COMP7500 – Assignment 1

Joel Thomas 44793203

Part A

1.

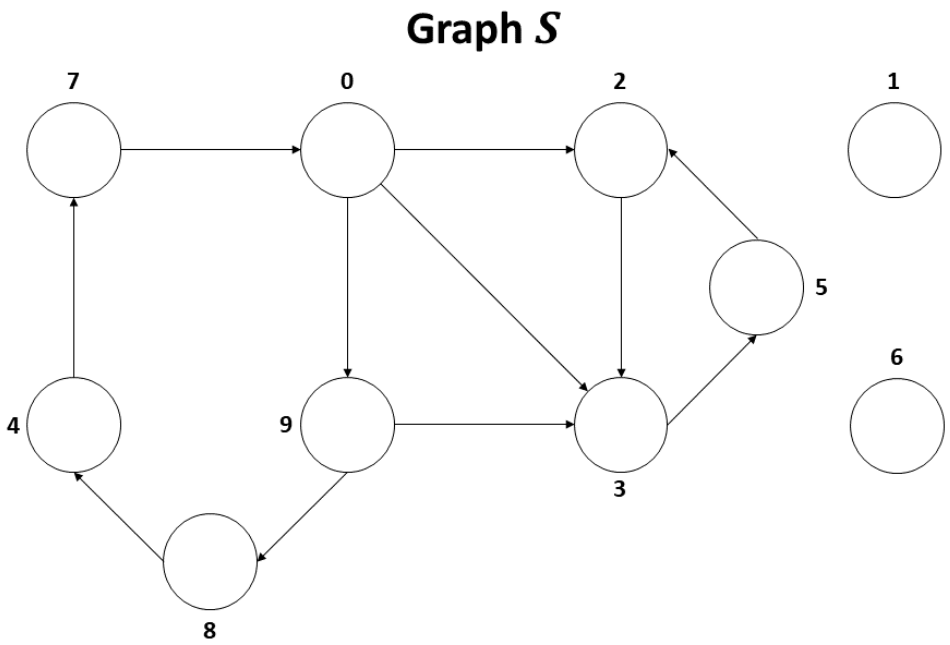
a)

The input student number used is 44793203. Hence, the 12-digit initial input number after prefixing by "98" and postfixing by "52" is given by 984479320352. After running the provided algorithm, the resulting value of d that forms the 12-digit SNI is given by 984709320352. Note that the only differences are at indices 4 and 5 (using the same indexing system starting from 1). This is because:

- $d[4] = d[3] = 4 \rightarrow d[4] = (d[4] + 3)\ mod\ 10 = (4 + 3)\ mod\ 10 = 7$
- $d[5] = d[4] = 7 \rightarrow d[5] = (d[5] + 3)\ mod\ 10 = (7 + 3)\ mod\ 10 = 0$

b)

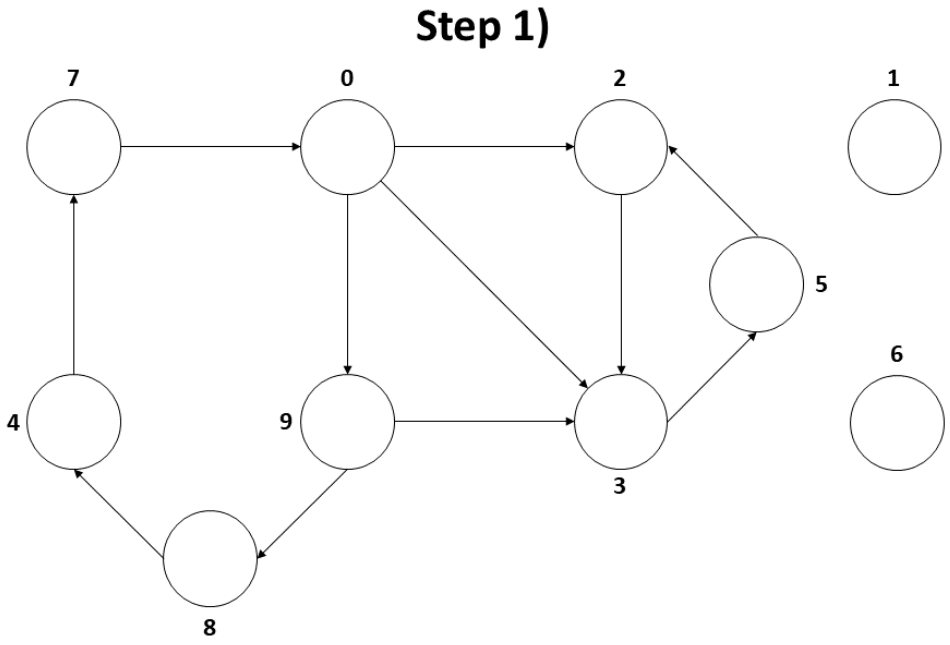
The graph S, constructed using the SNI from **a)**, with nodes represented by all the digits from $0,1, \dots ,9$ is provided below:

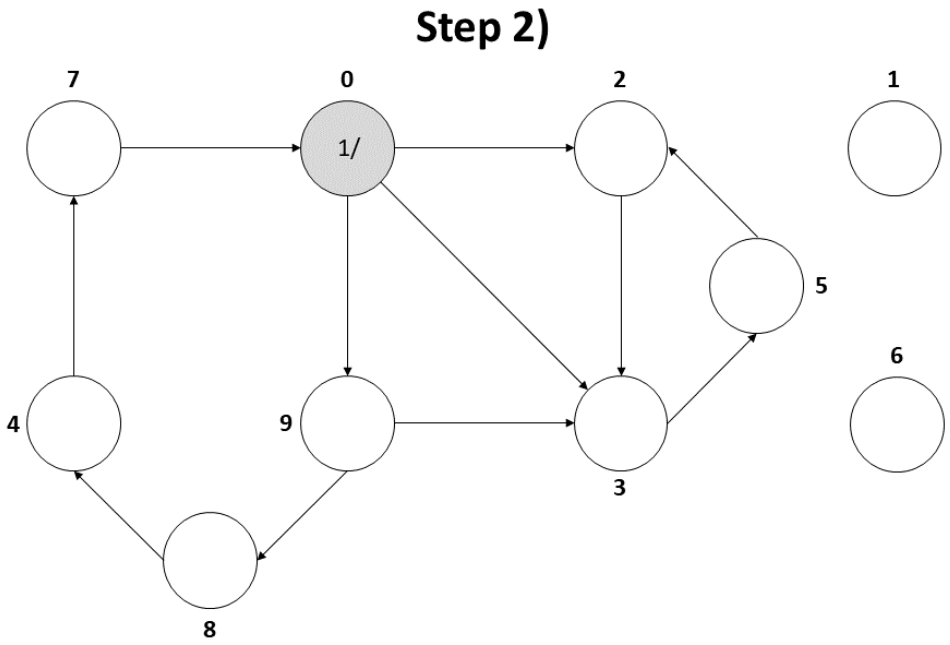


2.

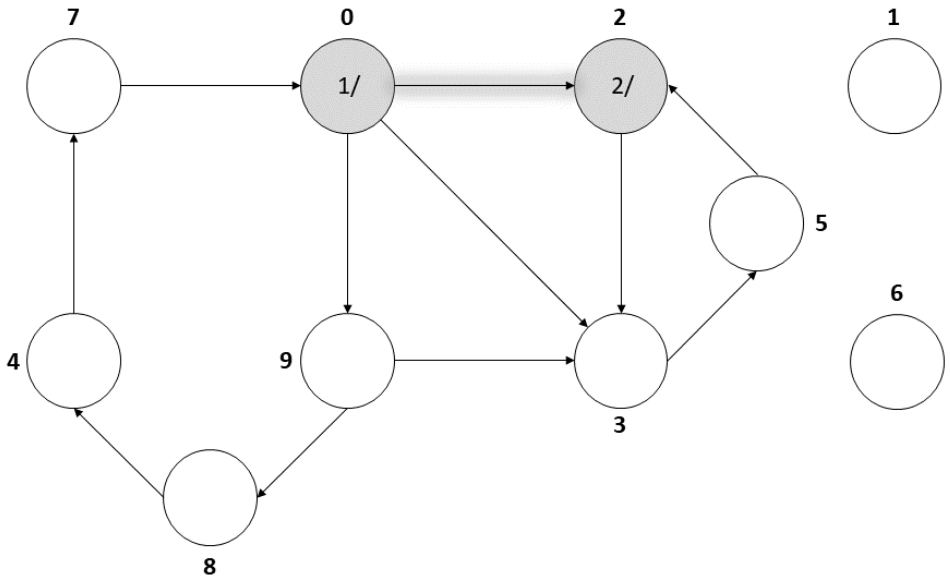
a)

The resulting output from line 1 of the SCC algorithm using S as input is provided below:

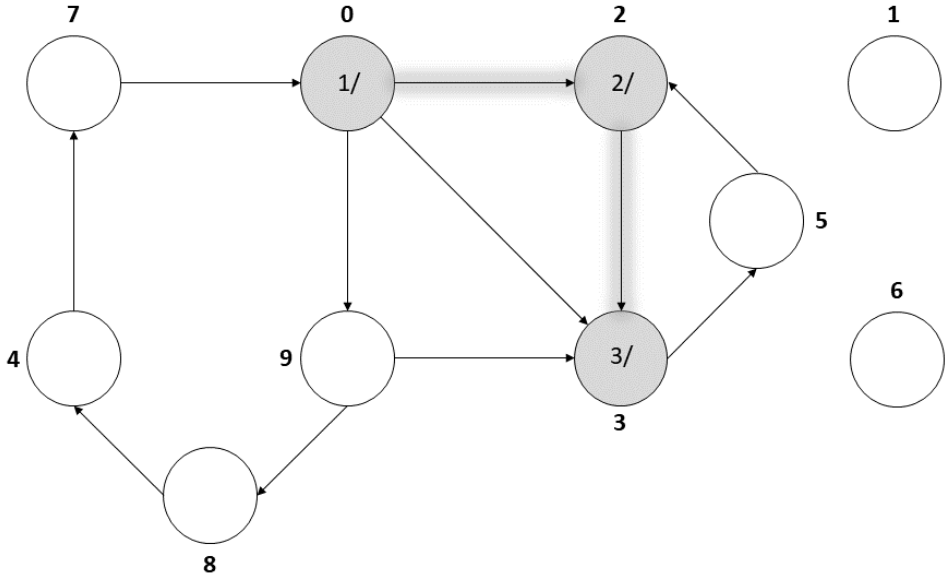




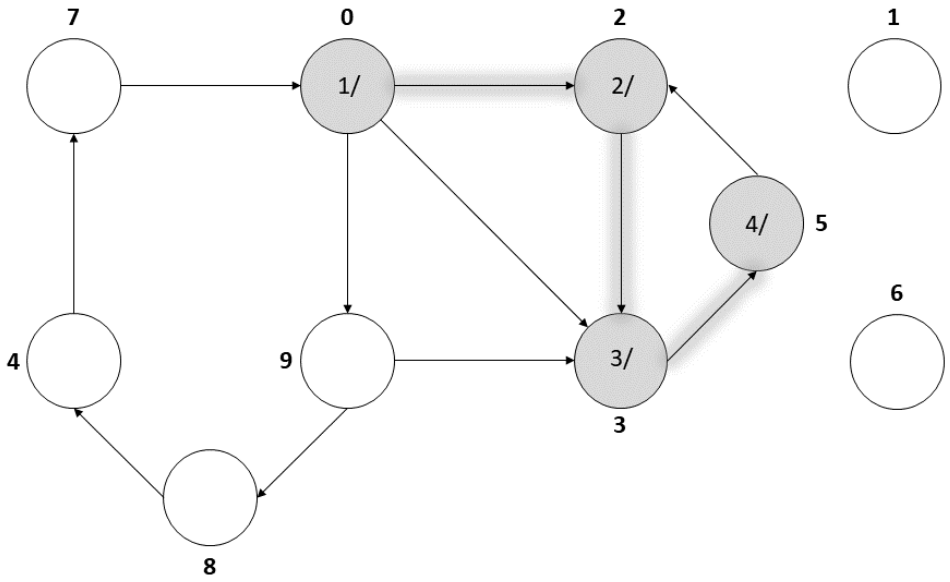
Step 3)



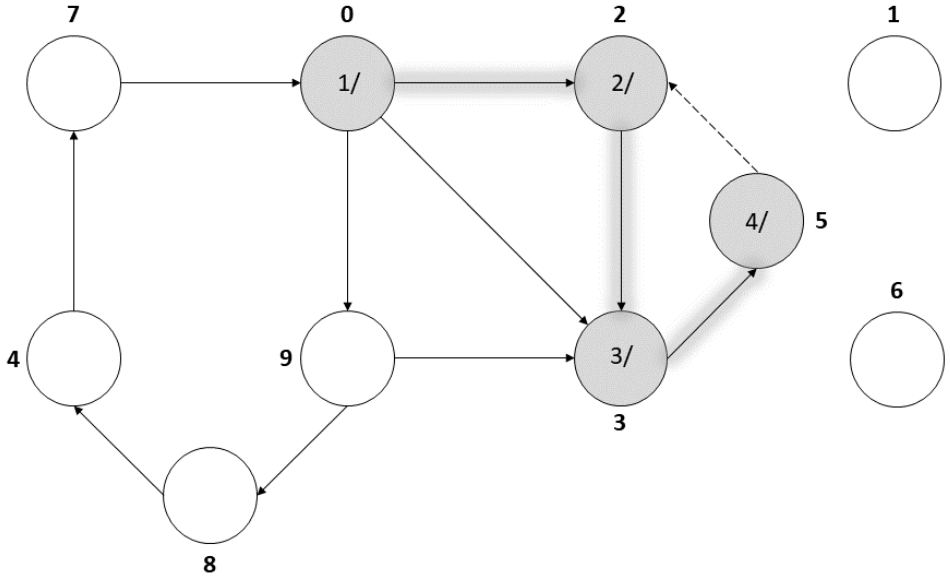
Step 4)



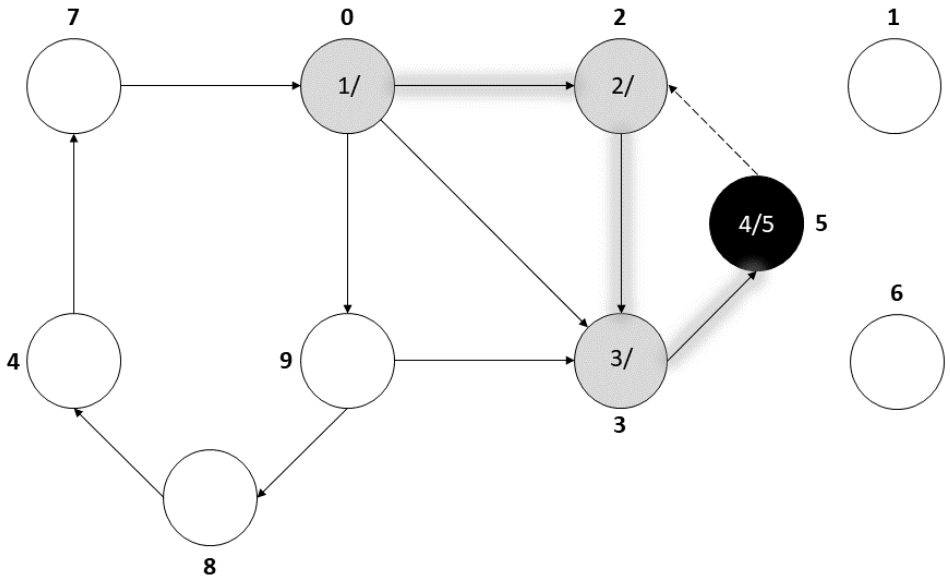
Step 5)



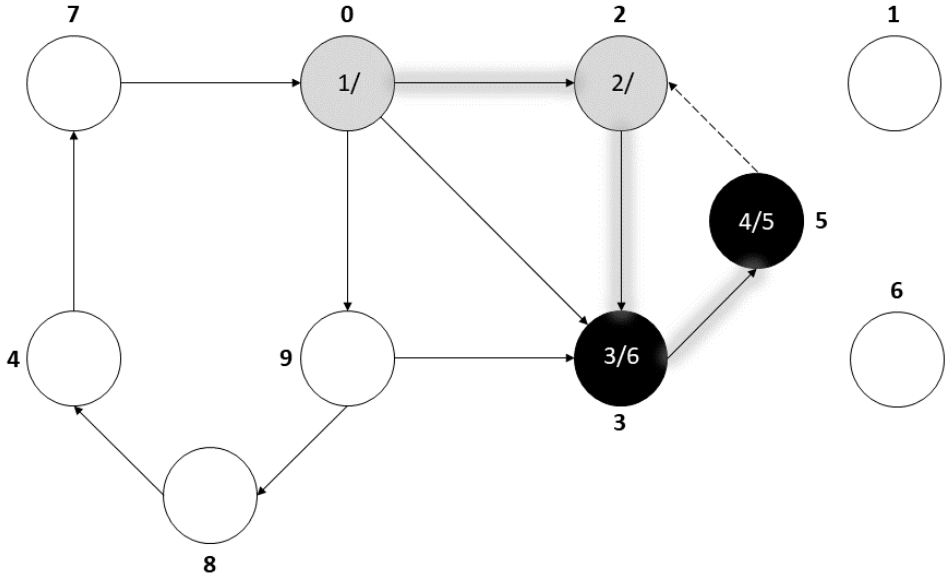
Step 6)



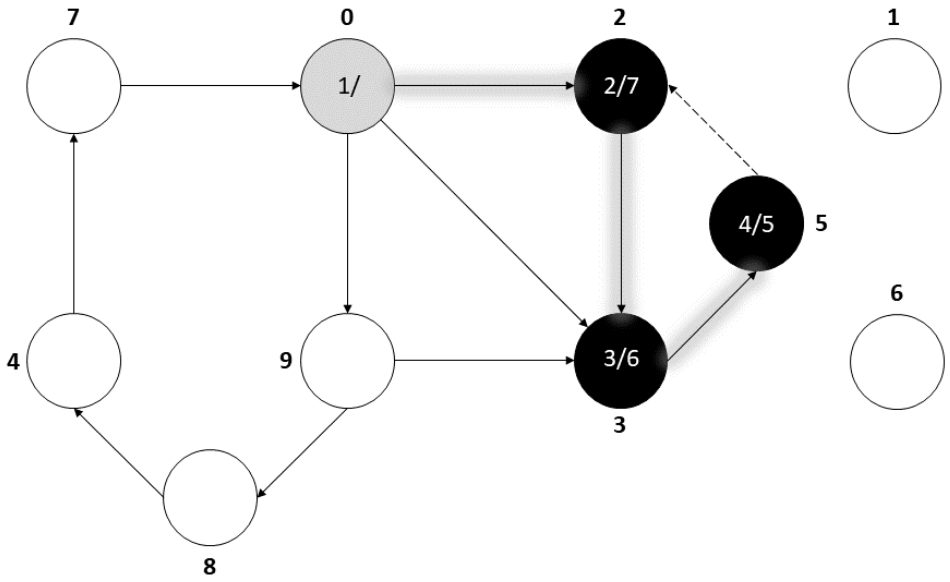
Step 7)



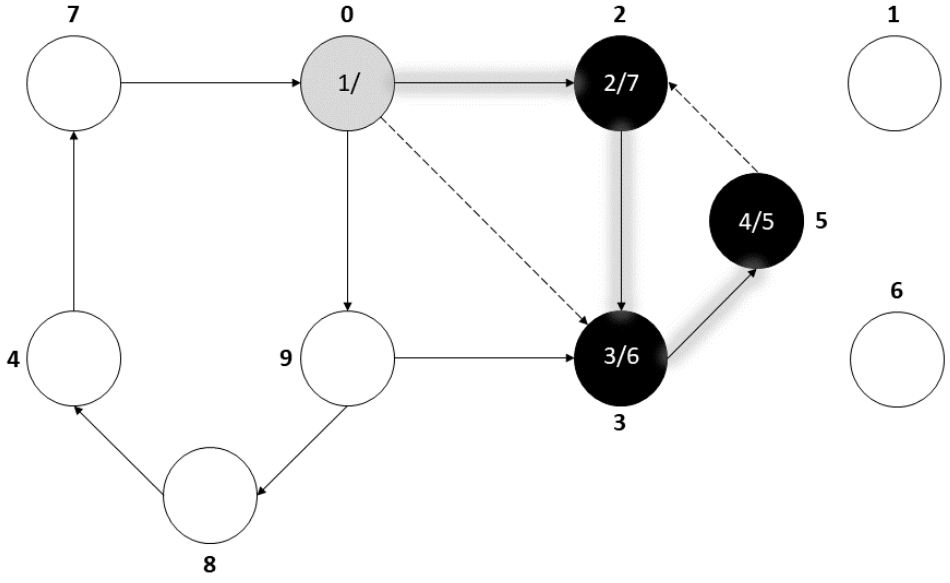
Step 8)



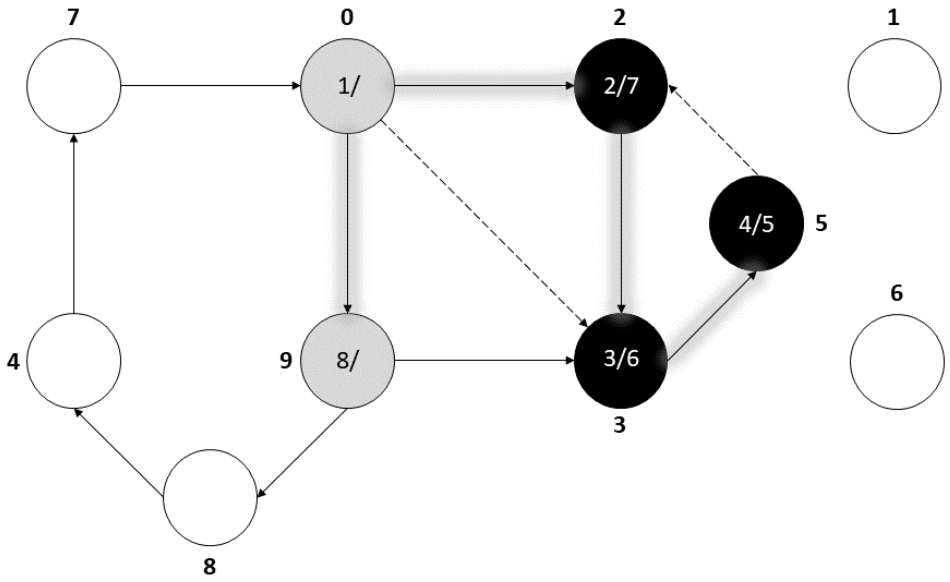
Step 9)



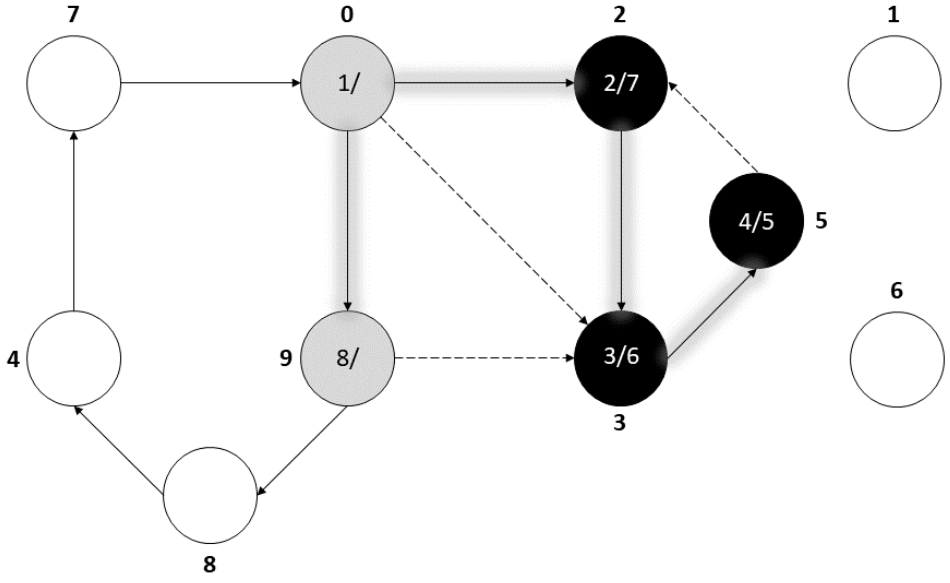
Step 10)



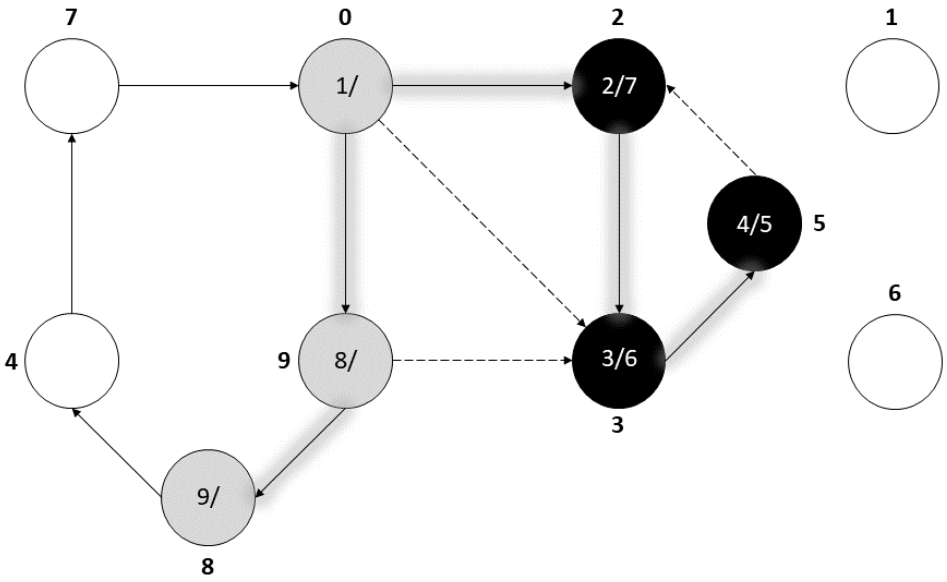
Step 11)



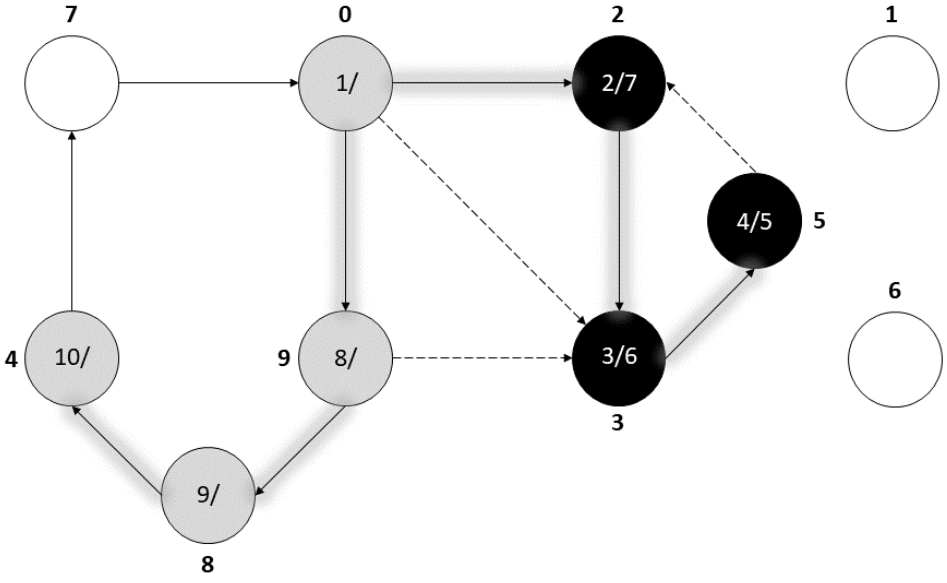
Step 12)



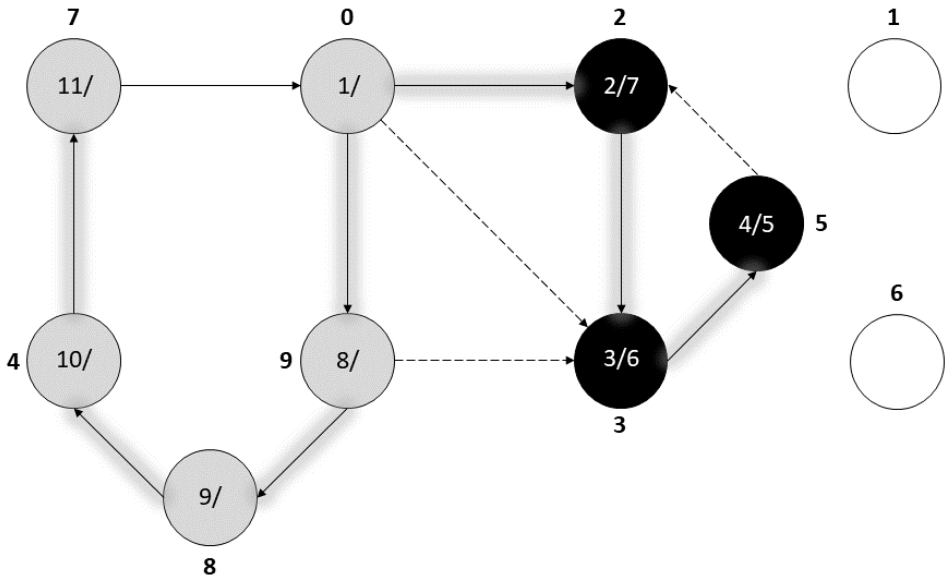
Step 13)



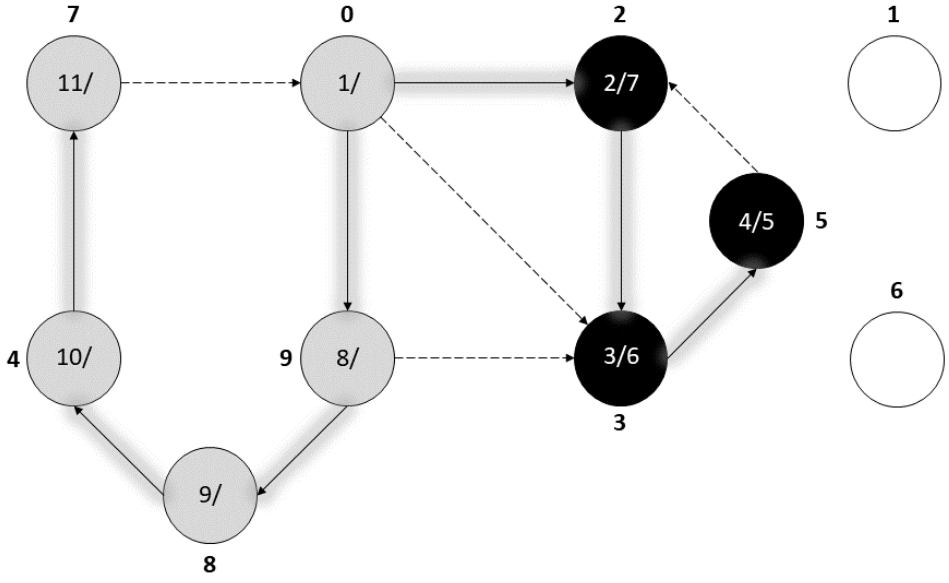
Step 14)



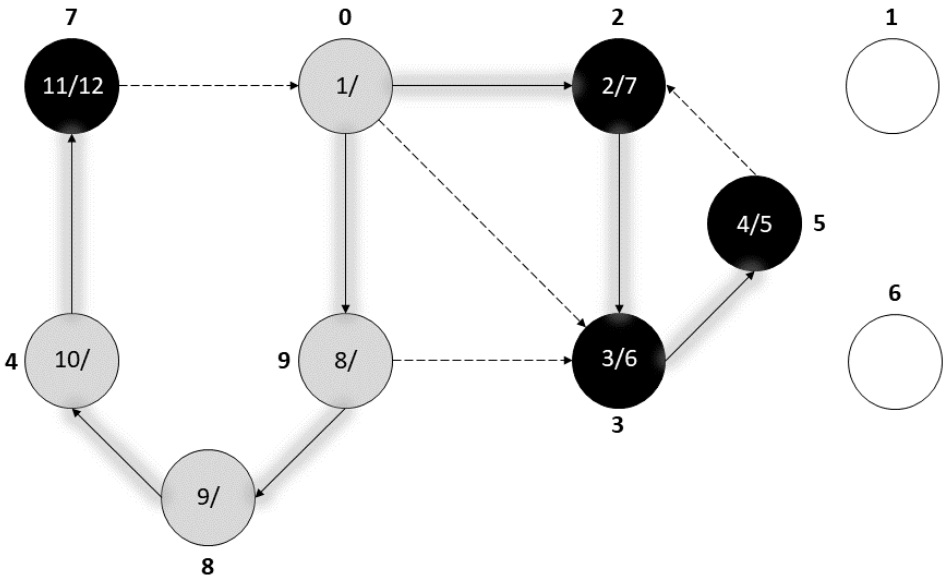
Step 15)



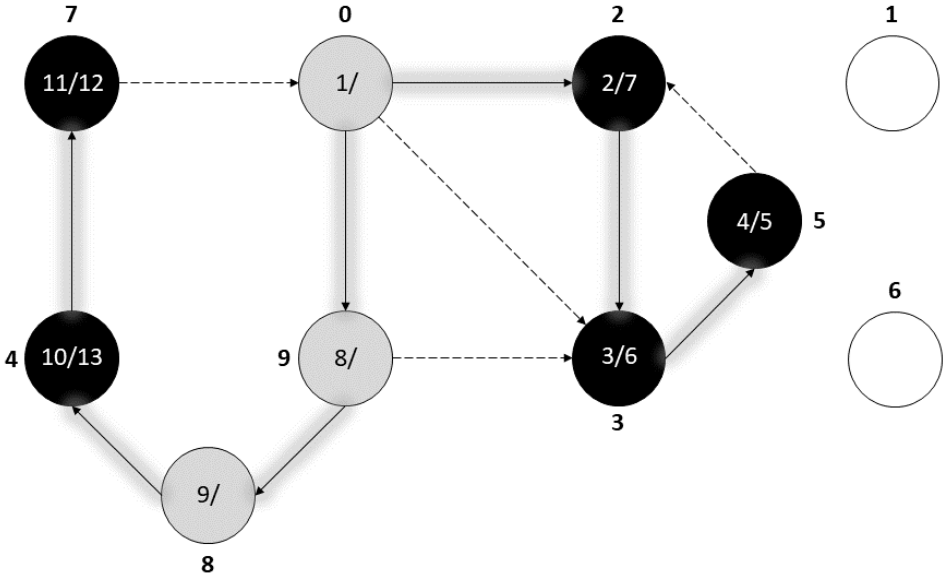
Step 16)



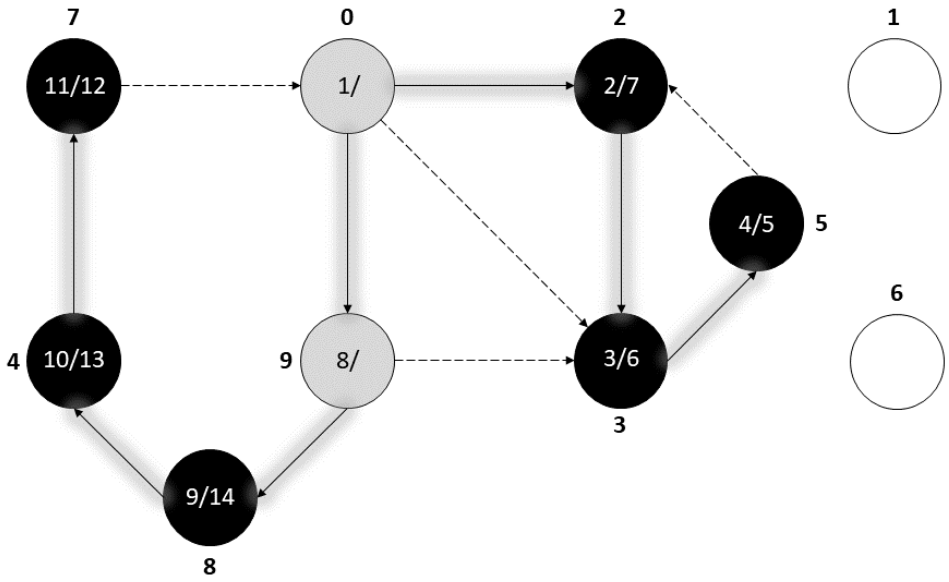
Step 17)



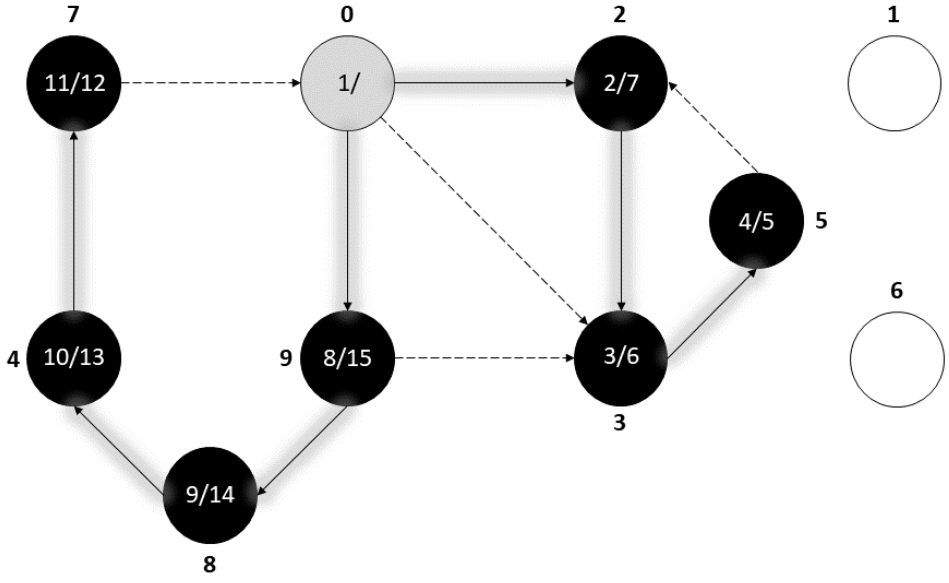
Step 18)



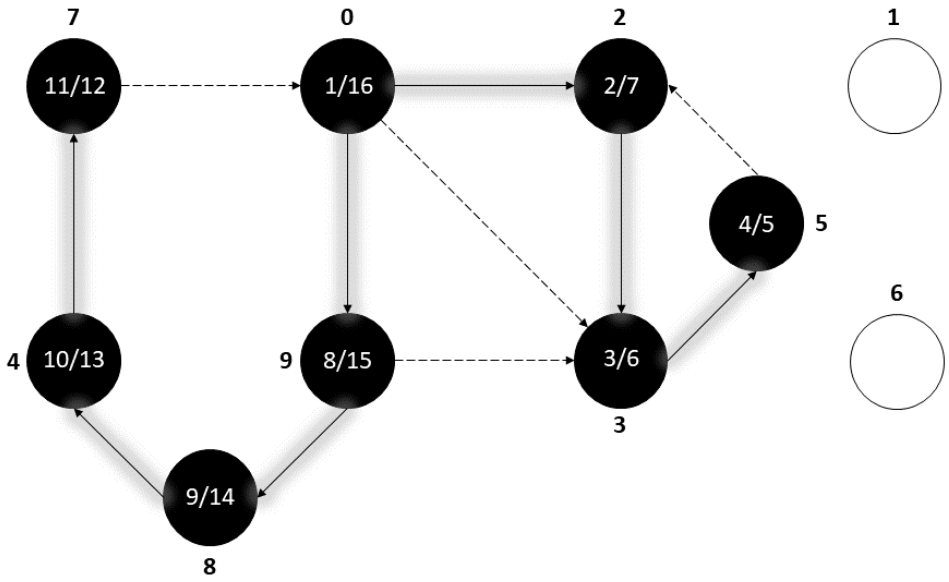
Step 19)



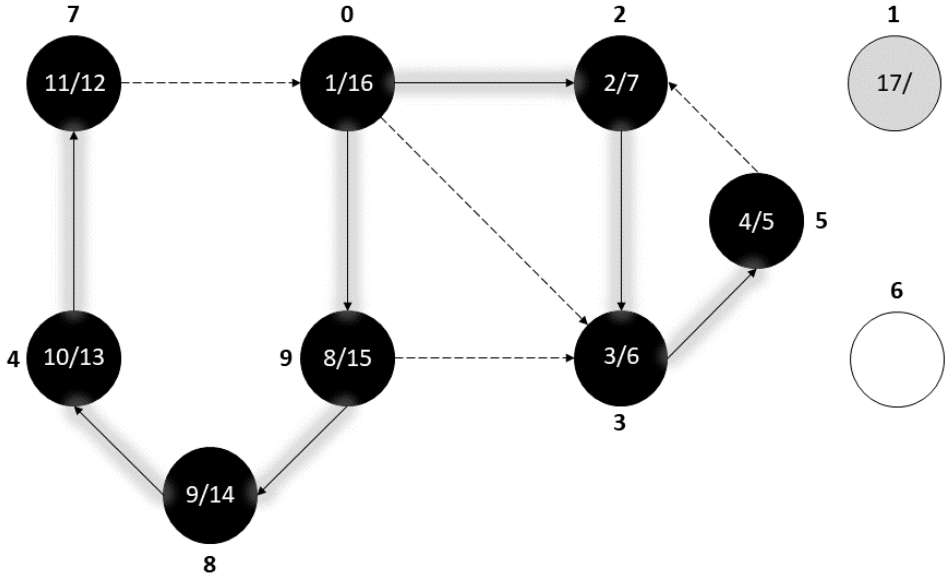
Step 20)



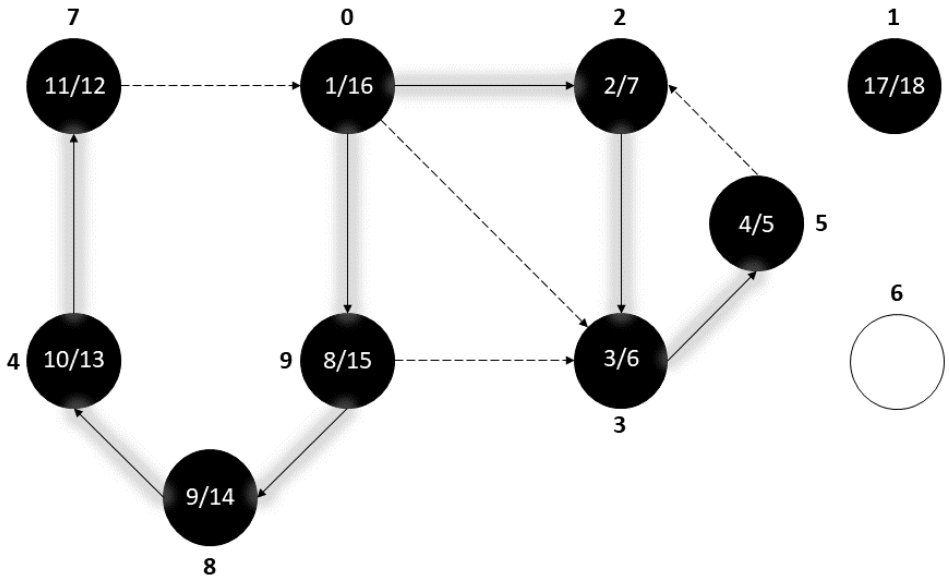
Step 21)



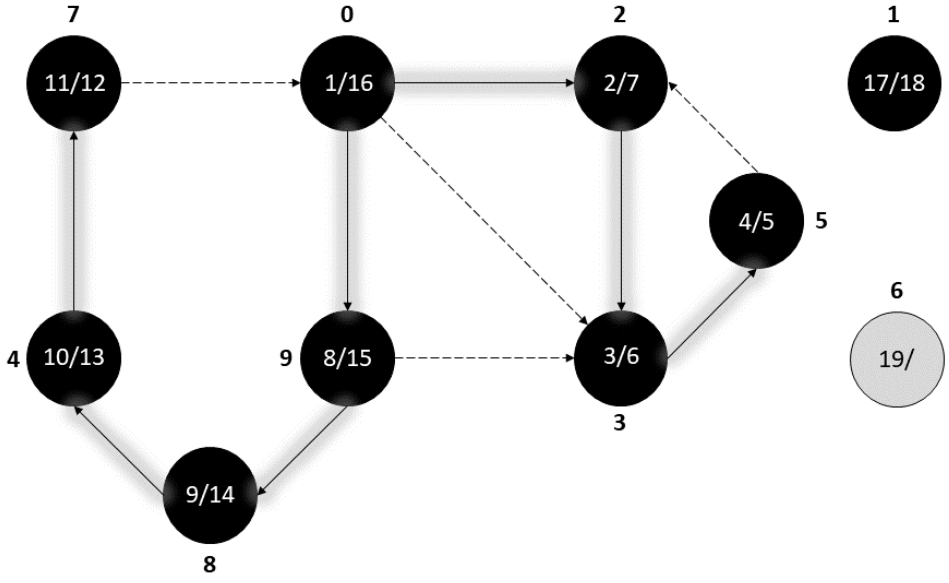
Step 22)



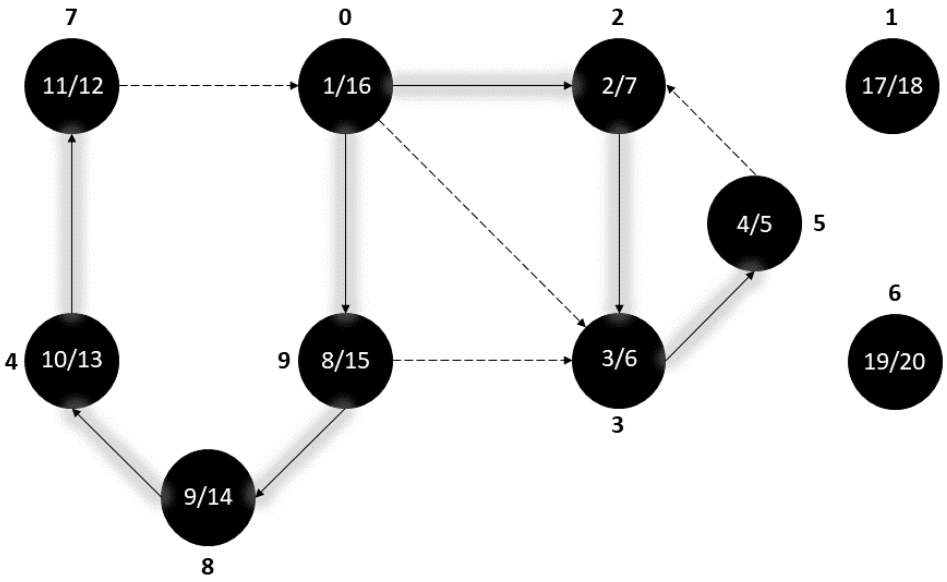
Step 23)



Step 24)

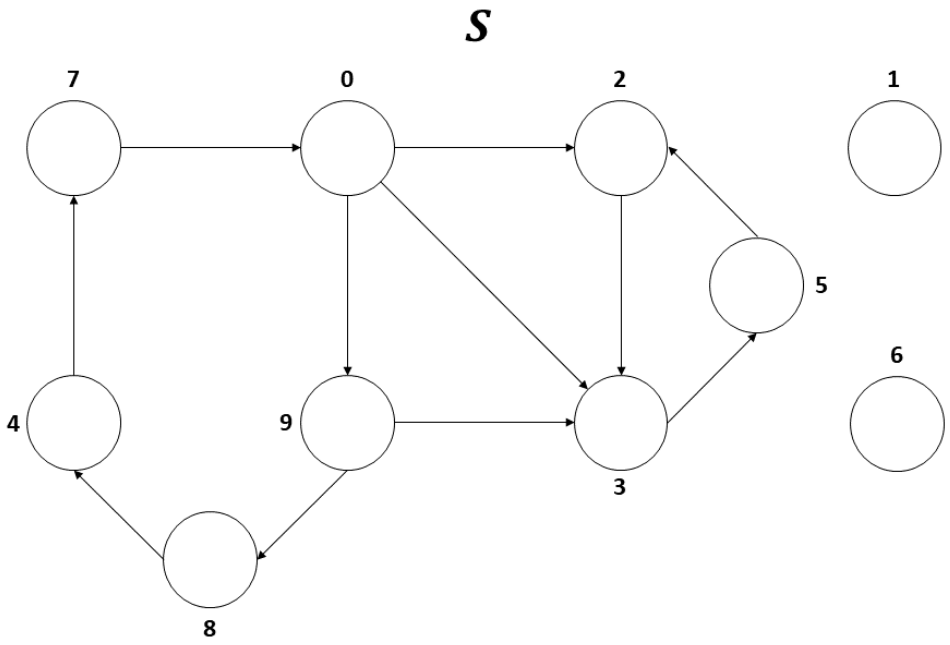


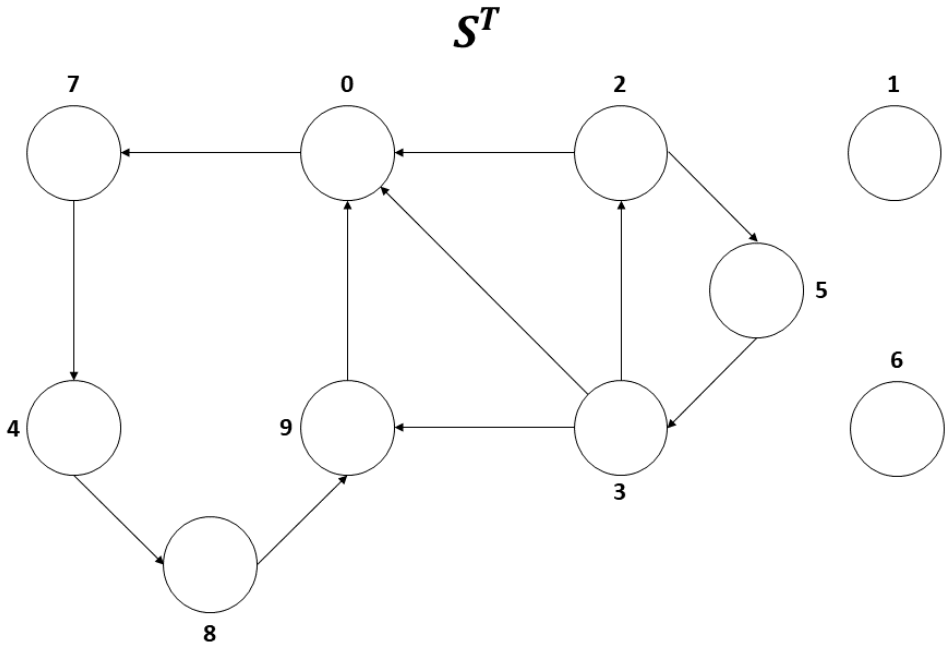
Step 25)



b)

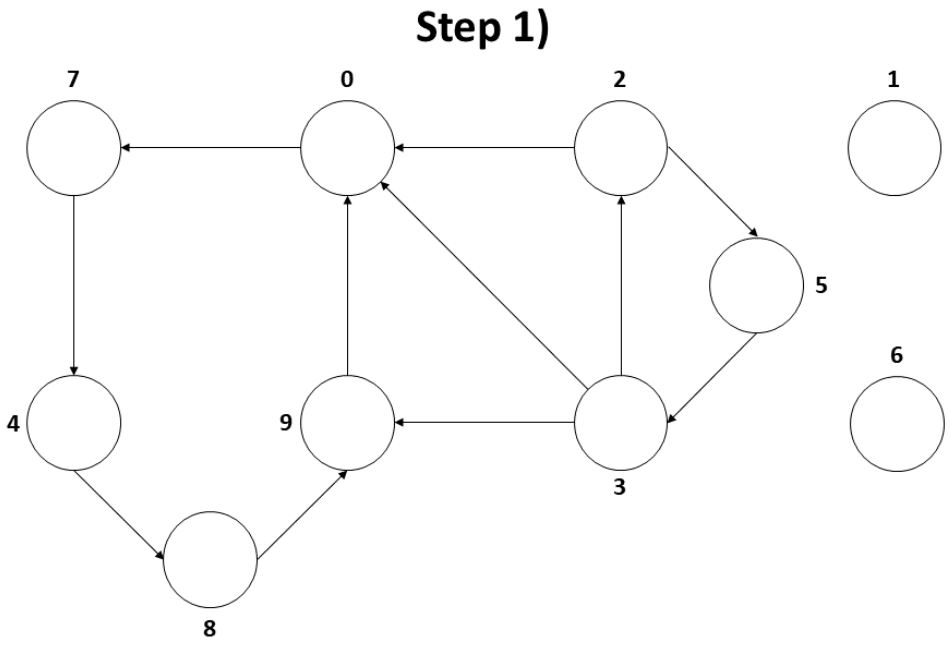
The resulting output from line 2 of the SCC algorithm using S as input is provided below:

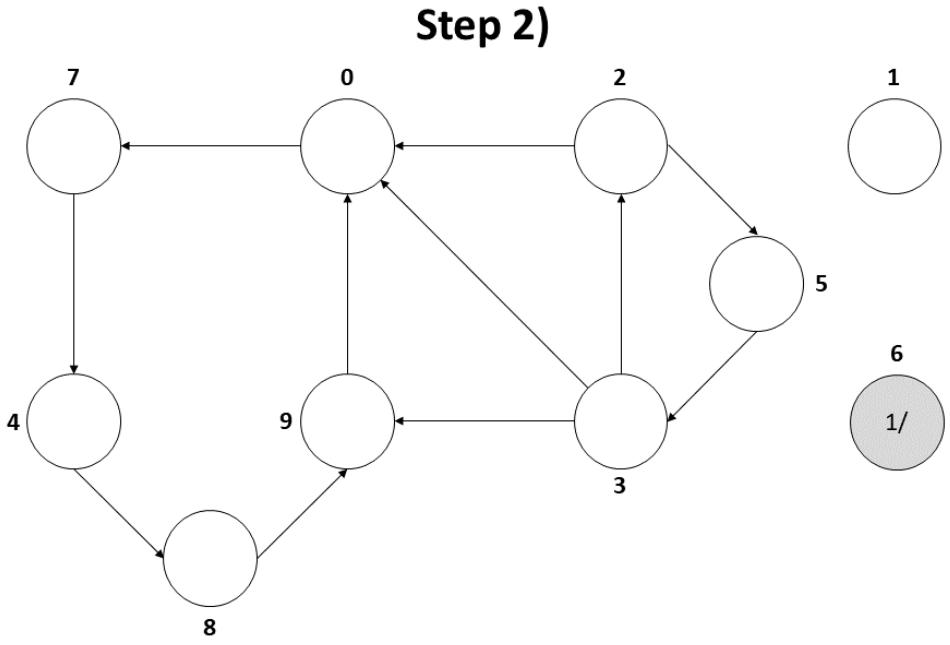




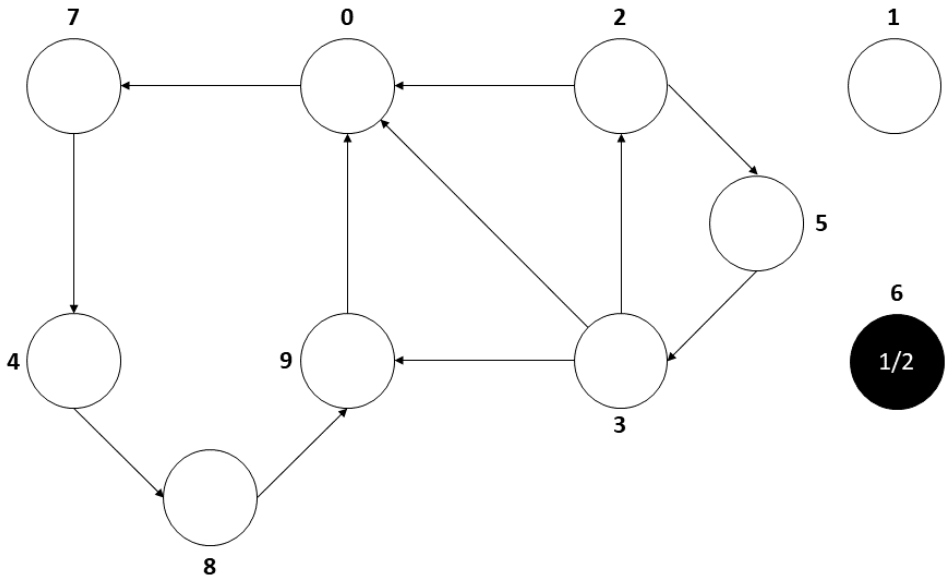
c)

The resulting output from line 3 of the SCC algorithm using S as input is provided below:

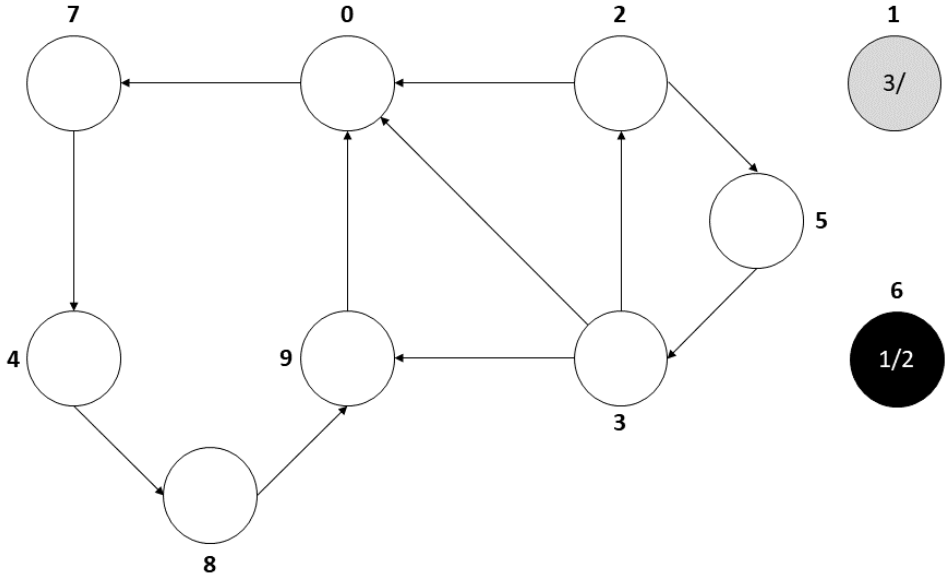




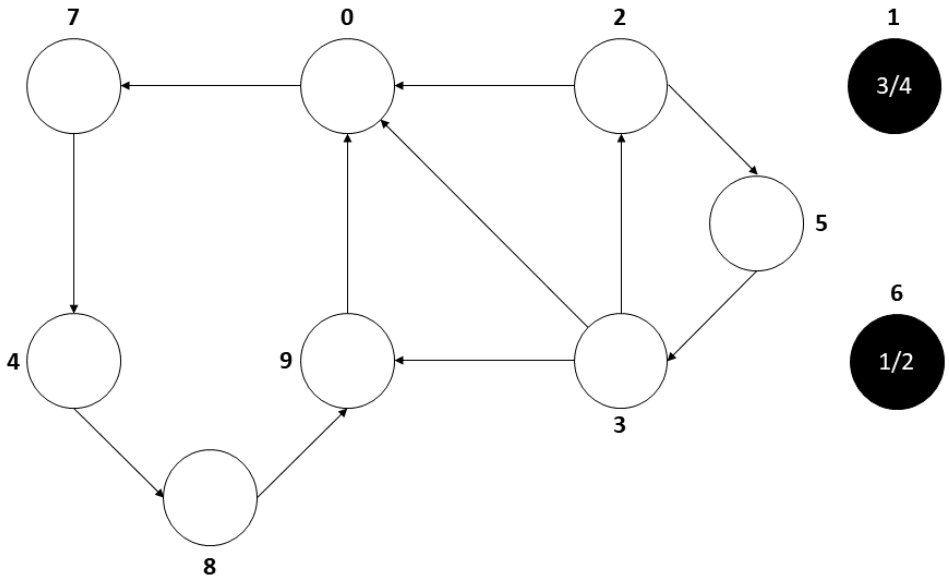
Step 3)



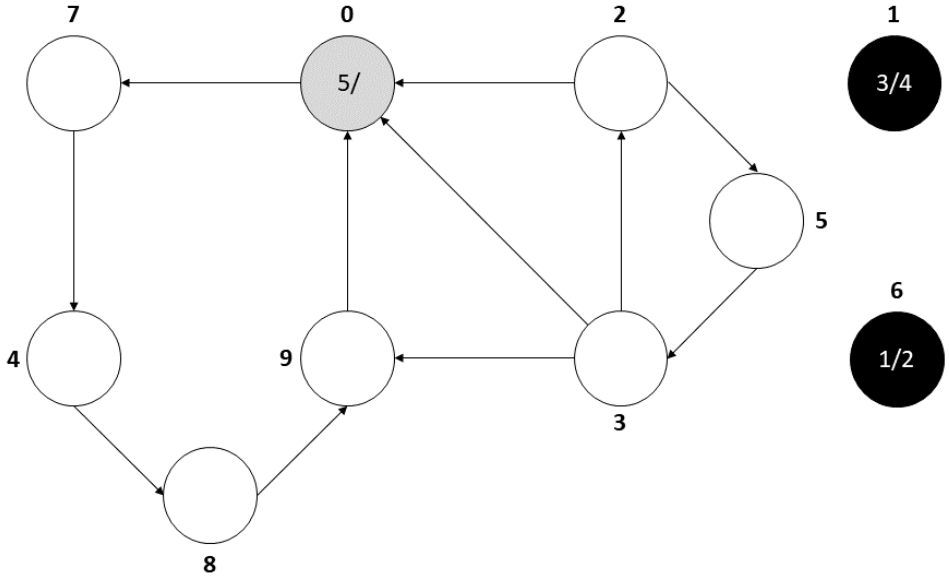
Step 4)



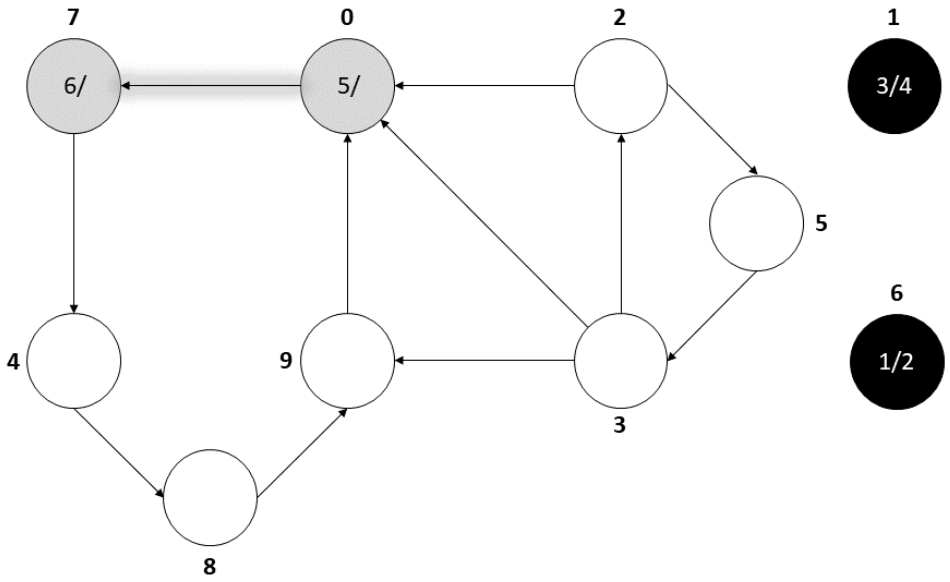
Step 5)



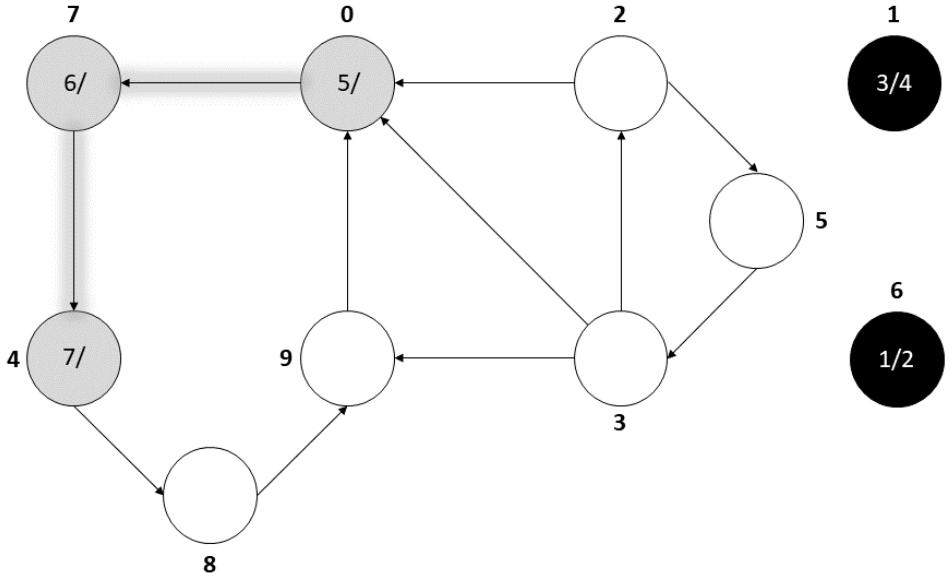
Step 6)



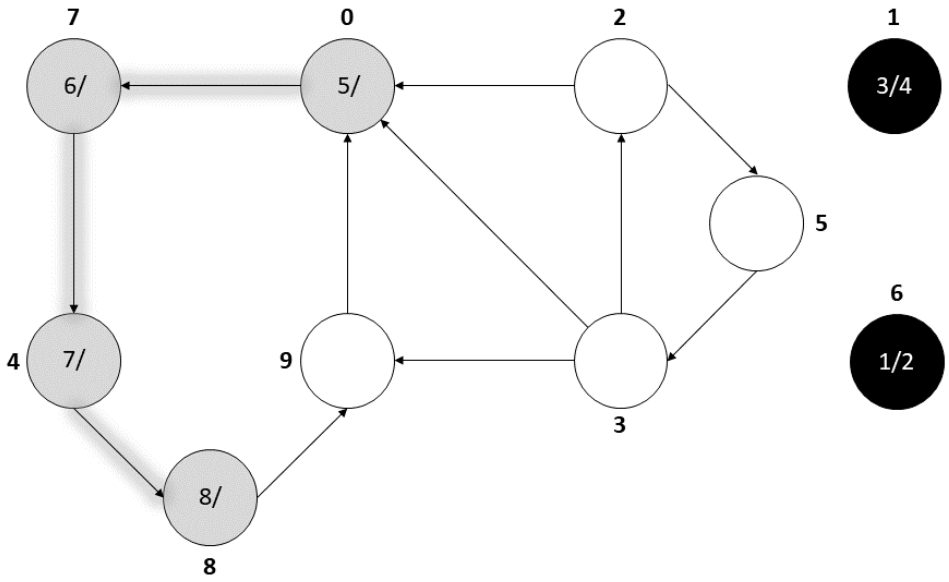
Step 7)



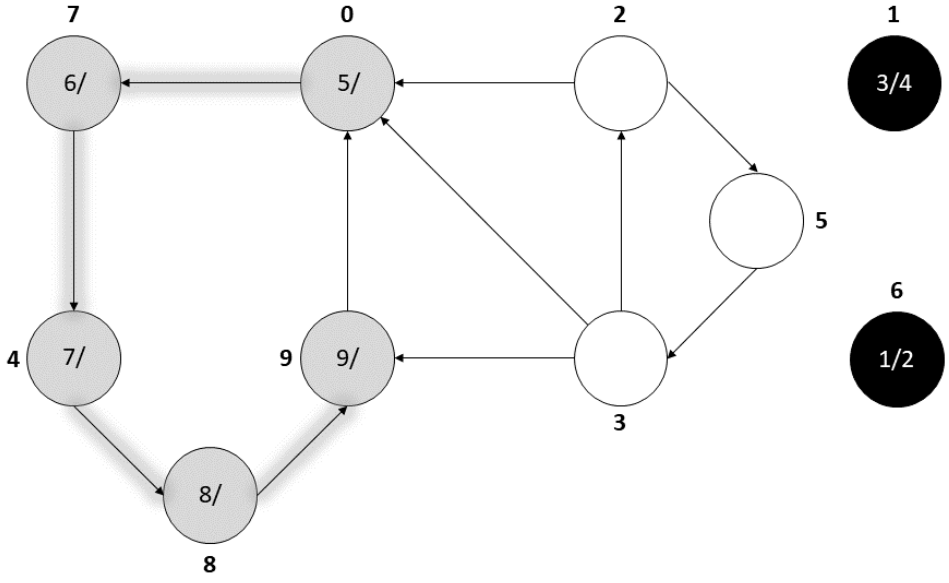
Step 8)



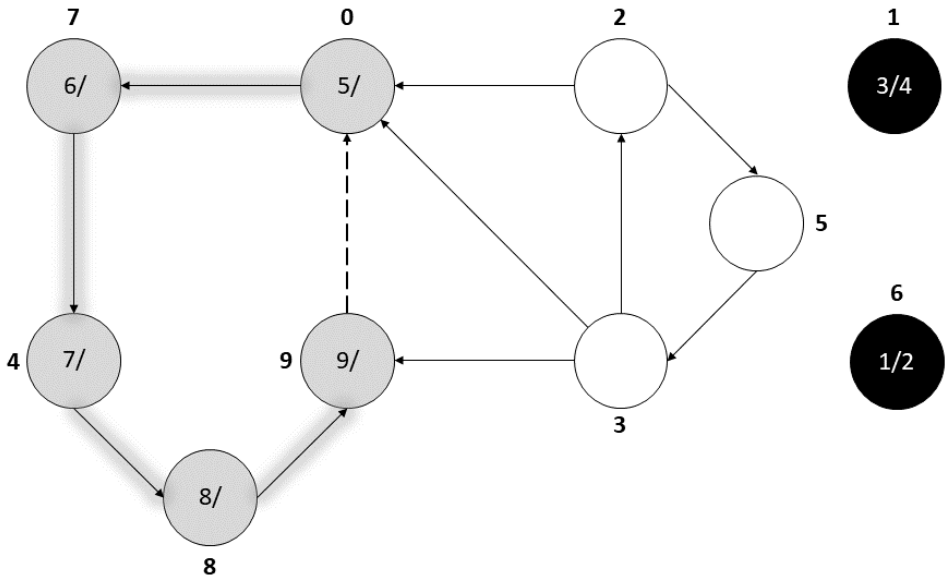
Step 9)



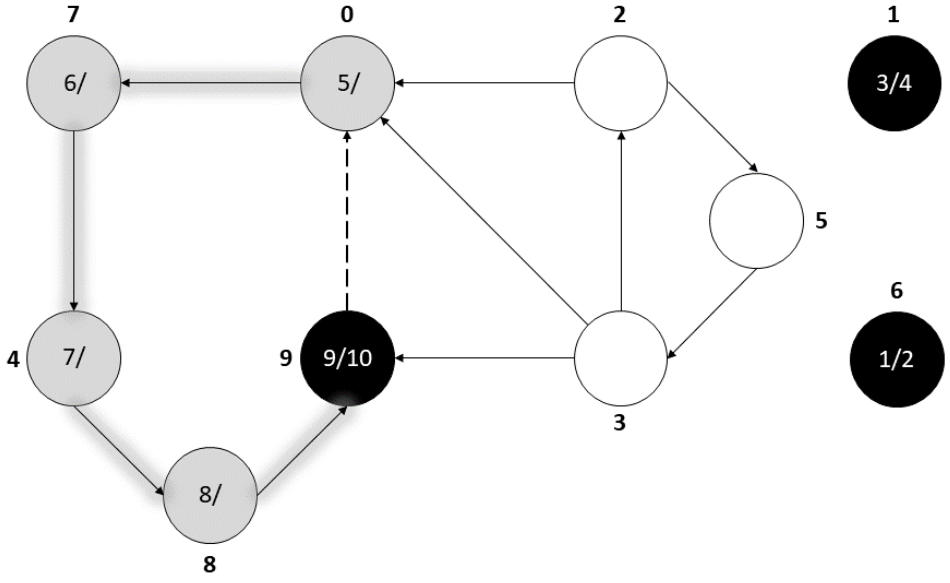
Step 10)



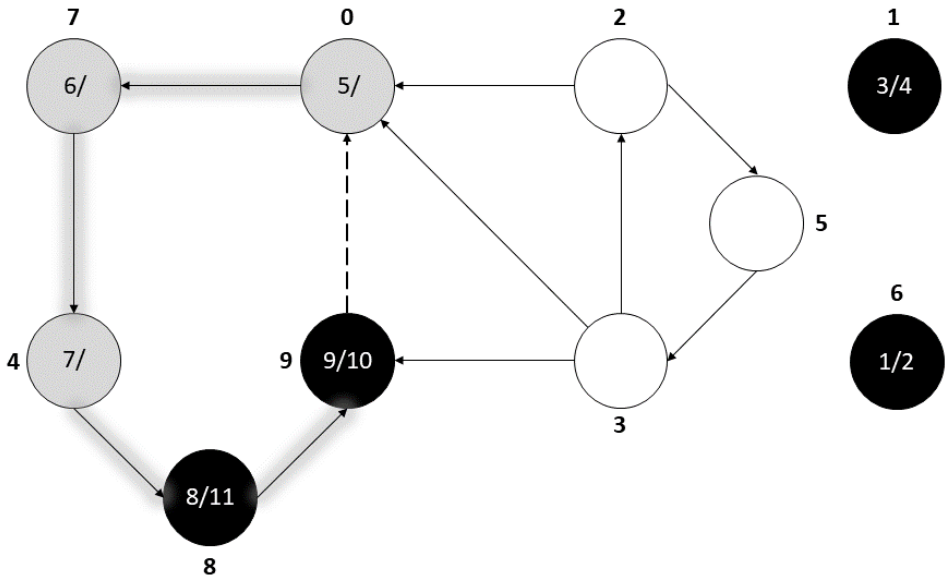
Step 11)



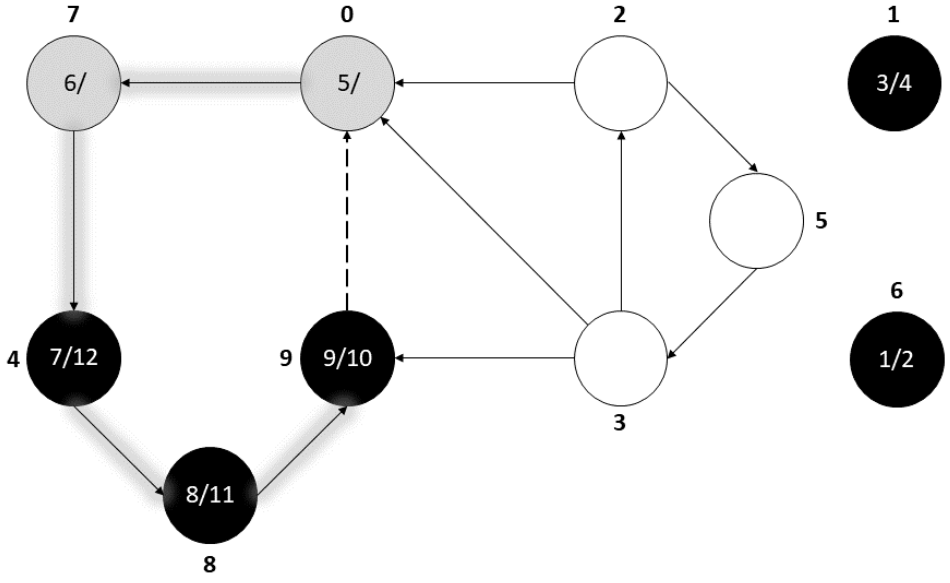
Step 12)



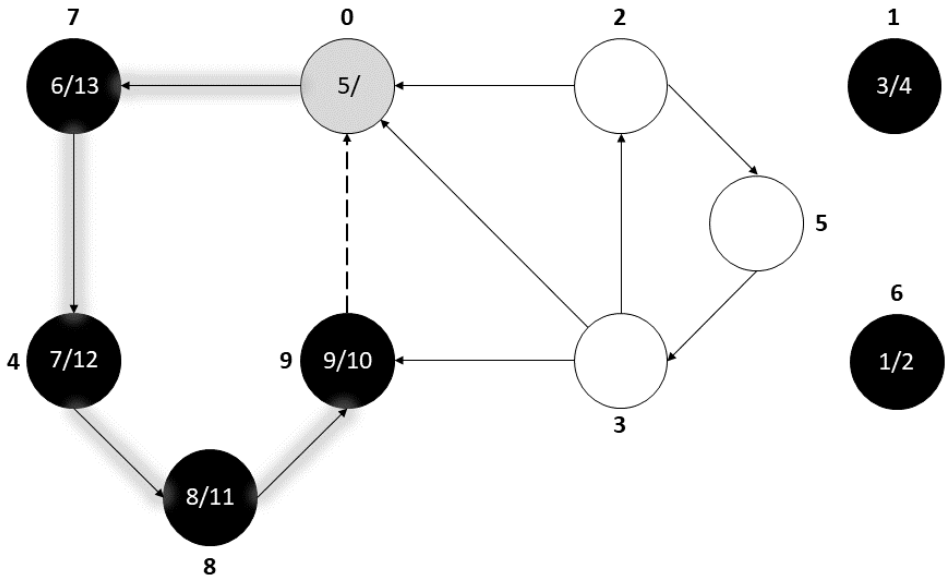
Step 13)



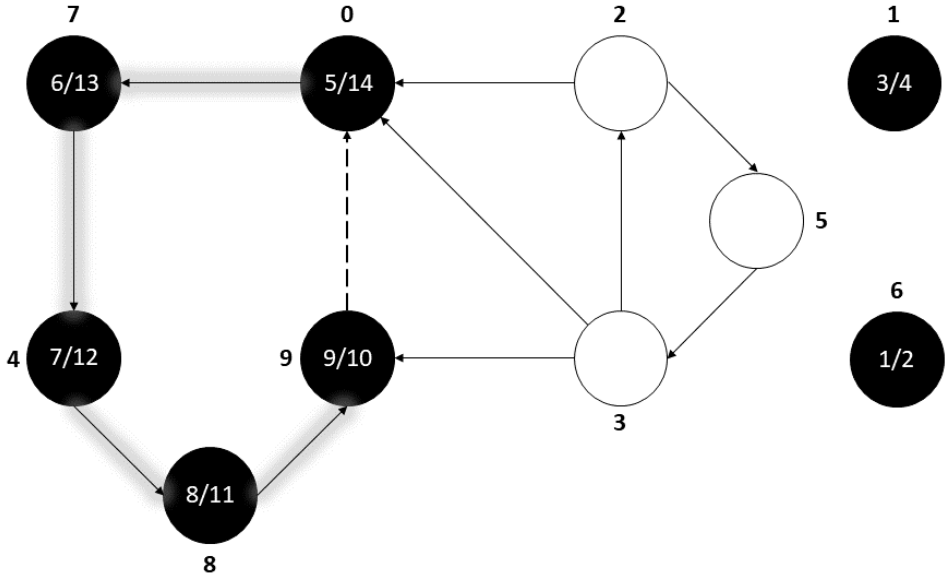
Step 14)



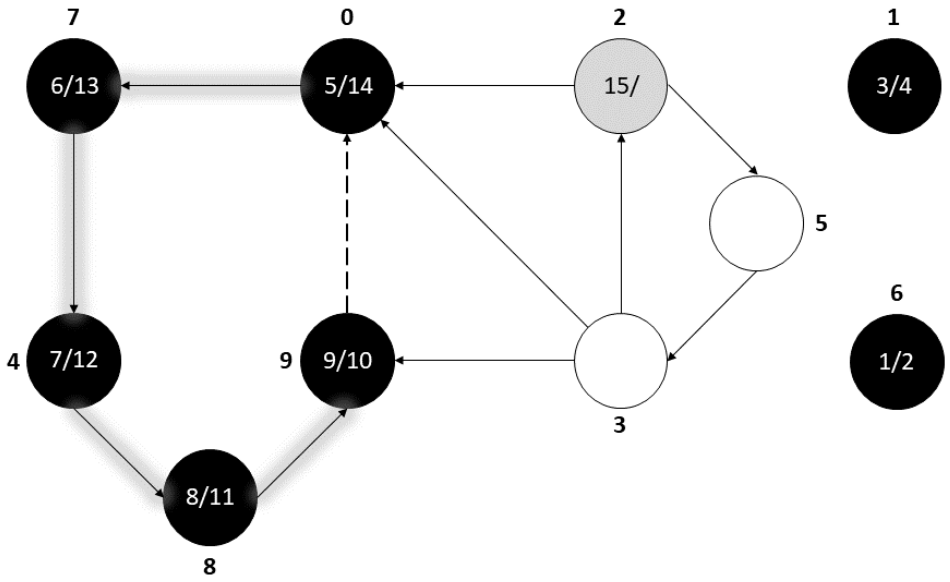
Step 15)



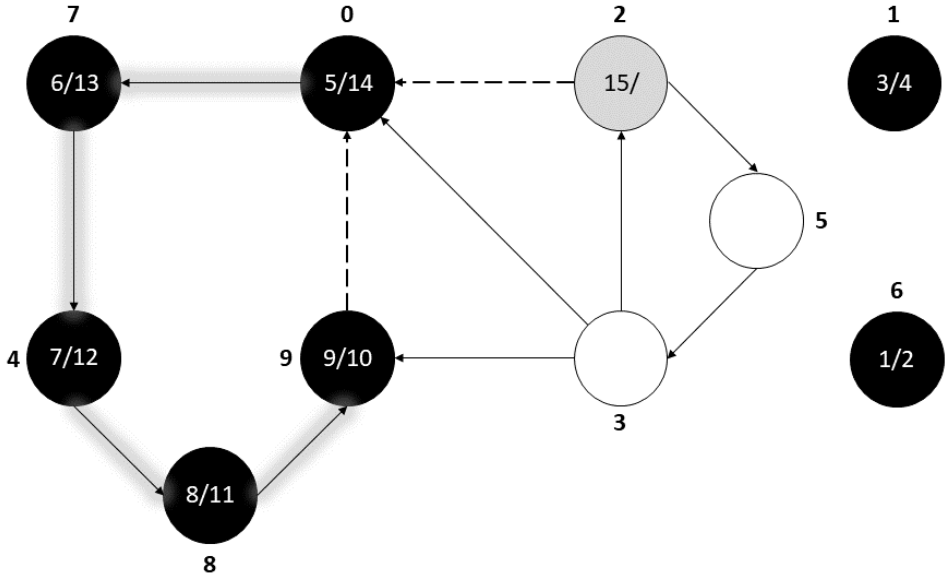
Step 16)



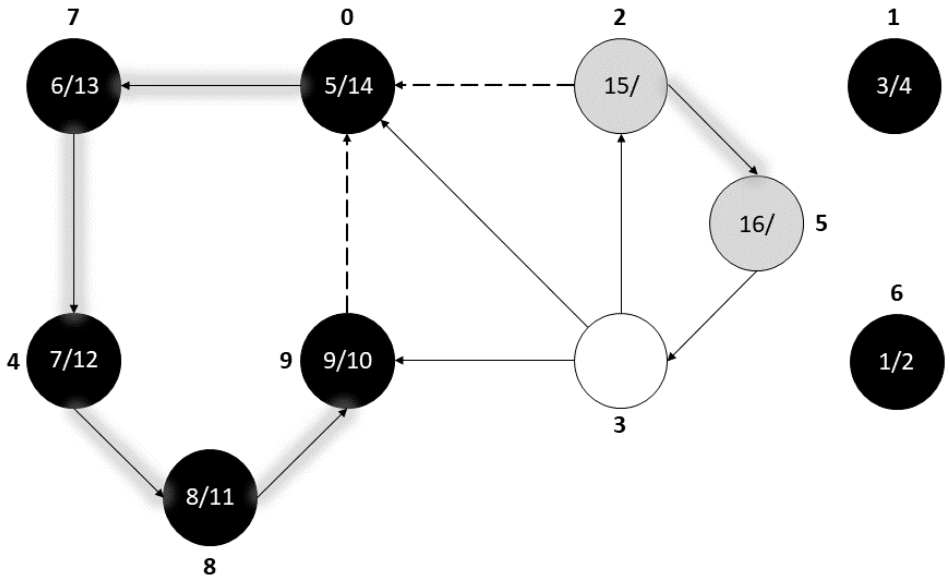
Step 17)



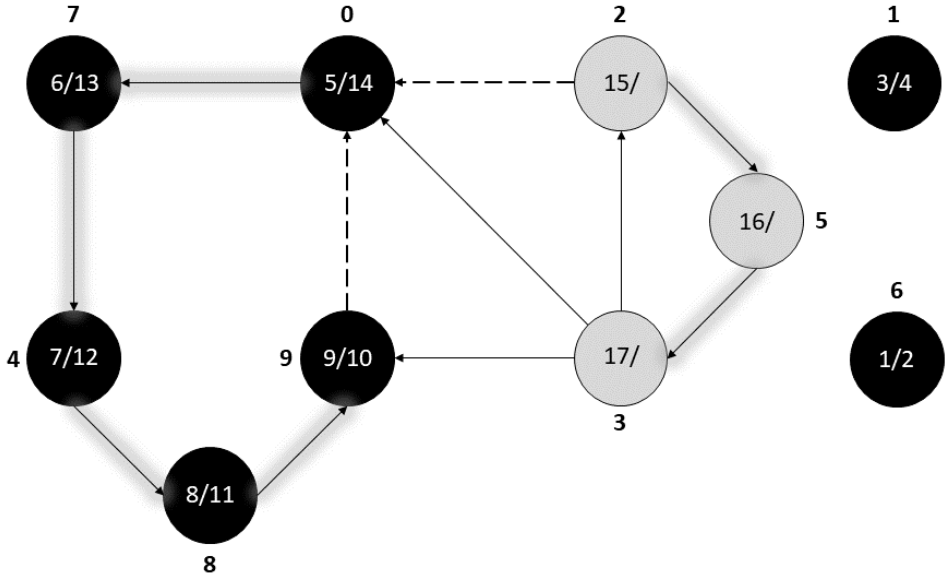
Step 18)



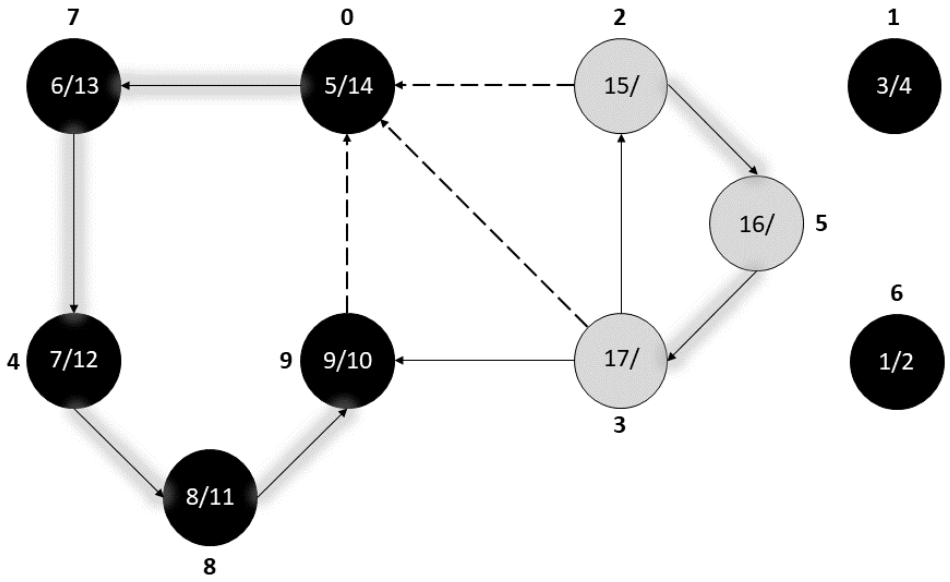
Step 19)



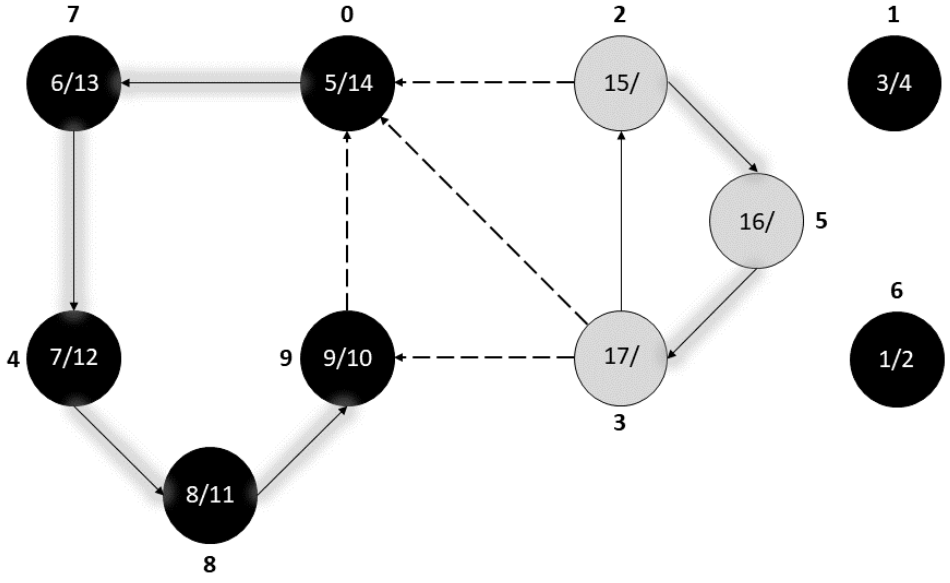
Step 20)



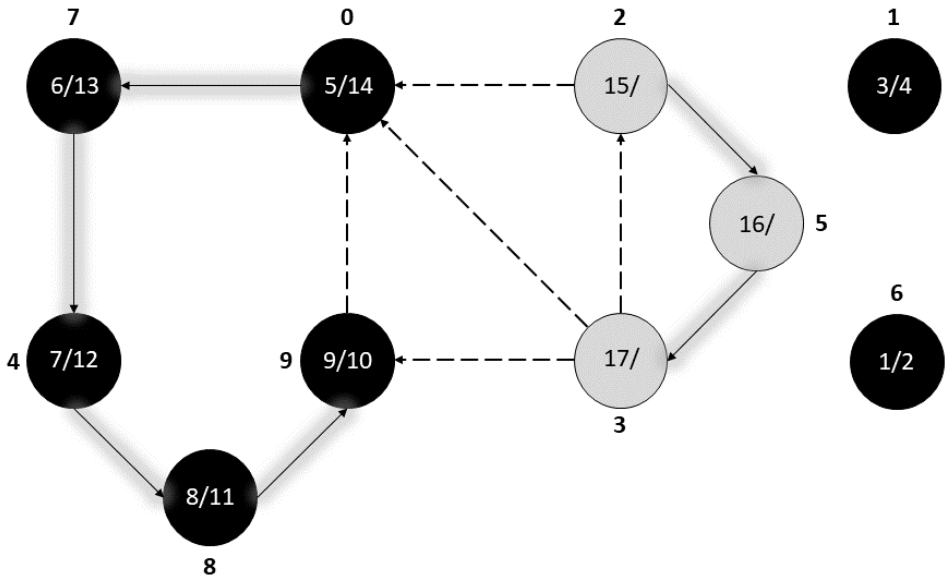
Step 21)



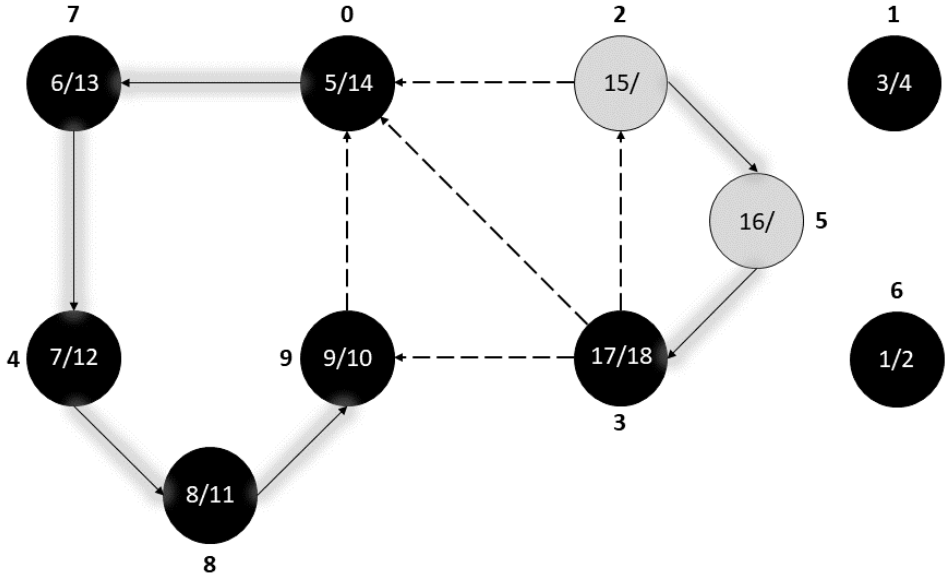
Step 22)



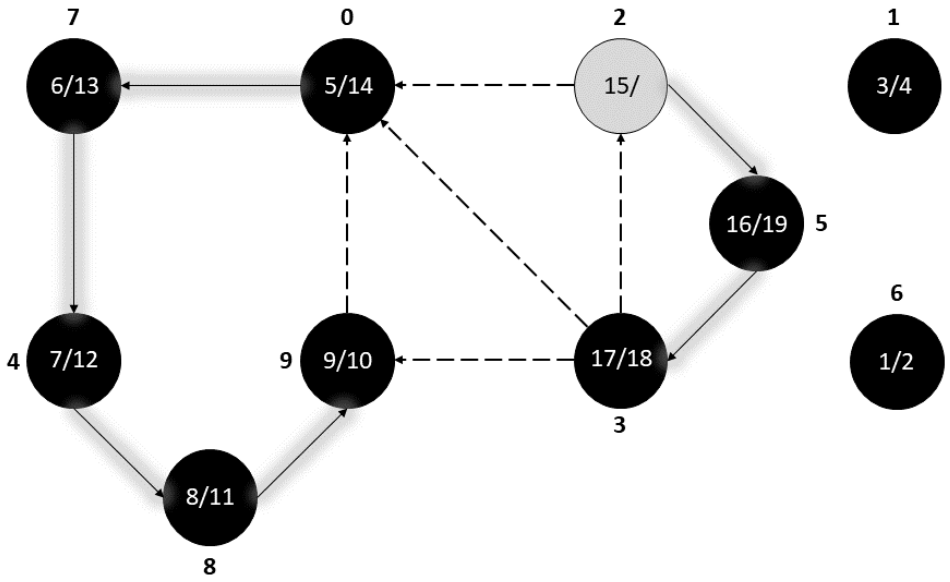
Step 23)



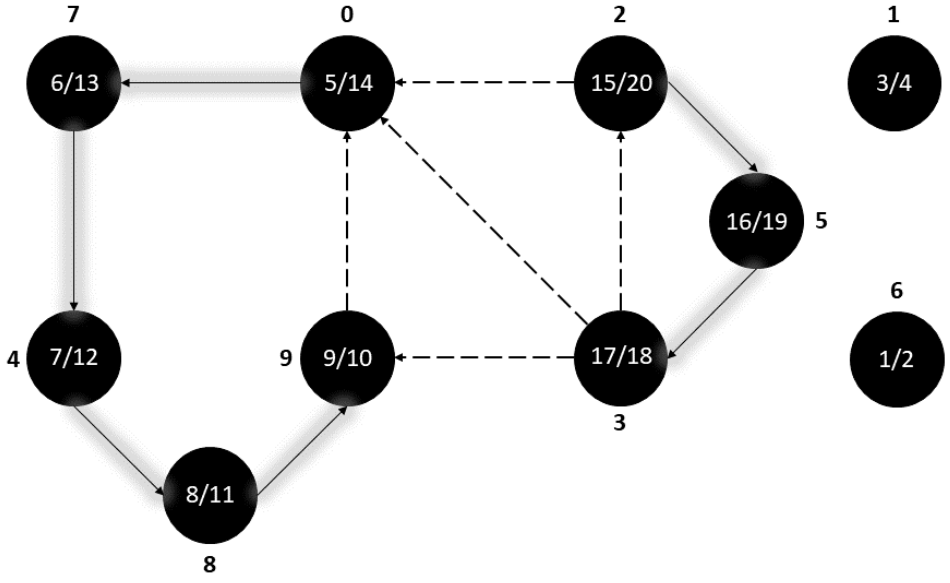
Step 24)



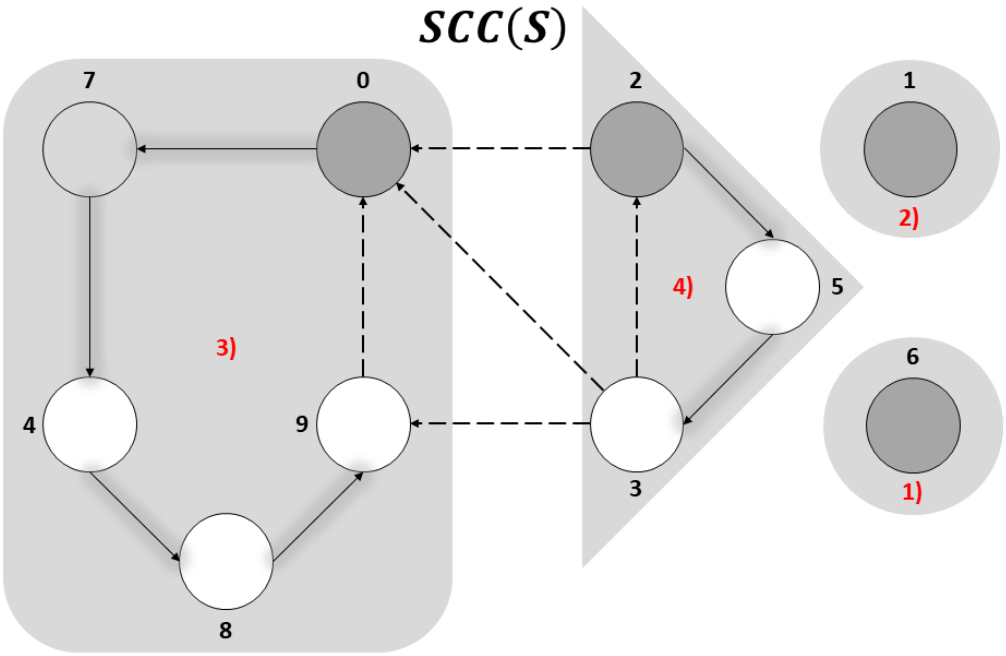
Step 25)



Step 26)



The resulting output from line 4 of the SCC algorithm using S as input is provided below. Note that the trees in the depth-first forest have been drawn in the order in which they were constructed as required. This is indicated by the labels in red 1),2),3),4). Furthermore, the heavily shaded nodes in each strongly connected component represent the roots of the depth-first trees produced by the depth-first search of S^T (line 3). In other words, vertices $6,1,2,0$ are the root nodes of the depth-first trees (produced in the order in which they were constructed) by the depth-first search of S^T.



Part B

3.

Please use the submitted $Graph.java$ and $TradeFinder.java$ files for the design and implementation of the algorithm.

4.

a)

Pseudocode summarising the main algorithm along with subroutines used in the implementation for question 3 is provided below:

Algorithm 1: $canTrade(T, t, g)$	
1:	// T is the set of traders, t is a trader ∈ T and g is the type of item trader t is willing to trade.
2:	// Trivial case – trader t naturally produces item g.
3:	**if** $t.produces == g$ **then**
4:	**return** $true$
5:	// Initialise a new graph G where the vertices $G.V = T$ and the edges $G.E$ are based on items that two
6:	// traders are mutually willing to trade.
7:	$G = initialiseGraph(T)$
8:	// Solve problem using edge relaxation methodology derived from the Bellman-Ford algorithm.
9:	// Relax each edge up to a total of $\|G.V\| - 1$ times.
10:	**for** $i = 1$ **to** $\|G.V\| - 1$ **do**
11:	**for** each edge $(u, v) \in G.E$ **do**
12:	// $relaxEdge$ early returns true iff trader t ends up being able to trade g after an edge relaxation.
13:	// Stops further unnecessary computation.
14:	**if** $relaxEdge\big(G, t, g, (u, v)\big) == true$ **then**
15:	**return** $true$
16:	**end**
17:	**end**
18:	**return** $false$

Algorithm 2: $initialiseGraph(T)$	
1:	// T is the set of traders.
2:	// Create a single vertex for each trader in traders.
3:	**for** $t \in T$ **do**
4:	// This function creates a vertex u, adds it to the list of graph vertices $G.V$ and adds the resulting $\{t: u\}$
5:	// key-value pair into a map structure.
6:	$createVertex(t)$
7:	// Create edges based on the items any pair of traders are mutually willing to trade.
8:	**for** $i = 1$ **to** $\|G.V\|$ **do**
9:	// Ensure traders aren't the same person by setting $j = i + 1$. This also prevents duplicate edges from
10:	// forming (problem is best described as an undirected graph).
11:	**for** $j = i + 1$ **to** $\|G.V\|$ **do**
12:	$u = G.V[i]$
13:	$v = G.V[j]$
14:	// Set of items both traders are mutually willing to trade.
15:	$mutualWilling = u.willing \cap v.willing$
16:	// Need at least 2 or more items in the intersection.
17:	// A valid trade must always result in the exchange of two items.
18:	**if** $size(mutualWilling) > 1$ **then**
19:	$createEdge(u, v, mutualWilling)$
20:	**return** G

Algorithm 3: $relaxEdge(G, t, g, (u, v))$

```
1:  // G is a graph, t is a trader ∈ T, g is the type of item trader t is willing to trade and (u, v) is an edge ∈ G.E
2:  // Iterate over all possible item pairs to check for a trade agreement between the two traders in (u, v)
3:  for g_i ∈ (u, v).willing do
4:      for g_j ∈ (u, v).willing do
5:          // Ignore same item trade agreements.
6:          // Then check whether traders represented by vertices u and v can trade g_i and g_j respectively so
7:          // that after the trade agreement, u and v can now trade g_j and g_i respectively (swapped).
8:          if g_i ≠ g_j and g_i ∈ u.can and g_j ∈ v.can then
9:              u.can = u.can ∪ g_j
10:             v.can = v.can ∪ g_i
11:             // Early return to stop forming additional trade agreements if the latest trade agreement
12:             // results in the original trader t being able to trade items of type g.
13:             if g ∈ w.can then
14:                 return true
15:     end
16: end
17: return false
```

b)

$x = |T| = |G.V|, \qquad y = |\{g_i\}_{i \geq 1}|$

Prior to providing an asymptotic upper bound (big-O) on the worst-case time complexity of the algorithm in terms of x and y, the following time-complexity assumptions are made based on the implemented data structures:

- A $HashSet$ is implemented for the $vertex.can$ method available to each vertex $u \in G.V$.
  - The $\in$, $\cup$ and $\cap$ set operations are assumed to have $O(1)$ worst-case time complexity.
- A $HashMap$ is implemented in the graph class to be able to find the vertex u associated to a given trader t.
  - In other words, stores $\{t: u\}$ key-value pairs for each $t \in T$ and each $u \in G.V$.
  - The $HashMap.put$ and $HashMap.get$ operations are assumed to have $O(1)$ worst-case time complexity.
- Two $ArrayList$s are additionally implemented in the graph class to store all the vertices $u \in G.V$ and edges $(u, v) \in G.E$.
  - The $ArrayList.add$ and $Arraylist.get$ operations are assumed to have $O(1)$ worst-case time complexity.

Starting in Algorithm 1, lines 3-4 are trivially $O(1)$. Line 7 shifts the compiler's focus onto Algorithm 2 where the first $\boldsymbol{for}$ loop is bounded above by $O(x)$ since based on the assumptions made earlier, $createVertex(t) \in O(1)$ and running this at most x times results in $x \times O(1) = O(x)$. Moving onto the set of nested $\boldsymbol{for}$ loops, it is first observed that the inner $\boldsymbol{for}$ loop contents are all $O(1)$ based on earlier assumptions. The outer $\boldsymbol{for}$ loop runs at most x times in the worst-case. The inner $\boldsymbol{for}$ loop reduces by 1 on each iteration of the outer $\boldsymbol{for}$ loop i.e. $x - 1$ times on the first iteration, $x - 2$ on the second, etc. This is summarised mathematically as:

$$x - 1 + x - 2 + x - 3 + \cdots + 1 + 0 = \sum_{i=0}^{x-1} i = \frac{(x-1)(x-1+1)}{2} = \frac{x(x-1)}{2}$$

Thus, in total, lines 8-19 in Algorithm 2 have an upper bound on the worst-case time complexity of $O\left(\frac{x(x-1)}{2}\right) \in O(x^2)$. Hence, $initialiseGraph(T) \in O(x^2)$.

Now focusing on lines 10-17 of Algorithm 1, it is crucial to determine exactly what setup of the problem will result in worst-case time complexity. If T is formed in a way such that every $t \in T$ is connected to every other trader $\in T$ i.e. each t is willing to produce every possible good ($size(t.willing) = y, t \in T$), then the inner $\boldsymbol{for}$ loop will run at most $\frac{x(x-1)}{2}$ times in the worst-case (upper bound on the maximum number of edges in an undirected graph).

Considering the $relaxEdge\big(G, t, g, (u, v)\big)$ function appearing on line 14, this setup of T also yields the worst-case runtime for the nested $\boldsymbol{for}$ loops on lines 3-16 in Algorithm 3 because each $\boldsymbol{for}$ loop runs at most y times. Lines 8-14 are trivially $O(1)$ based on assumptions made earlier. Hence, in total and in the worst-case, $relaxEdge\big(G, t, g, (u, v)\big) \in O(y \times y) = O(y^2)$.

Returning to the outer $\boldsymbol{for}$ loop on line 10 in Algorithm 1, in the worst-case, only one iteration passes through before the $\boldsymbol{return}$ $true$ statement on line 15 is triggered. This is because after the inner $\boldsymbol{for}$ loop finishes on the first iteration of the outer $\boldsymbol{for}$ loop, for every vertex $u \in G.V$, its can trade set $u.can$ will have been updated to its maximum potential and the resulting $\boldsymbol{return}$ $true$ statement on line 15 must be triggered conducting an early stop. The final runtime of lines 10-17 of Algorithm 1 is $O(x^2) \times O(y^2) = O(x^2 y^2)$. Hence, based on all the assumptions and arguments made above:

$$canTrade(T, t, g) \in O(x^2 y^2)$$

c)

$x = |T| = |G.V|, \qquad y = |\{g_i\}_{i \geq 1}|$

Prior to providing an asymptotic upper bound (big-O) on the worst-case space complexity of the algorithm in terms of x and y, the following space-complexity assumptions are made based on the implemented data structures:

- A $HashSet$ is implemented for the $vertex.can$ method available to each vertex $u \in G.V$.
  - Assume space usage scales with more vertices and more items.
- A $HashMap$ is implemented in the graph class to be able to find the vertex u associated to a given trader t.
  - In other words, stores $\{t: u\}$ key-value pairs for each $t \in T$ and each $u \in G.V$.
  - Assume space usage scales linearly **only** with more vertices
    - Each trader t is contained within u, hence, the memory is shared.
- Two $ArrayList$s are additionally implemented in the graph class to store all the vertices $u \in G.V$ and edges $(u, v) \in G.E$.
  - Assume space usage scales linearly with more vertices and edges.

First it is observed that obtaining an upper bound on the worst-case space complexity of $canTrade(T, t, g)$ can be simplified to obtaining an upper bound on the worst-case space complexity of $initialiseGraph(T)$. This is because the rest of Algorithm 1 and Algorithm 3 do not create variables/call functions that use space/memory scalable with either x or y (they simply call functions and methods associated with the constructed graph G). Thus, an upper bound on the worst-case space complexity of $canTrade(T, t, g)$ is provided by $initialiseGraph(T)$.

Within the graph G generated by $initialiseGraph(T)$, there are several subcomponents whose worst-case space complexity must be considered. Starting with items, this is trivial because each item stores worst-case $O(1)$ space. Next, considering traders and using the same worst-case scenario as from **b)** above ($size(t.willing) = y, t \in T$), each $t \in T$ stores at most $O(y)$ space due to storing the willing to trade items. Each vertex $u \in G.V$ stores its representative trader $t \in T$ and its current can trade set $u.can$. The can trade set can store a maximum of y elements (same as $t.willing$ in the worst-case) so its worst-case space usage is also bounded above by $O(y)$. In total, each vertex $u \in O(y) + O(y) \in O(y)$. Similar, each edge $(u, v) \in G.E$ consumes space for each vertex as well as the $mutualWilling$ set of items, the latter of which is bounded above by $O(y)$ (same as $t.willing$ in the worst-case). Hence, in total, each edge (u, v) also consumes $O(y) + O(y) + O(y) \in O(y)$ space.

Based on the arguments above, the vertices $ArrayList$ has worst-case space of $|G.V| \times O(y) = O(xy)$ whilst the edges $ArrayList$ has worst-case space of $\frac{|G.V|(|G.V|-1)}{2} \times O(y) \in O(x^2 y)$ (upper bound on the maximum number of edges in an undirected graph). The implemented $HashSet$ has worst-case space of $|G.V| \times O(y) = O(xy)$ whilst the implemented $HashMap$ has worst-case space complexity of $|T| \times O(y) = O(xy)$. The latter does not require multiplying by another factor of $O(y)$ because each trader t is contained in its representative vertex u – in other words, the memory is shared.

Finally, the upper bound on the worst-case space complexity of $initialiseGraph(T)$ is $O(x^2 y)$ due to the edges $ArrayList$ needing to store $\frac{x(x-1)}{2}$ edges in the worst-case scenario (which was explained above in **b)**). Hence, based on all the assumptions and arguments made above:

$$canTrade(T, t, g) \in O(x^2 y)$$