, the fact that Dijkstra algorithm picked vertex *v* ahead of vertex *y* implies that the *s*...*y* path is of weight larger than or equal to the *s*...*v* path. Hence, if *y* had to be an intermediate vertex on the path P' from *s* to *v*, the weight(P') = weight(*s*...*y* path) + weight (*y*...*v* path). Since the edges on the *y*...*v* path are required to have weight greater than 0, weight(P') > weight(*s*...*y* path) ≥ weight(P). This is a contradiction to our assumption that weight(P') < weight(P).

5) Given the pseudo code of the Dijkstra algorithm (below), analyze the run-time complexity of the Dijkstra's shortest path algorithm.

```
Begin Algorithm Dijkstra (G, s)
1   For each vertex v ∈ V                                  } O(V) time
2       d [v] ← ∞ // an estimate of the min-weight path from s to v
3   End For                                                } O(V) time to Construct a Min-heap
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  ←—— done |V| times = O(V) time
8       u ← EXTRACT-MIN(Q)  ←—— Each extraction takes O(logV) time
9       S ← S U {u}
10      For each vertex v such that (u, v) ∈ E }  done O(E) times totally
11          If v ∈ Q and d [v] > d [u] + w (u, v) then   } It takes O(logV) time when done once
12              d [v] ← d [u] + w (u, v)
13              Predecessor (v) = u
13          End If
14      End For
15  End While
16  End Dijkstra
```

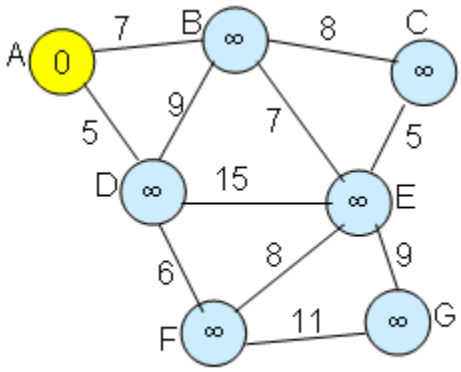
Overall Complexity: O(V) + O(V) + O(VlogV) + O(ElogV)
Since |V| = Ω(|E|), the VlogV term is dominated by the ElogV term. Hence, overall complexity = O(|E|*log|V|)

- The run-time complexity of Dijkstra algorithm is O(|*E*|**log*|*V*|), because, we visit all the |E| edges of the graph and see if the weight of a path to a vertex (from the source) can be further optimized as part of the relaxation step. Each time, we do a relaxation, the Priority-Queue of the vertices needs to be rearranged to sort the vertices in the increasing order of weights for the path from the source. If the Priority-Queue of the |V| vertices is maintained as a heap, we would spend log|V| time to rearrange the Priority-Queue for each relaxation. There can be at most |E| such relaxation across all the vertices. Hence, the run-time complexity of the Dijkstra algorithm is O(|*E*|**log*|*V*|).

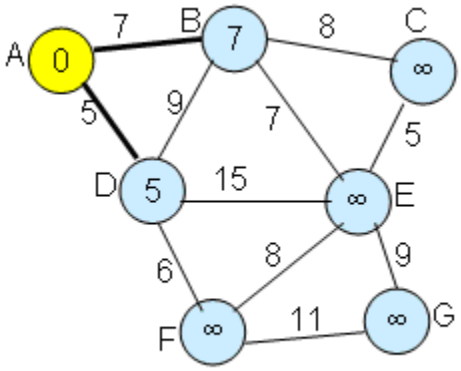
CSC 323 Algorithm Design and Analysis
Module 5: Graph Algorithms
5.5: Minimum Spanning Tree Algorithms
Instructor: Dr. Natarajan Meghanathan
Sample Questions and Solutions

1) **Determine a minimum spanning tree of the following graph using Prim's algorithm.**

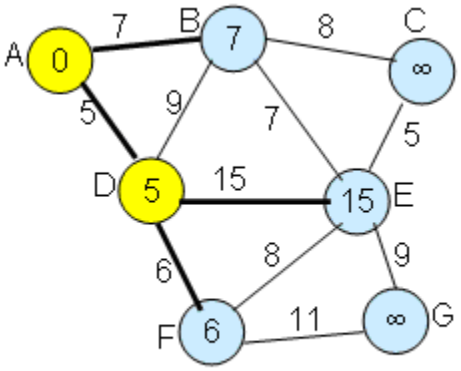
(a)



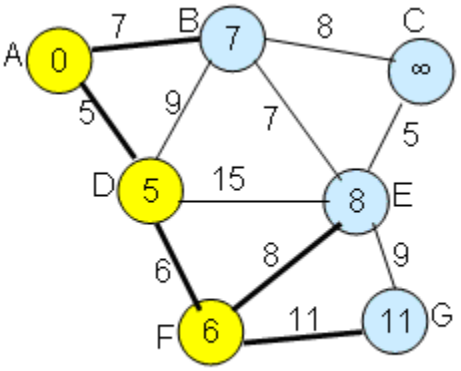
Initial



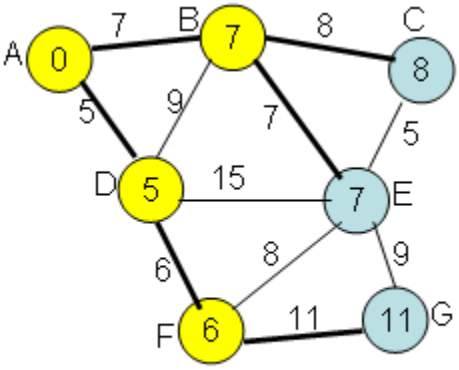
Iteration 1



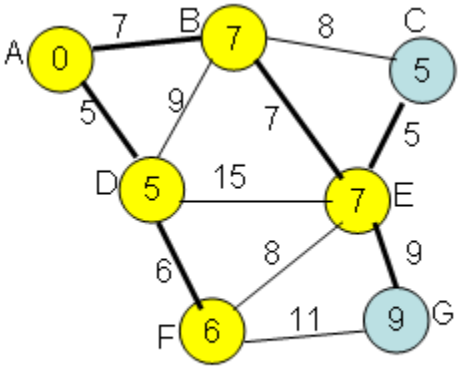
Iteration 2



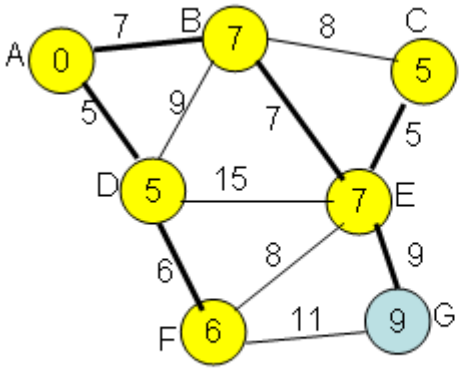
Iteration 3



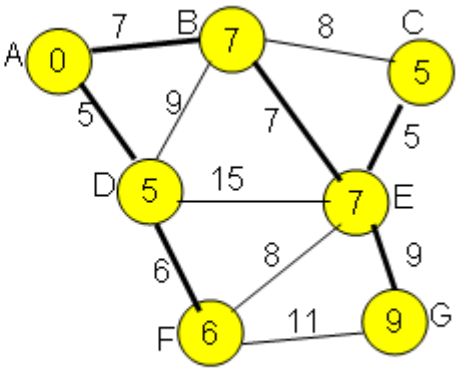
Iteration 4



Iteration 5

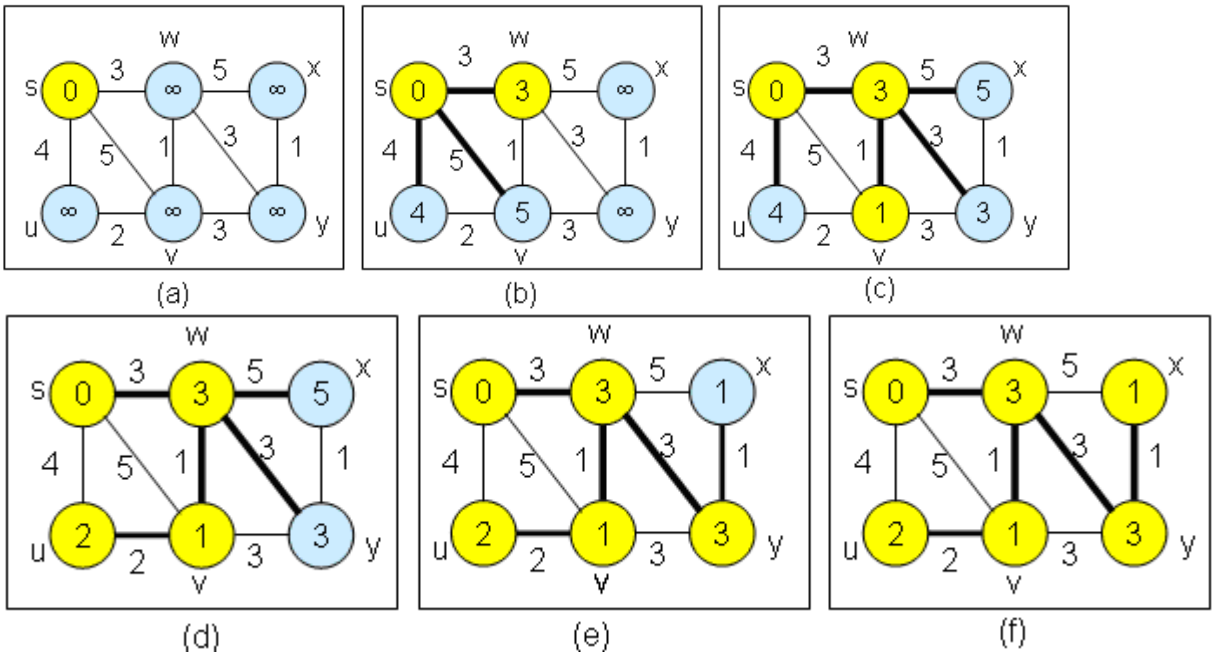


Iteration 6



Iteration 7

(b)

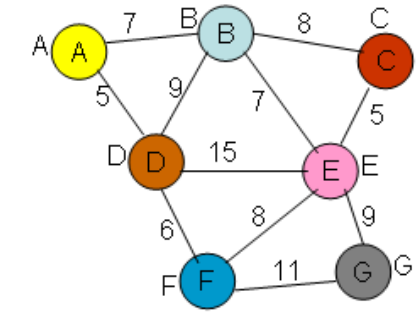


2) Provide a proof of correctness for the Prim's algorithm.

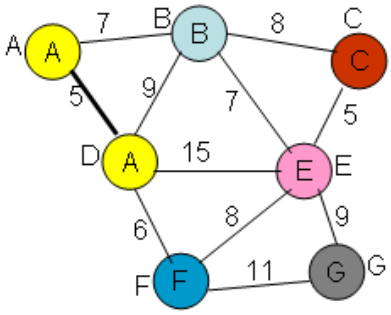
- If we can prove that the spanning tree of the vertices in the Optimal Set at the end of each iteration is a minimum spanning tree, then the spanning tree of all the vertices in the Optimal Set at the end of the final iteration is a minimum spanning tree of the entire graph.
- We will prove using induction.
- To start with, at the end of iteration 1, we have an one-vertex minimum spanning tree comprising of the starting vertex with no edges.
- At the end of iteration i-1, assume we have a minimum spanning tree of the vertices in the Optimal Set.
- We will have to prove that at the end of iteration i, the spanning tree Ti that we obtain for the vertices in the Optimal Set, expanded by one vertex and an edge (v, u) that was the minimum weight edge among the edges that connected the vertices *v* in the Optimal Set and the vertices *u* in the Fringe Set, is a minimum spanning tree.

- Assume that there exists another (a hypothetical one) spanning tree Ti' of the vertices in the Optimal Set that is of weight lower than the weight of the spanning tree Ti formed by Prim's algorithm; i.e., Wt(Ti') < Wt(Ti).
- Note that Ti = Ti-1 U {u-v} according to Prim's algorithm and for our hypothetical MST Ti' = Ti-1 U {u'-v'}. That is, Wt(Ti) = Wt(Ti-1) + Wt(u-v). Similarly, Wt(Ti') = Wt(Ti-1) + Wt(u'-v').
- For edge {u-v} selected by Prim's algorithm, the end vertex u should be in the Optimal Set and the end vertex v should be in the Fringe Set (i.e., both the end vertices cannot be in the Optimal Set or in the Fringe Set). In other words, for {u-v} to be included to Ti-1 to expand the MST by one more edge leading to Ti, the edge {u-v} should cross the Optimal Set and Fringe Set.
- Similarly, for edge {u'-v'} to be part of Ti', its end vertex u' should have been already part of the MST Ti-1 and its other end vertex v' should not be part of Ti-1 (i.e., v' should be among the list of vertices that are yet to be part of the MST). This implies that edge {u'-v'} should also be an edge crossing from the Optimal Set to the Fringe Set in the context of Prim's algorithm.
- The way Prim's algorithm operates, the weight of an edge selected during an iteration is the minimum among all edges crossing the Optimal Set and Fringe Set that exists during that iteration. Hence, for iteration i, the weight of edge {u, v} is the minimal among edges crossing the Optimal Set to Fringe Set. So, wt{u-v} ≤ wt{u'-v'}. Hence, Wt(Ti) has to be only ≤ Wt(Ti') (i.e., Wt(Ti') ≥ Wt(Ti)) which contradicts our assumption that the weight of the hypothetical MST Ti', Wt(Ti') < Wt(Ti).

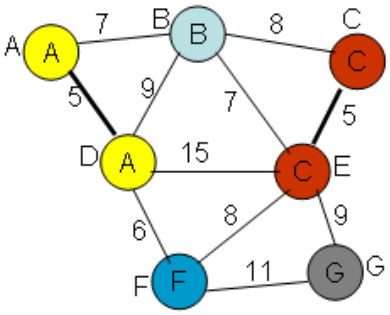
3) Determine a minimum spanning tree of the following graph using Kruskal's algorithm.



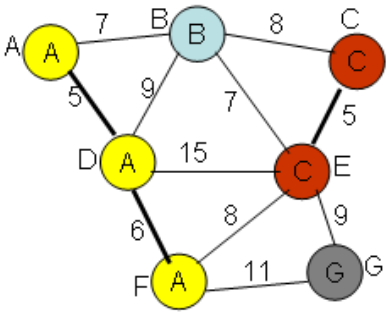
Initial



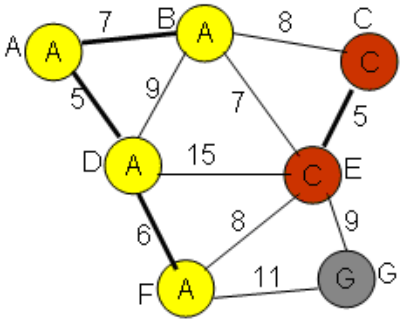
Iteration 1



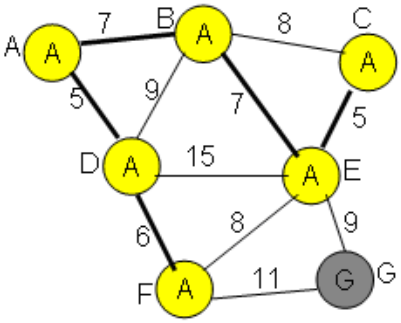
Iteration 2



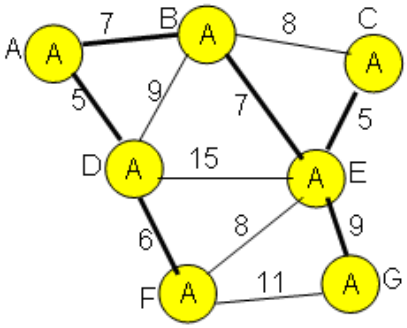
Iteration 3



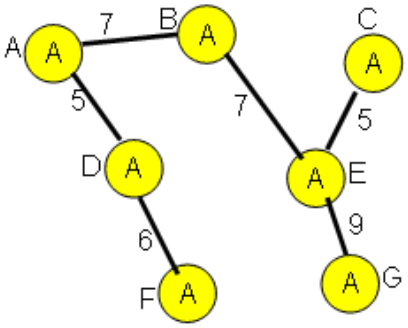
Iteration 4



Iteration 5



Iteration 6



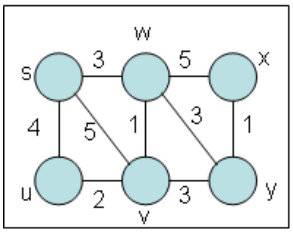
Minimum Spanning Tree

4) Provide a proof of correctness of the Kruskal's algorithm.

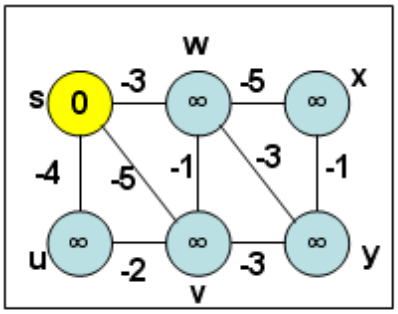
- Let *T* be the spanning tree generated by Kruskal's algorithm for a graph *G*. Let *T'* be a minimum spanning tree for *G*. We need to show that both *T* and *T'* have the same weight.
- Assume that wt(*T'*) < wt(*T*).
- Hence, there should be an edge *e* in *T* that is not in *T'*. Because, if every edge of *T* is in *T'*, then *T* = *T'* and wt(*T*) = wt(*T'*).
- Pick up the edge *e* ε *T* and *e* ε *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'*) < 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.

5) A maximum spanning tree is the spanning tree with the largest sum of the edge weights. Use one of the two minimum spanning tree algorithms that we saw in this course to compute a maximum spanning tree of the graph given below.
Explain the necessary transformations involved.
Is there any difference in the time complexity of the algorithms to compute the maximum and the minimum spanning trees? Justify.

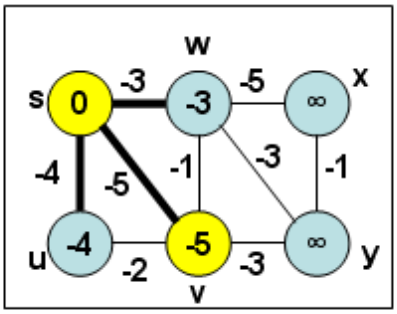
The initial transformation is that we change the sign of all the edge weights (i.e., +ve to –ve and vice-versa). This can be done in O(|E|) time. We run a minimum spanning tree algorithm on the modified graph. We revert the signs of all the edges in the minimum spanning tree to obtain the maximum spanning tree – the sum of the edge weights is the maximum. The reversion of the signs can be done in O(|V|) time. The overall complexity is O(|E|) + O(|V|) + O(|E|*log|V|), if we use the Prim's algorithm. Since |E| = Ω(|E|*log|V|) and |V| = Ω(|E|*log|V|), we can say that the overall-time complexity of determining the maximum spanning tree using the Prim's algorithm is still O(|E|*log|V|).



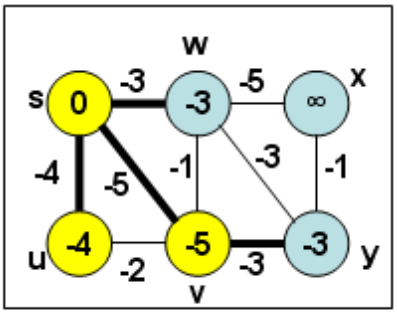
Given Graph



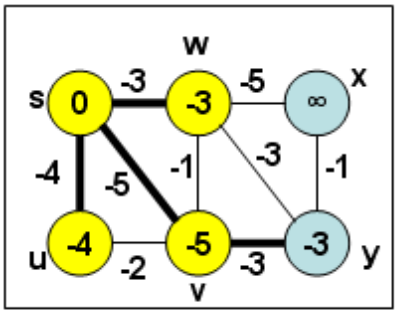
(a)



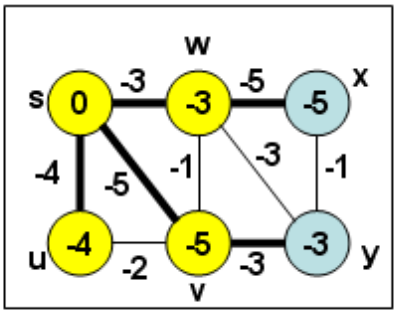
(b)



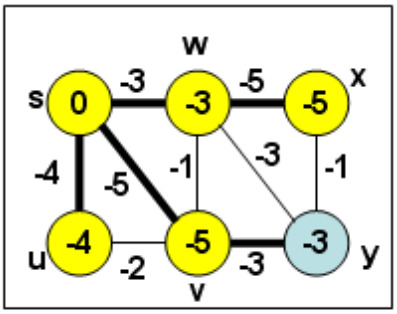
(c)



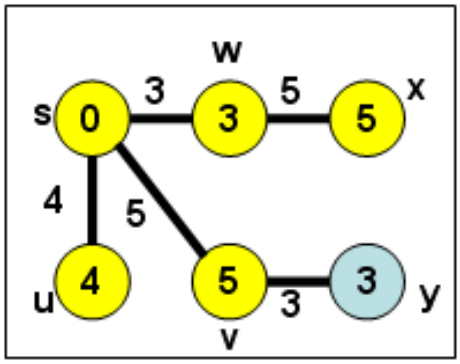
(d)



(e)



(f)



Max. Weight = 20

6) Prove that if a weighted graph with unique edge weights has a cycle, then the heaviest weight edge (edge of maximum weight) in the cycle will not be part of a minimum spanning tree of the graph.

Let there be a weighted graph that has a cycle v_i ... v_x - v_y .... v_j - v_k ... v_i such that the edge v_j - v_k is the heaviest weight edge.

Assume v_j - v_k is in the minimum spanning tree (MST) T. Now remove the edge v_j - v_k from T to split the vertices of T to two disjoint subsets J and K (note the union of the two disjoint subsets of these vertices will be the set of all vertices in the graph) such that v_j is in J and v_k is in K. Since v_j - v_k is part of a cycle in the original graph, there should be an alternate path from v_j to v_k and (since all edges in the graph have unique edge weights) the weight of any edge in this alternate path should be less than the weight of the edge v_j - v_k. Hence, there should be an edge v_x - v_y that crosses the two disjoint subsets of vertices J and K whose weight is less than the weight of the edge v_j - v_k. We could use the edge v_x - v_y to connect the disjoint subsets J and K and obtain a spanning tree T ' that spans all the vertices of the graph. In other words T ' = T - {v_j - v_k} U {v_x - v_y}. Since weight(v_x - v_y) < weight(v_j - v_k), the weight(T ') < weight(T) which contradicts the assumption that T is a MST.

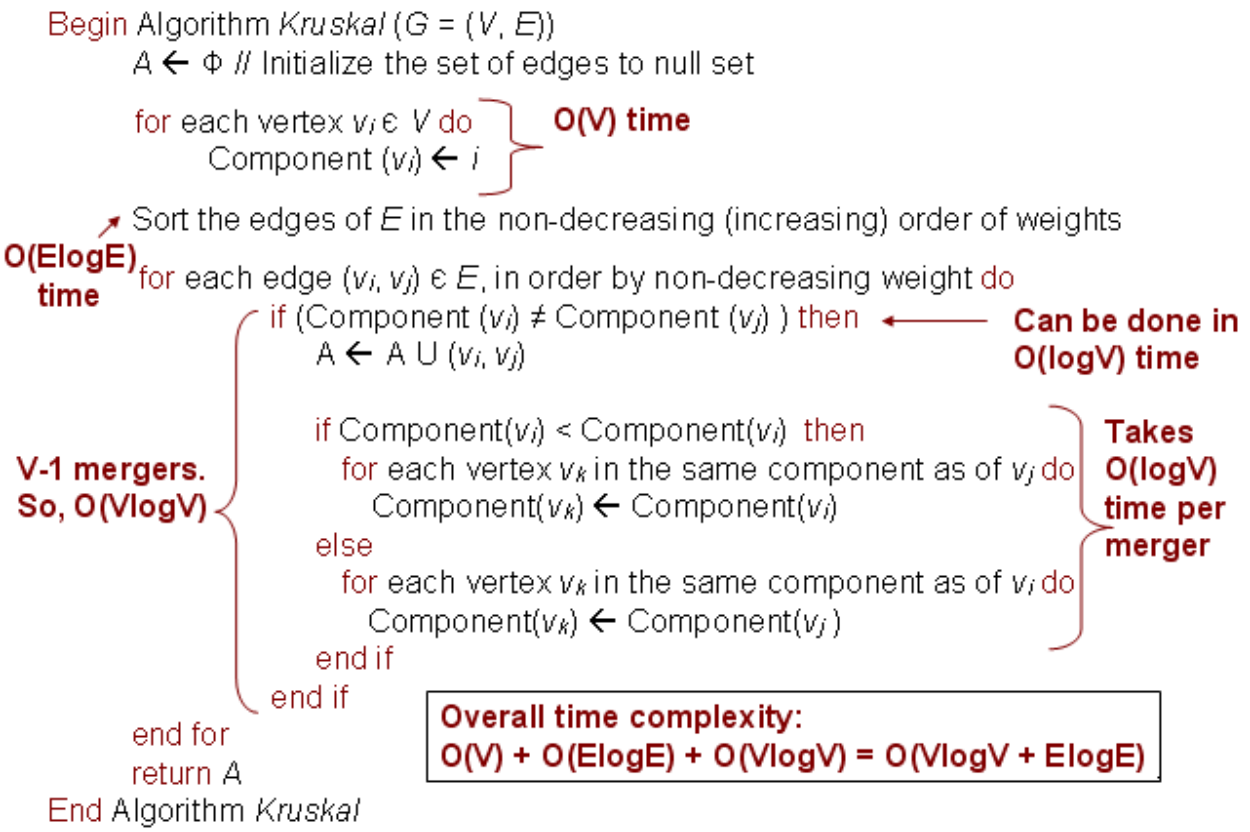
Hence, if all edges in a weighted graph have unique weights, the heaviest weight edge in a cycle of the graph is guaranteed not to be in the Minimum Spanning Tree of the graph.

7) Show that if the edges in a graph are of unique (distinct) weight (i.e., no two edges in the graph have different weights), then there is only one minimum spanning tree of the graph.

We will prove this by contradiction.

- 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.
- 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) $\leq$ 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.

8) Given the pseudo code of the Kruskal's algorithm below, analyze its run-time complexity:



9) Prove the following theorem for minimum spanning trees:
Let us define a IJ-cut of the set of vertices V of a weighted graph to be two disjoint sets I and J such that I ∪ J = V and I ∩ J = ϕ. Let IJ-cut-set be the set of edges that connect the vertices in the two disjoint sets I and J in the weighted graph. If an edge (i, j) is a minimal weight edge in an IJ-cut-set, then the edge (i, j) has to be part of a minimum spanning tree of the graph.

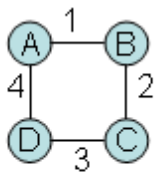
Proof: We will prove this by contradiction. Let (i, j) be the minimum weight edge in an IJ-cut-set and it cannot be part of a minimum spanning tree. Then, some other edge (i', j') in the IJ-cut-set should be part of the minimum spanning tree; otherwise, if no edge in the IJ-cut-set is part of a minimum spanning tree, then there exists no spanning tree of the entire graph. However, the weight(i', j') ≥ weight(i, j) since (i, j) is the edge of minimum weight in the IJ-cut-set. If this is the case, then we can remove (i', j') from the minimum spanning tree and connect the IJ-cut of vertices using (i, j) to restore the connectivity of the spanning tree, and the weight of this new spanning tree involving edge (i, j) can be only less than the weight of the original spanning tree involving (i', j') since weight(i, j) ≤ weight(i', j'). This contradicts the assumption that (i, j) cannot be part of a minimum spanning tree.

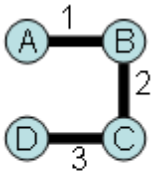
10) Prove that if an edge (i, j) is part of a minimum spanning tree T of a weighted graph of V vertices, its two end vertices are part of an IJ-cut and (i, j) is the minimum weight edge in the IJ-cut-set.

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

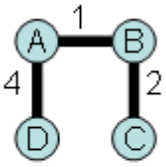
and restore its connectivity and spanning nature by adding (i', j') instead. By doing this, we will only lower the weight of T contradicting the assumption that T is a minimum spanning tree to start with. Hence, every edge (i, j) of a minimum spanning tree has to be the minimum weight edge in an IJ-cut such that $i \in I$ and $j \in J$, and $I \cup J = V$ and $I \cap J = \phi$.

11) Show (using a simple graph example) that the shortest path tree and the minimum spanning tree obtained for a graph need not be the same. Discuss any potential tradeoffs you would typically observe between the two types of trees.





Minimum Spanning Tree
obtained with Prim's or Kruskal's

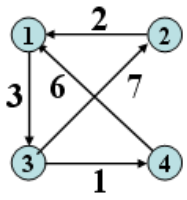


Shortest Path Tree (rooted at A)
obtained with Dijkstra's

As noted in the above example, the weight of a path from one vertex to another vertex on a minimum spanning tree need not be always the minimum. In the figure above, the weight of the path A – B – C – D on the minimum spanning tree is 7 whereas, we notice from the shortest path tree rooted at A, that there is a path (A – D) of weight 4 from A to D. On the other hand, the shortest path tree rooted at any vertex need not be the minimum spanning tree for the entire graph and the weight of the shortest path tree may typically exceed that of the minimum spanning tree, as observed in the above example. The shortest path tree rooted at A weights 7 whereas, the minimum spanning tree weighs 6.

CSC 323 Algorithm Design and Analysis
Module 5: Graph Algorithms
5.5: All Pairs Shortest Paths Algorithm
Instructor: Dr. Natarajan Meghanathan
Sample Questions and Solutions

1) Run the Floyd's algorithm on the following digraph and deduce the paths from v2 to v4 and vice-versa.



$D^{(0)}$

	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

$D^{(3)}$

	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

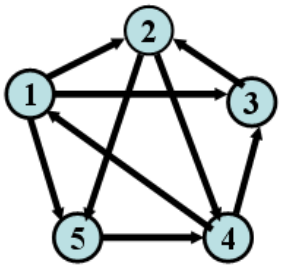
$\Pi^{(4)}$

	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
π(v2-v4) = π(v2-v3) --> v3 --> v4
= π(v2-v1) --> v1 --> v3 --> v4
= v2 --> v1 --> v3 --> v4

Deducing path for v4 to v2
π(v4-v2) = π(v4-v3) --> v3 --> v2
= π(v4-v1) --> v1 --> v3 --> v2
= v4 --> v1 --> v3 --> v2

2) Run the Floyd's algorithm on the following digraph (given its adjacency matrix) and deduce the paths from v1 to v3 and vice-versa.



	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

$D^{(3)}$

	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

$D^{(4)}$

	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

$D^{(5)}$

	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

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

$\pi(v1\text{-}v3) = \pi(v1\text{-}v4) \dashrightarrow v4 \dashrightarrow v3$
$\pi(v1\text{-}v5) \dashrightarrow v5 \dashrightarrow v4 \dashrightarrow v3$
$v1 \dashrightarrow v5 \dashrightarrow v4 \dashrightarrow v3$

3) Compare the time complexities of the Dijkstra algorithm and the Floyd's algorithm to determine the shortest paths (minimum weight paths) between all pairs of vertices for sparse graphs and dense graphs, and justify which algorithm you would use for each of these two types of graphs.

The Floyd's algorithm (of time complexity $\Theta(V^3)$ on a V-vertex graph) is designed to determine shortest paths between all pairs of vertices in a connected graph. Hence, when this algorithm is run once on a connected graph of V-vertices and E-edges, it can determine the shortest paths between all pairs of vertices, at a run-time complexity of $\Theta(V^3)$.

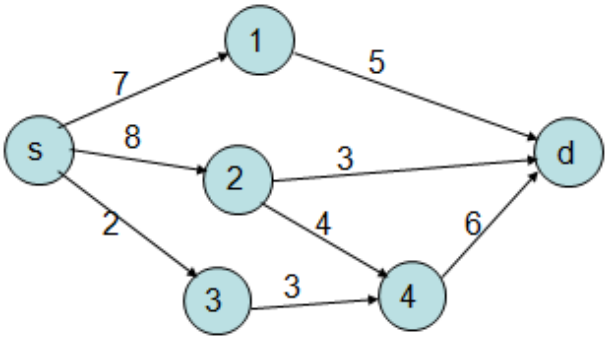
The Dijkstra's algorithm (of time complexity $\Theta(E*\log V)$ on a V-vertex and E-edge graph) is designed to determine the shortest path from one vertex (the source or the starting vertex) to all the other vertices in a connected graph. Hence, when this algorithm is to be used to determine the shortest paths between all pairs of vertices, the algorithm has to be run V-times, each time on a particular vertex. Hence, the overall time complexity of using the Dijkstra's algorithm for all-pairs-shortest-paths is $\Theta(V*E*\log V)$.

For sparse connected graphs, the minimum number of edges is $|E| = |V| - 1$. Hence, $E = \Theta(V)$. For such graphs, $O(V*E*\log V) = \Theta(V^2*\log V)$. Since, $\log V < V$, as $V \rightarrow \infty$, $V^2*\log V < V^3$. Hence, it would be prudent to use the Dijkstra's algorithm with a resulting time complexity of $\Theta(V^2*\log V)$ for sparse graphs.

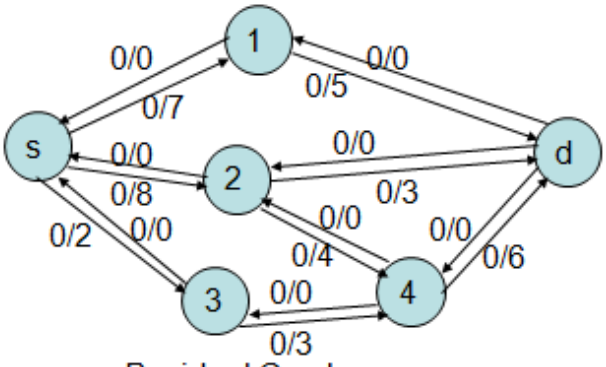
For dense connected graphs, the maximum number of edges is $|E| = |V|*(|V|-1)/2$. Hence, $E = \Theta(V^2)$. For such graphs, $\Theta(V*E*\log V) = \Theta(V^3*\log V) > \Theta(V^3)$. Hence, it would be prudent to use the Floyd's algorithm with a resulting time complexity of $\Theta(V^3)$ for dense graphs, especially for meshes – completely connected graphs.

CSC 323 Algorithm Design and Analysis
Instructor: Dr. Natarajan Meghanathan
Module 5.6 – Iterative Improvement

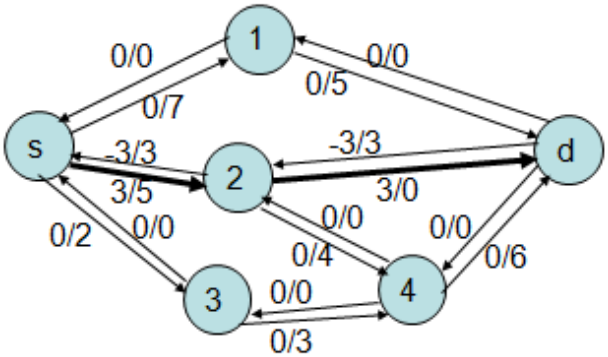
2) Find the maximum flow from the source *s* to the destination *d* in the following graph where the weights of the directed edges indicate the capacity of the edges. In addition, find the minimum set of edges (minimum cut) that when removed from this graph will disconnect *s* and *d*. Show all work.



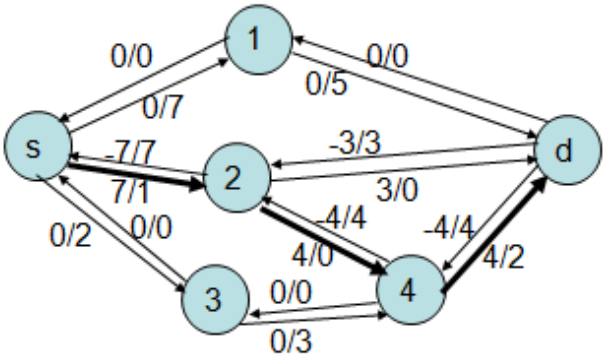
Initial Condition



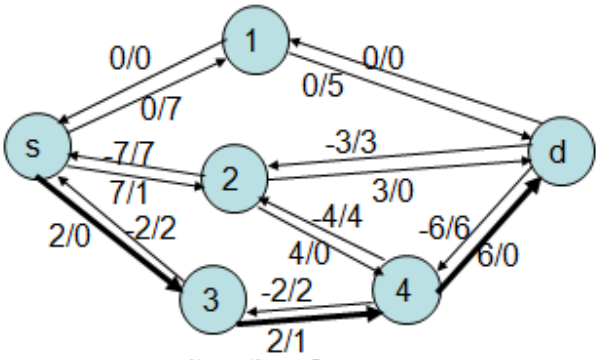
Residual Graph



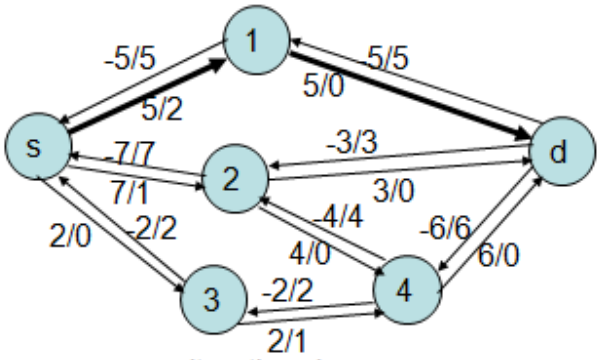
Iteration 1



Iteration 2



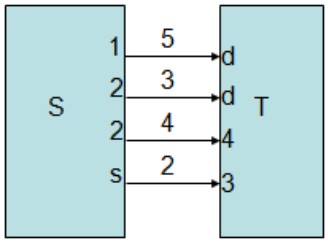
Iteration 3



Iteration 4

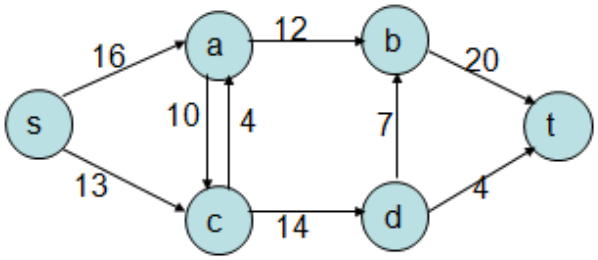
Maximum flow = 3 + 4 + 2 + 5 =14

In the final Residual graph, we see that Source *s* is reachable to nodes 1 and 2. Nodes 3, 4 and destination *d* are not reachable from *s*. S = {*s*, 1, 2} and T = {3, 4, *d*}

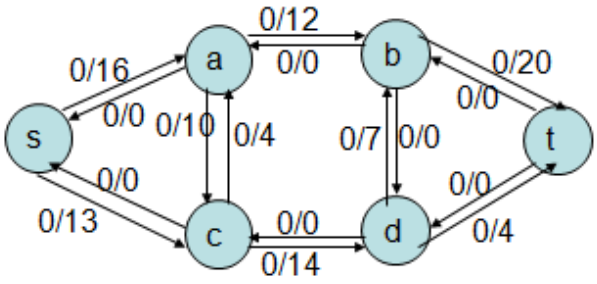


Find edges between S and T – These are the edges that form the bottleneck edges and if we remove these edges *s* and *d* are disconnected. Edges (1, d), (2, d), (2, 4) and (s, 3) are said to be the bottleneck edges in the input graph. If these edges are removed, the source s and destination d are disconnected.

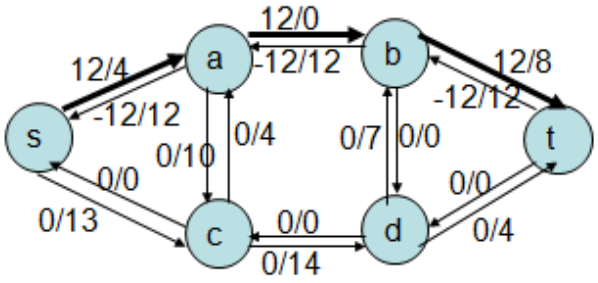
3) Find the maximum flow from the source *s* to the destination *t* in the following graph where the weights of the directed edges indicate the capacity of the edges. In addition, find the minimum set of edges (minimum cut) that when removed from this graph will disconnect *s* and *t*. Show all work.



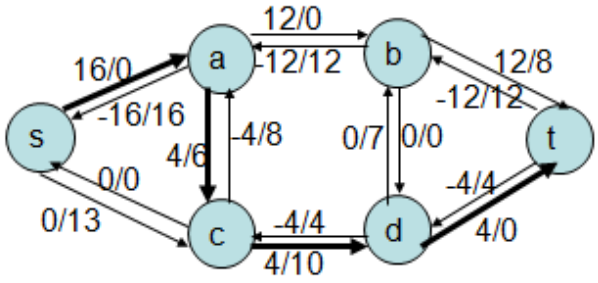
Initial Graph



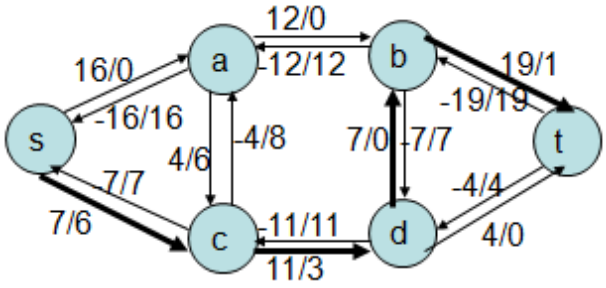
Residual Graph



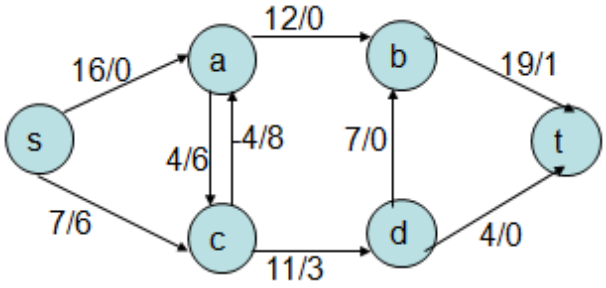
Iteration 1



Iteration 2

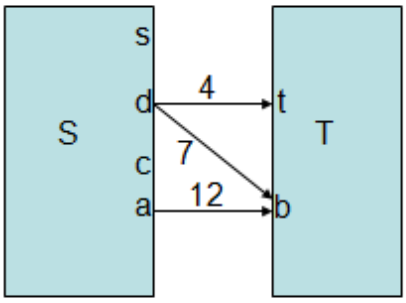


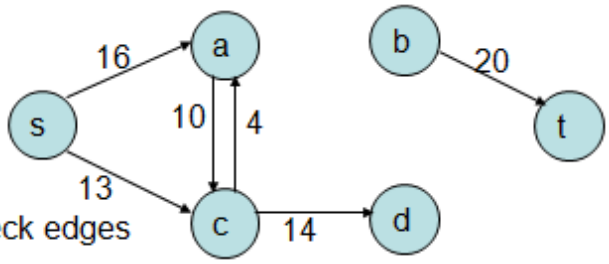
Final Residual Graph: Iteration 3



Final Flow Graph
Maximum Flow = 12 + 4 + 7 = 23

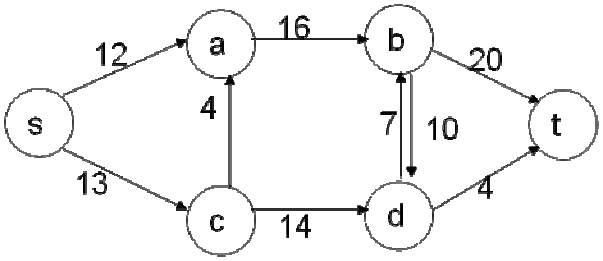
S = {s, a, d, c}
T = {b, t}



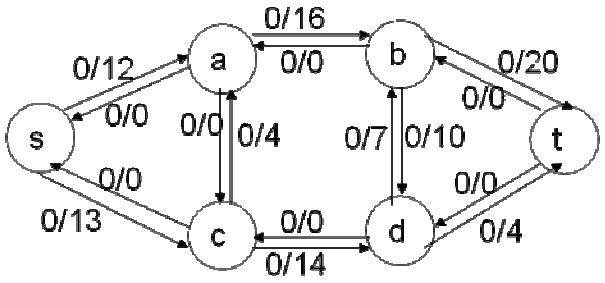


Initial Graph
With bottleneck edges removed
Edges (d, t), (d, b) and (a, b) are the bottleneck edges in the input graph. If these three edges are removed, then the source *s* and the sink *t* are disconnected

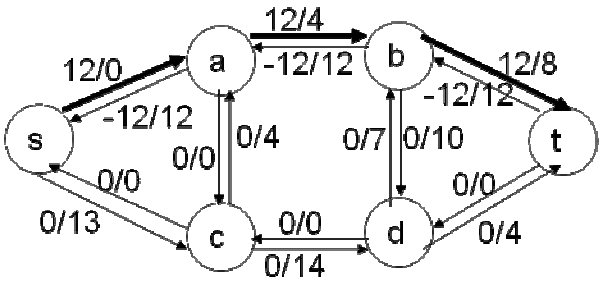
3) (15 points) Apply the Ford-Fulkerson algorithm to determine the maximum flow and the minimum cut for the following network where the edge weights indicate the capacity.



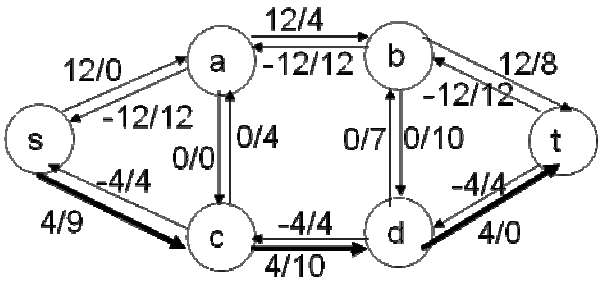
Initial Graph



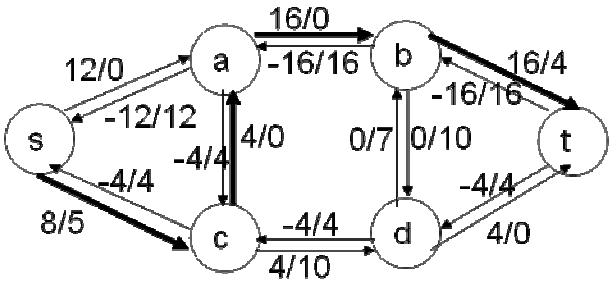
Residual Graph



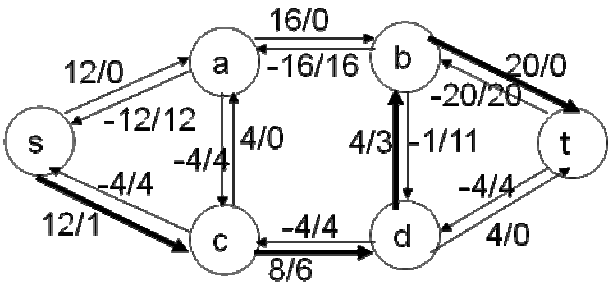
Iteration 1



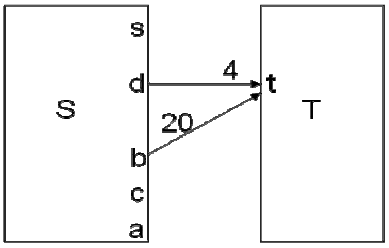
Iteration 2



Iteration 3

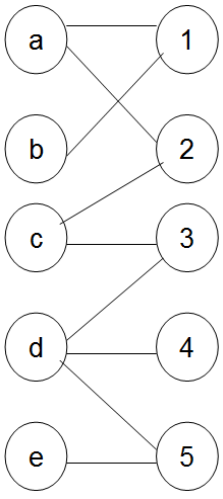


Iteration 4

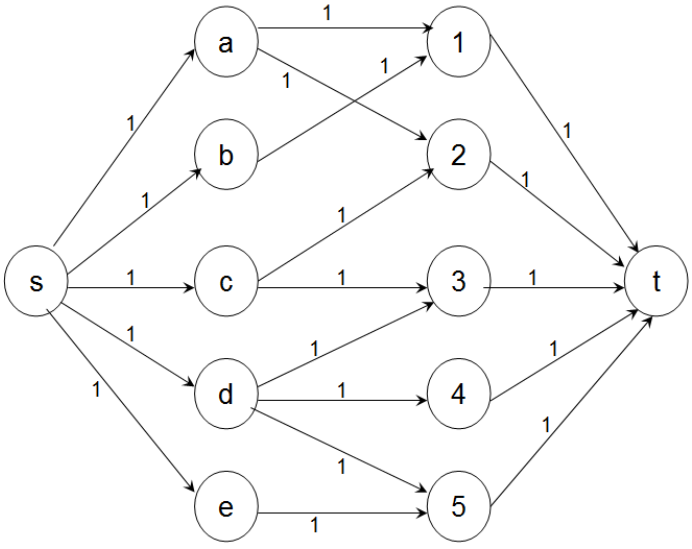


Max. flow = 12 + 4 + 4 + 4 = 24; Min. cut = {(d, t), (b, t)} = 2 edges

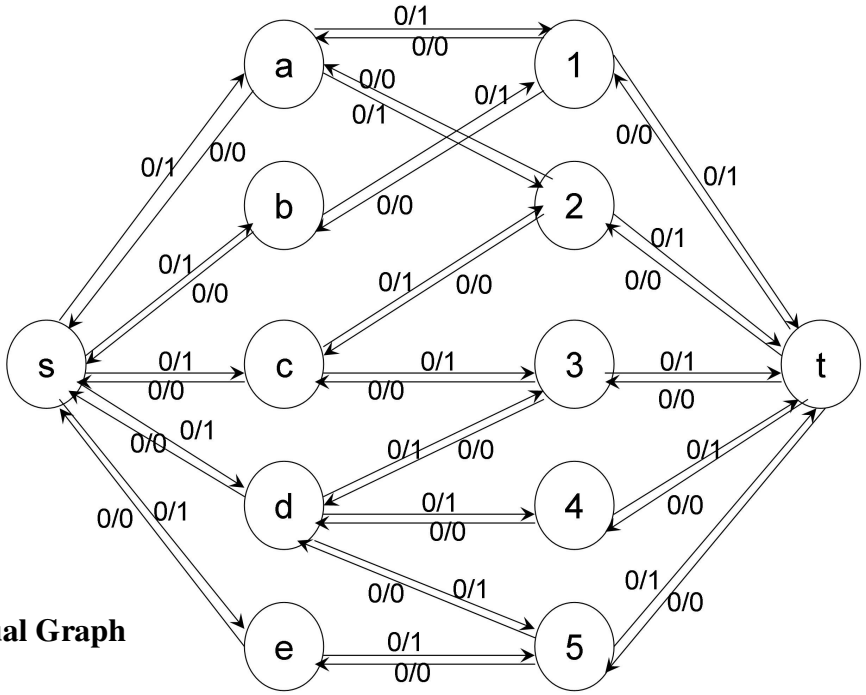
4) Apply the Ford-Fulkerson Algorithm to Compute a Maximum Matching for the following bipartite graph:



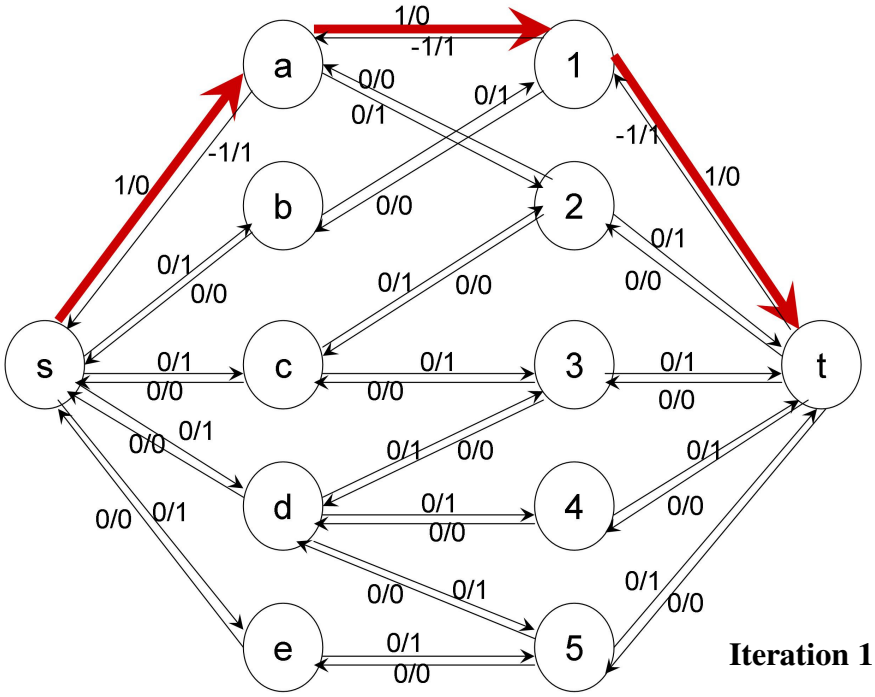
Given Bipartite Graph



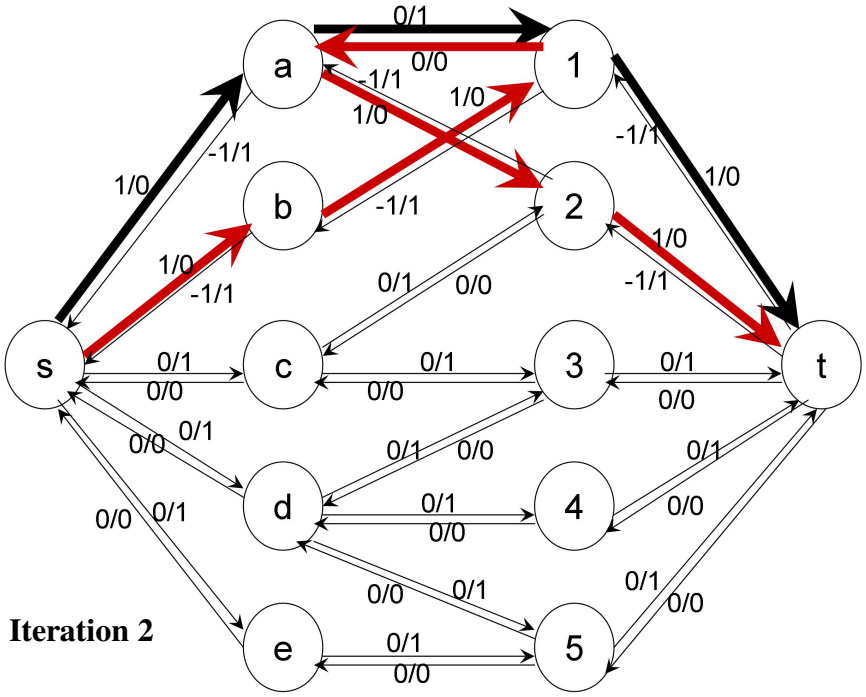
Initialization of the Flow Graph for Ford-Fulkerson Algorithm



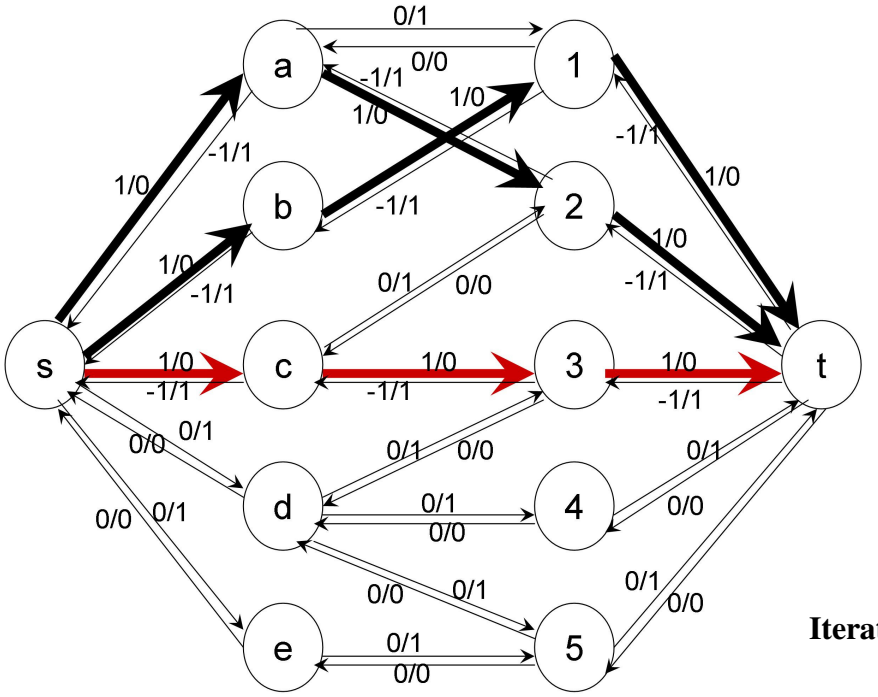
Initial Residual Graph



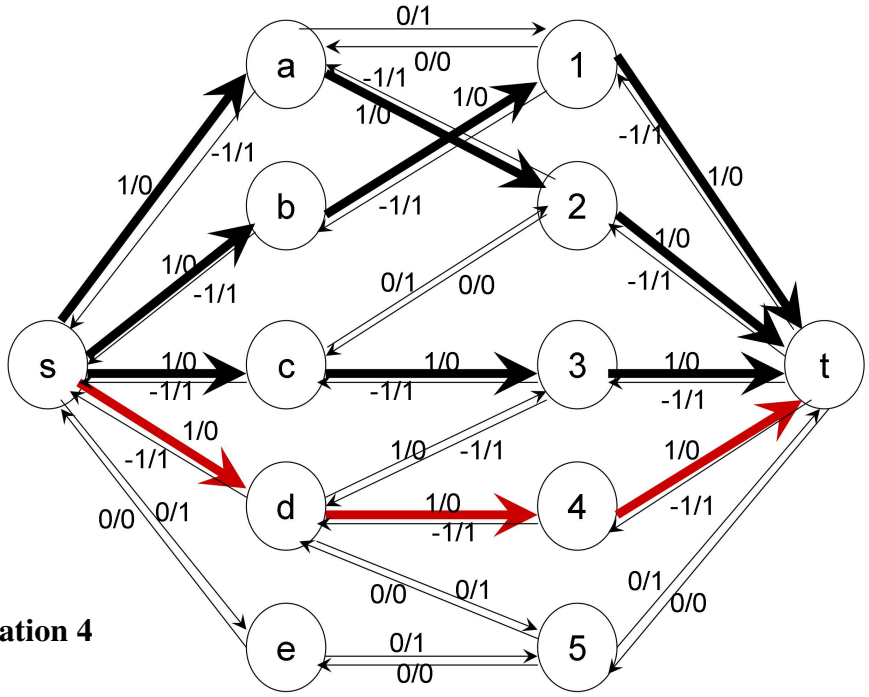
Iteration 1



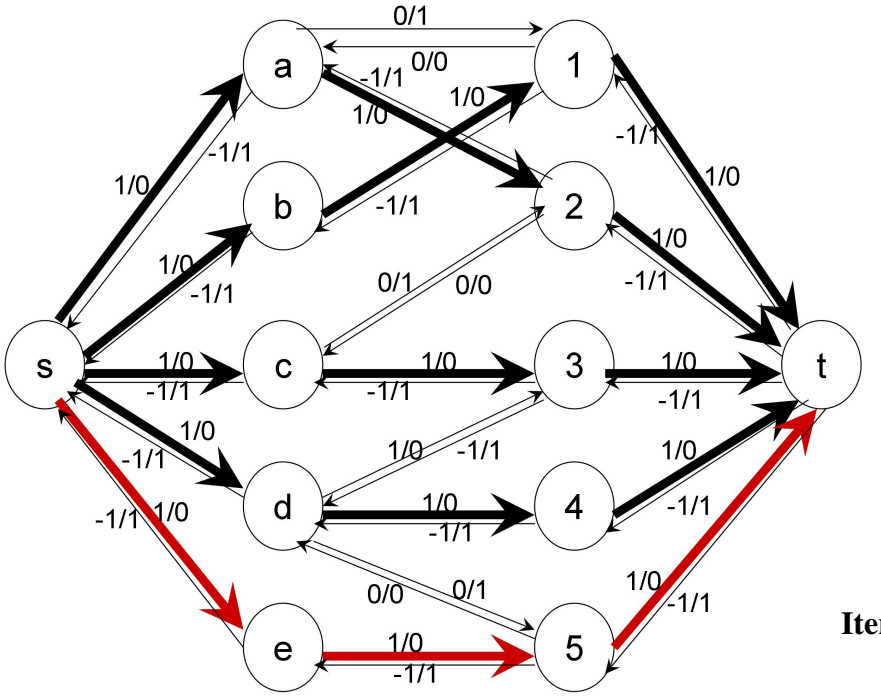
Iteration 2



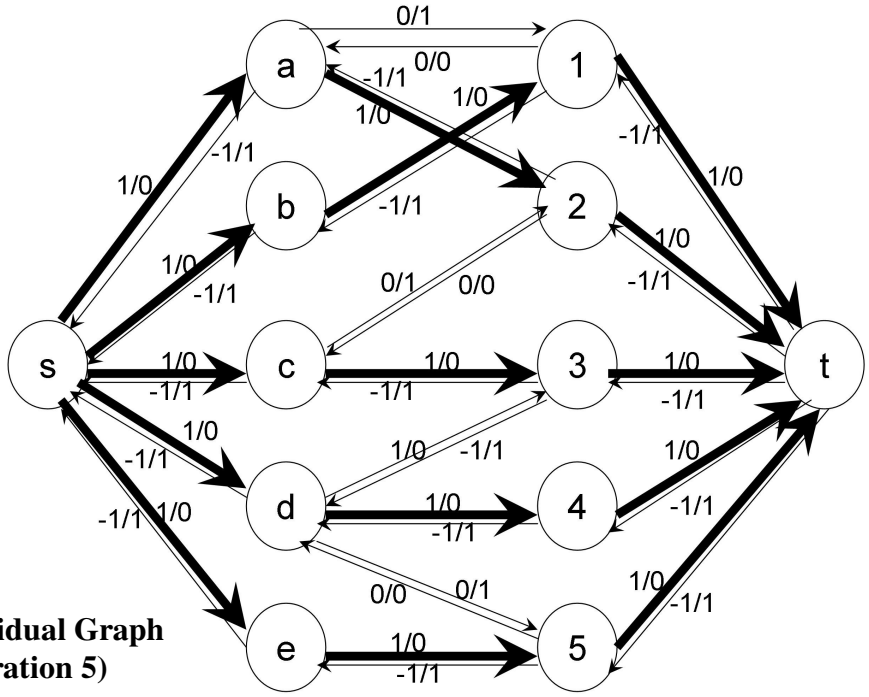
Iteration 3



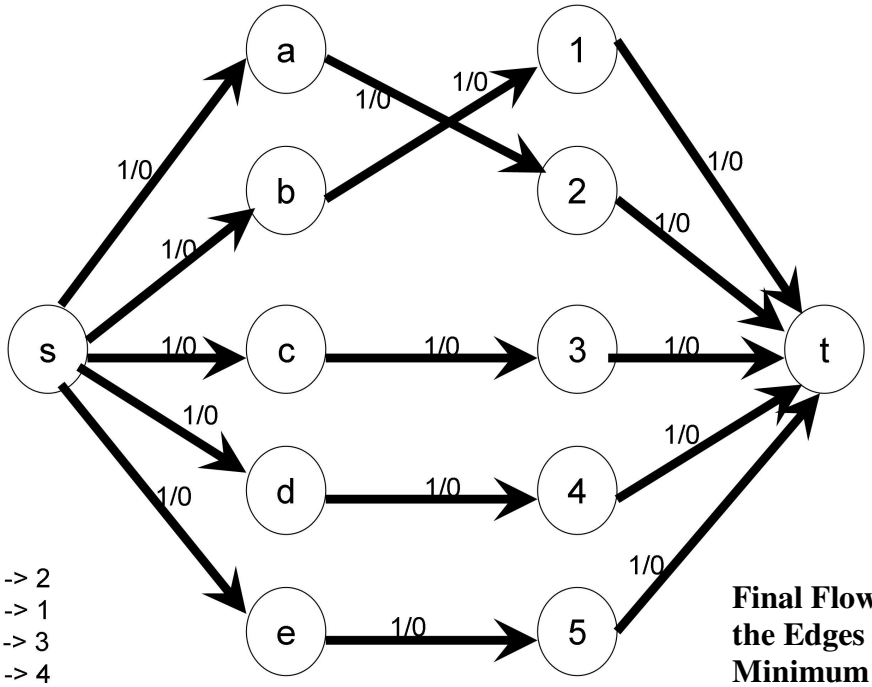
Iteration 4



Iteration 5

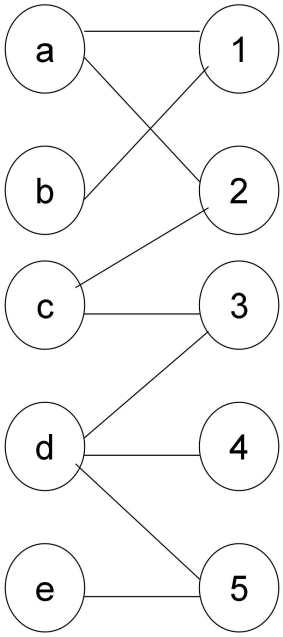


Final Residual Graph (after Iteration 5)

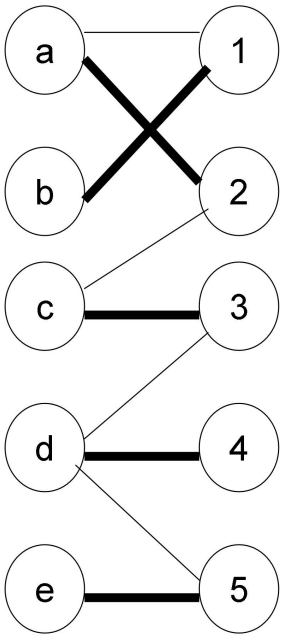


a -> 2
b -> 1
c -> 3
d -> 4
e -> 5

Final Flow Graph with the Edges Forming the Minimum Cut

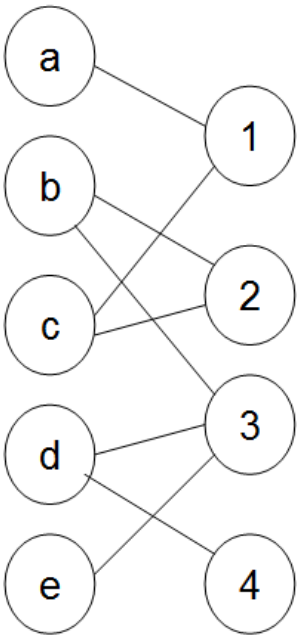


Bipartite Graph

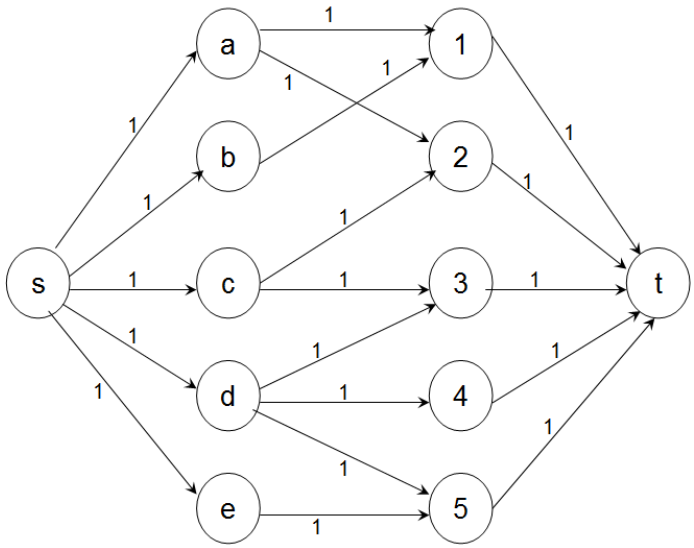


Maximum Matching on the Bipartite Graph (5 edges)

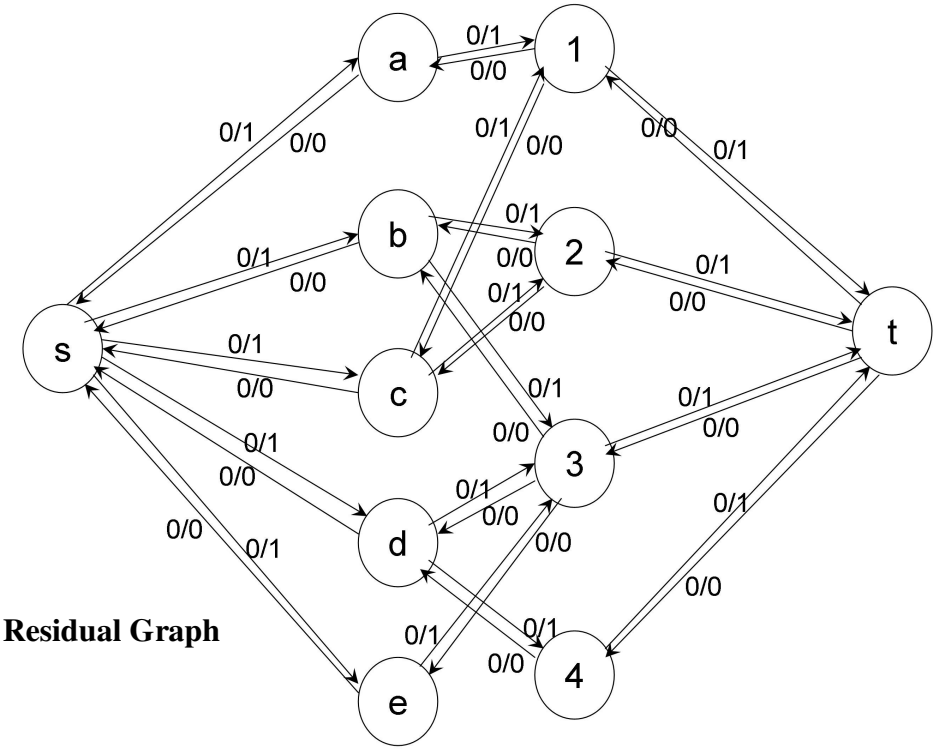
5) Apply the Ford-Fulkerson Algorithm to Compute a Maximum Matching for the following bipartite graph:



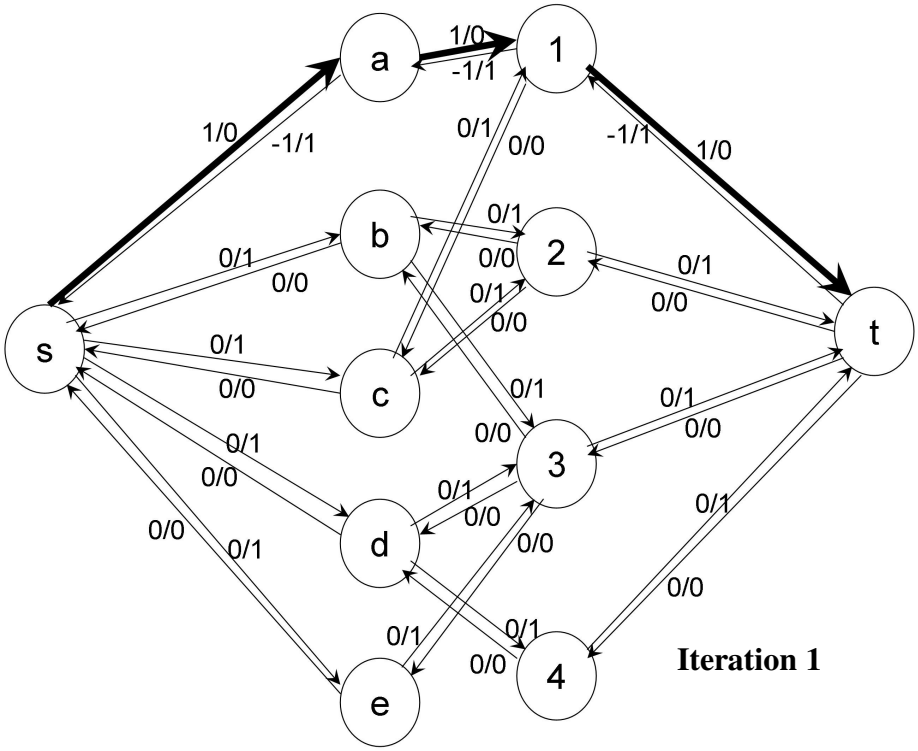
Given Bipartite Graph



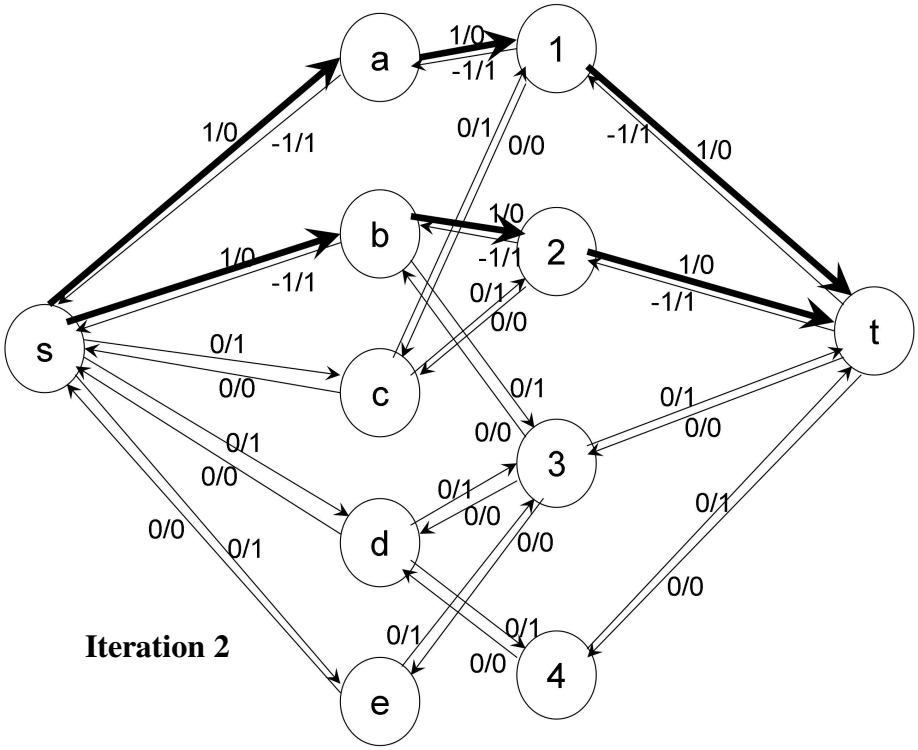
Initialization of the Flow Graph for Ford-Fulkerson Algorithm



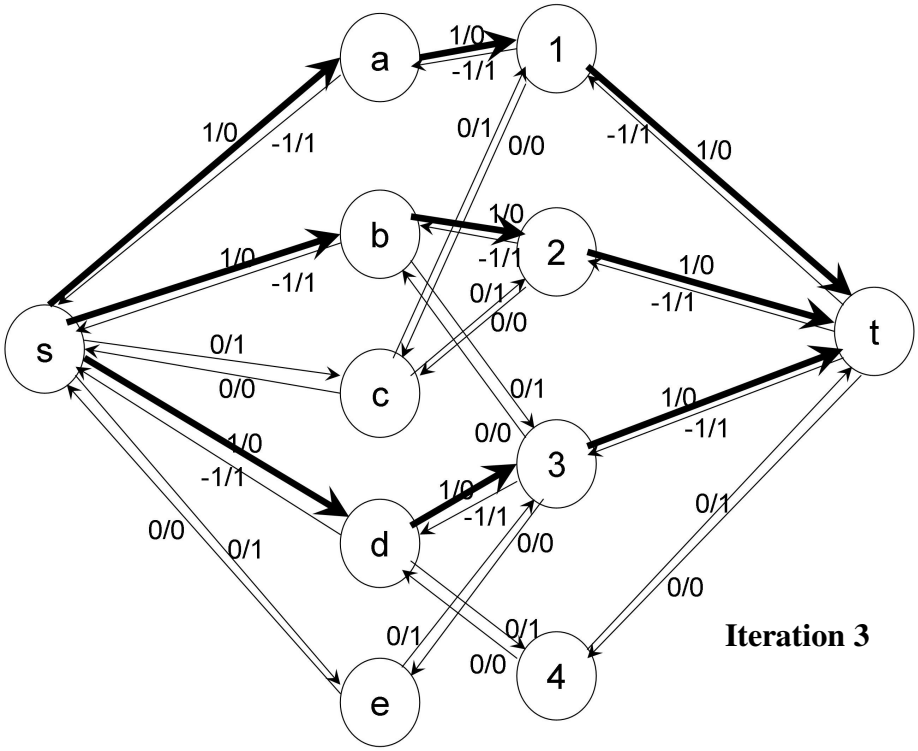
Initial Residual Graph



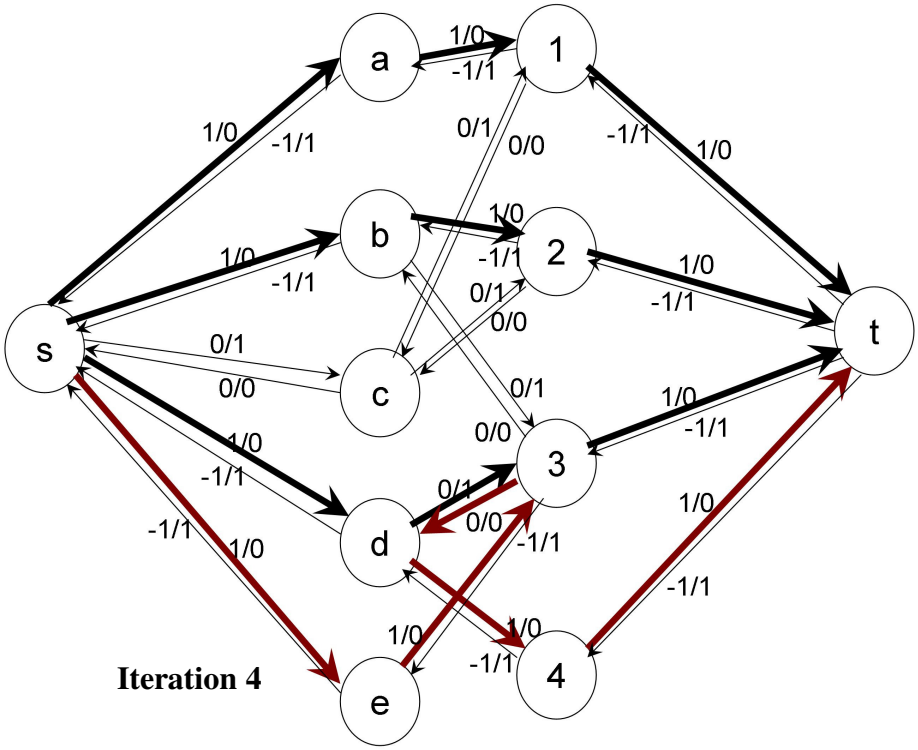
Iteration 1



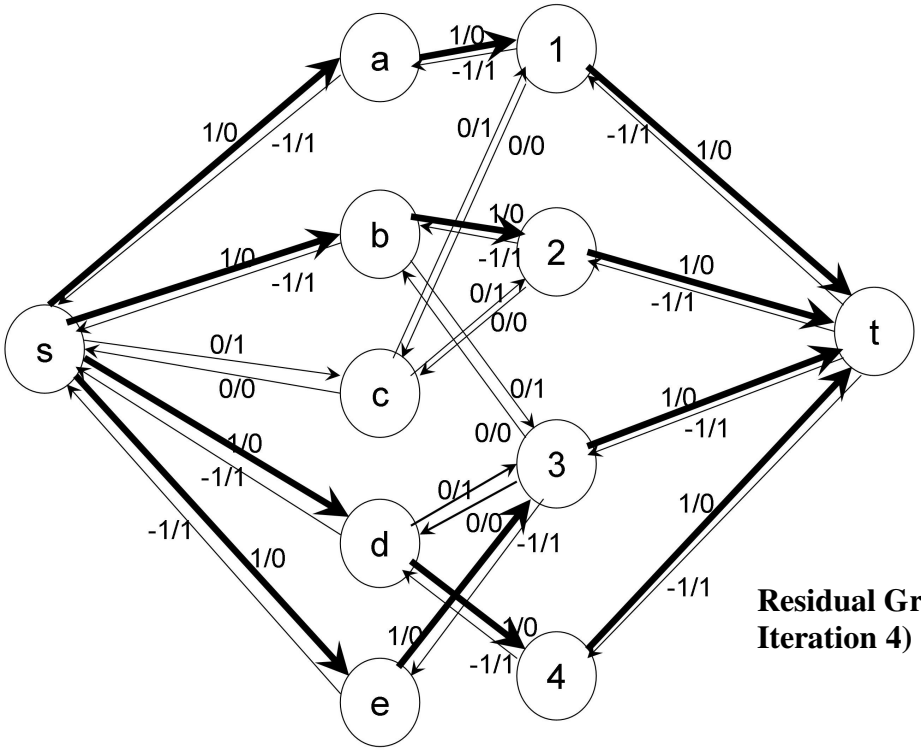
Iteration 2



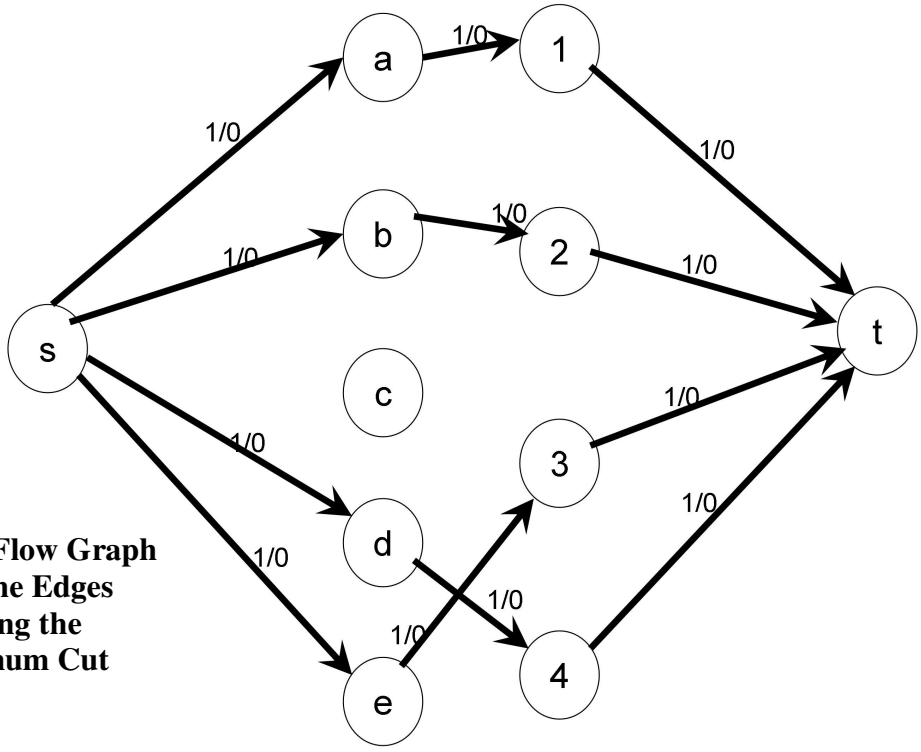
Iteration 3



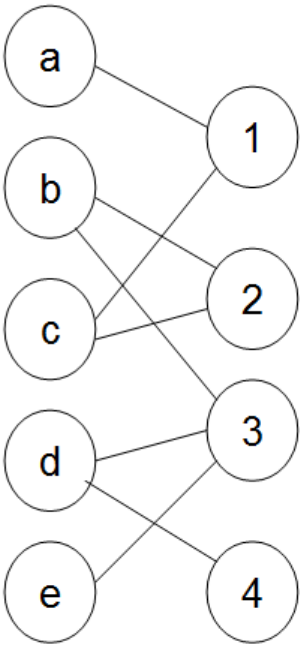
Iteration 4

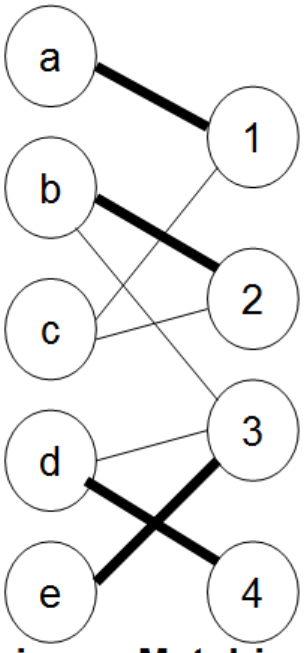


Residual Graph (after Iteration 4)



Final Flow Graph with the Edges Forming the Minimum Cut





Maximum Matching on the Bipartite Graph (4 edges)