5/11/2018

Christopher Hsu Chang

COMP3001

DESIGN AND ANALYSIS OF ALGORITHMS ASSIGNMENT

Christopher Hsu Chang ID: 18821354

Table of Contents

[Question 1a: 3](#_Toc514030641)

[Question 1b: 4](#_Toc514030642)

[Question 1c: 5](#_Toc514030643)

[Question 2a Adjacency List: 6](#_Toc514030644)

[Question 2b Greedy Algorithm: 7](#_Toc514030645)

[Question 2c Time Complexity: 8](#_Toc514030646)

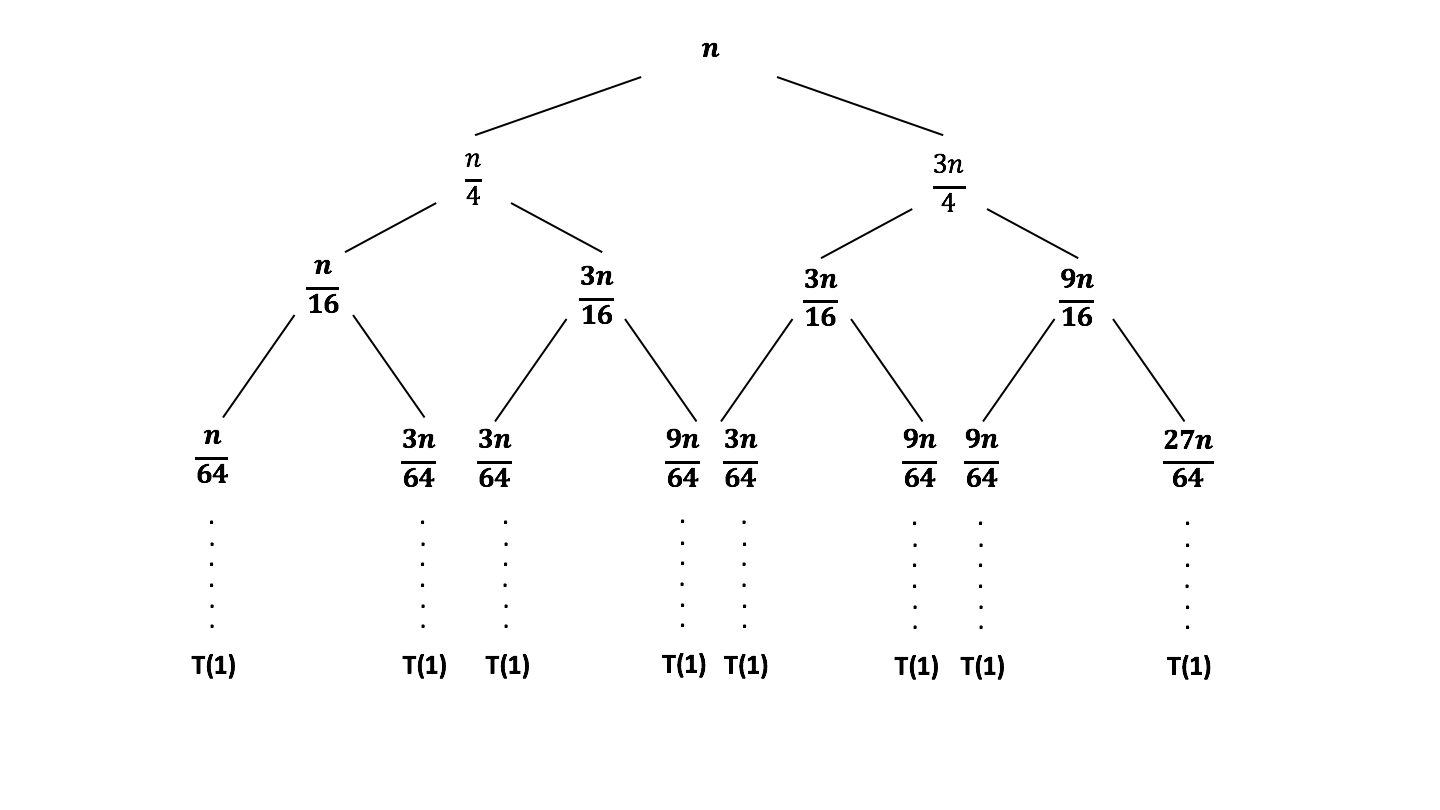
[Question 2d Minimum sized vertex cover for given graph: 9](#_Toc514030647)

[Question 2c Working and Non-working examples: 13](#_Toc514030648)

[Question 3b: 36](#_Toc514030649)

# Question 1a:

**Show the recurrence tree for to guess its asymptotic upper bound complexity. Hint.** The textbook shows an example similar to this problem.



∴ The Asymptotic upper bound complexity will be , because although the recurrence function has , it is still represented as a binary tree meaning there will be splits done n times where n is the height of the tree. The height of the tree is log4/3n and each level will add up to n.

# Question 1b:

**Use induction to verify your guess in part a).**

Prove is ,

Assume holds when

// Drop Denominator

// Rearrange Signs

// Factorise

Now Prove for Base Case where n = 1:

Where is not true!!

But, assume n0 is greater than 1, so try for base case where n = 2:

0 + 1 + 2 = 3

Where Which is not true!

So try n = 3:

Where Which is True ☺

∴ holds true for values of and for the upper bound time complexity.

# Question 1c:

**Use induction to verify if your guess in part a) also applies for its asymptotic lower bound complexity.**

Prove ,

Assume holds when

// Drop 3cnlg3n because the above

// line is still larger than this one

// Now find for c

c2lgn

c2lgn

// C must be less than or equal to a half

Now prove for the base case where n = 3, We don’t need start at 1 because we have already

proven that n = 3 for upper bound so of course it will work for n = 1 and n = 2.

Where Which is true but doesn’t say much

n = 2:

0 + 1 + 2 = 3

Where Which is true

n = 3:

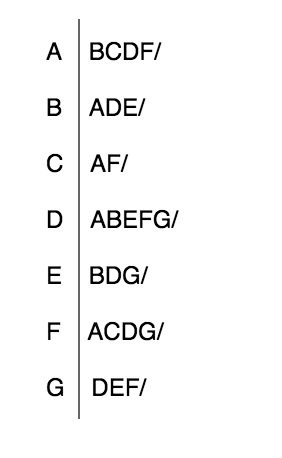
0 + 3 + 3 = 6

Where Which is true!

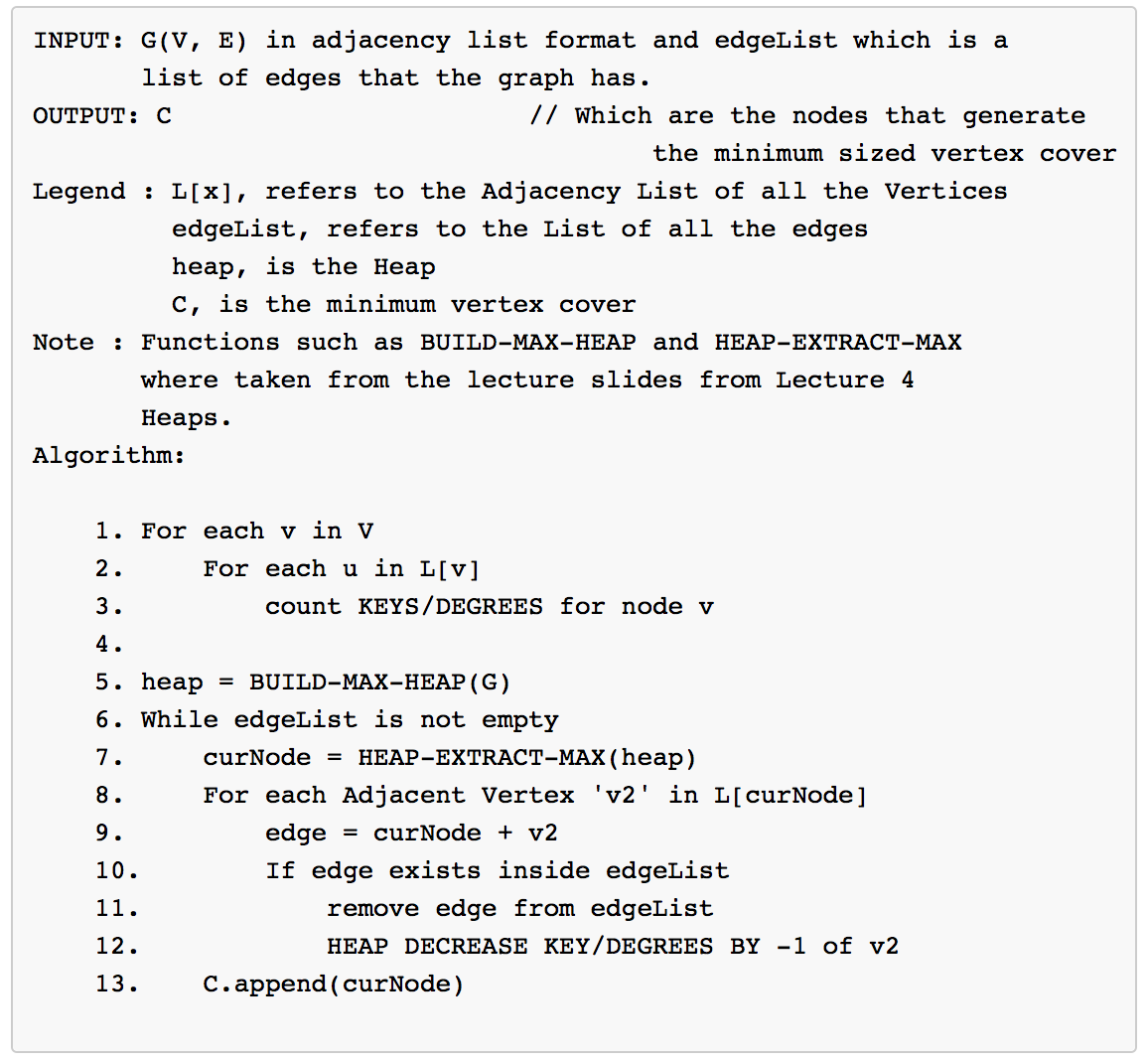
∴ holds true for values of and for the lower bound time complexity.

# Question 2a Adjacency List:

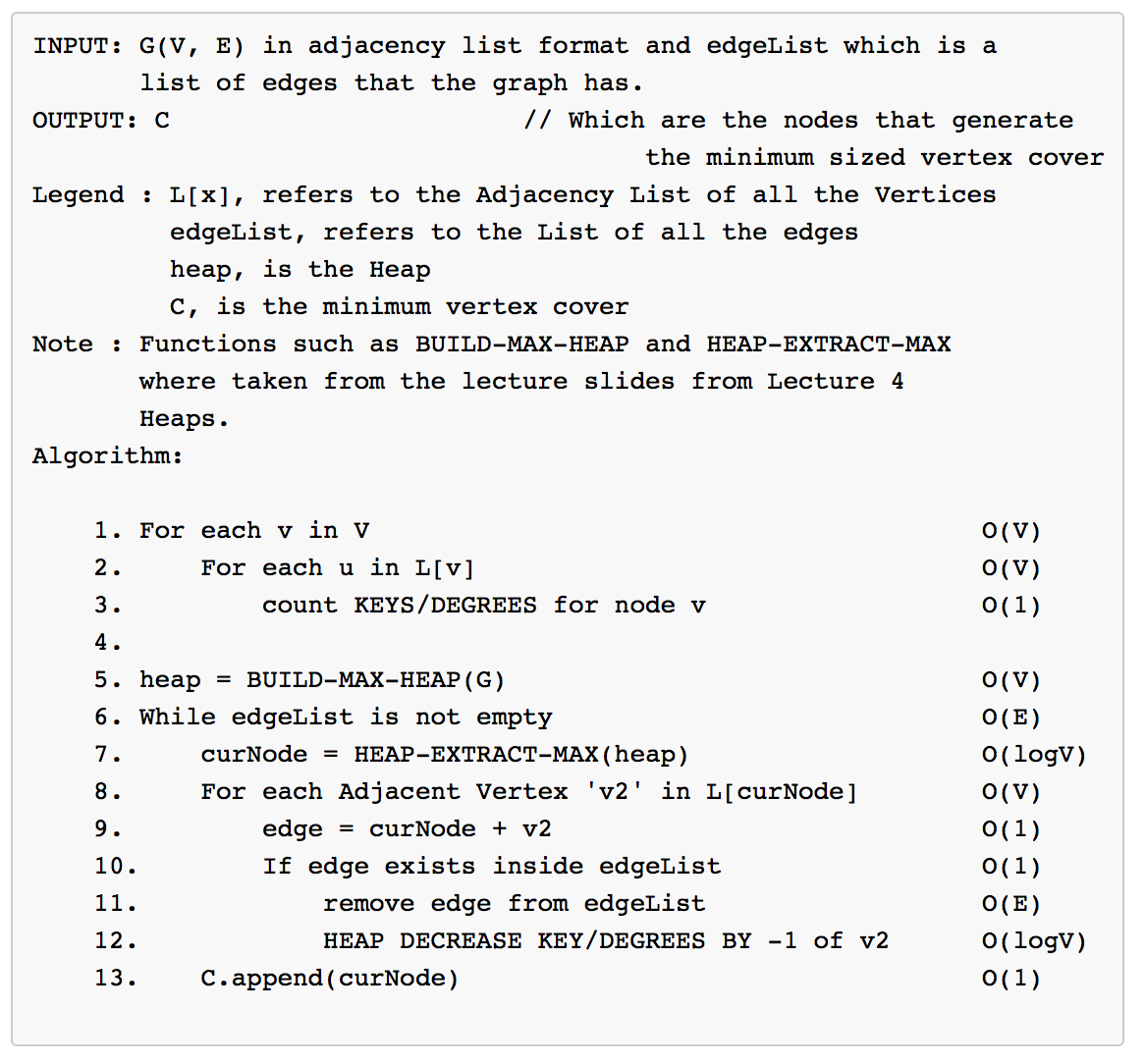
**Represent the Graph as an Adjacency List**



# Question 2b Greedy Algorithm:



# Question 2c Time Complexity:



**Time Complexity = 2EVlog(V)**

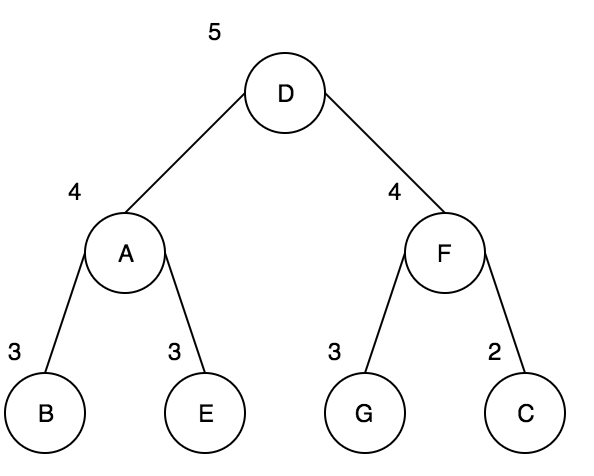
The time complexity is 2VElog(V) because when you build the heap you are inserting each vertex into the heap V times and then maintaining the heap property by calling max-heapify (From the lectures), which is log(V) but since it is bottlenecked by the for loop of build-max-heap it is still only O(V). What makes this algorithm O(E) is that we are constantly checking if the edgeList is empty because once that list is empty we are certain that we have found the Minimum sized vertex cover. Within the while loop we are also running through the adjacency list to find the correct nodes (seen in line 8) to check if an edge exists in the edgeList and at most that is going to run 2E times because it appears twice in the adjacency list. Log(V) is derived from the HEAP-EXTRACT-MAX and the HEAP-DECREASE KEY functions. In order to extract the max element from the heap the operation is O(1) but it takes O(logV) to maintain the heap-property because you have to trickle-down the heap.

# Question 2d Minimum sized vertex cover for given graph:

L[x] = // Same as the list from question 2a)

edgeList = {AB, AC, AD, AF, BD, BE, CF, DE, DF, DG, EG, FG}

C = {}

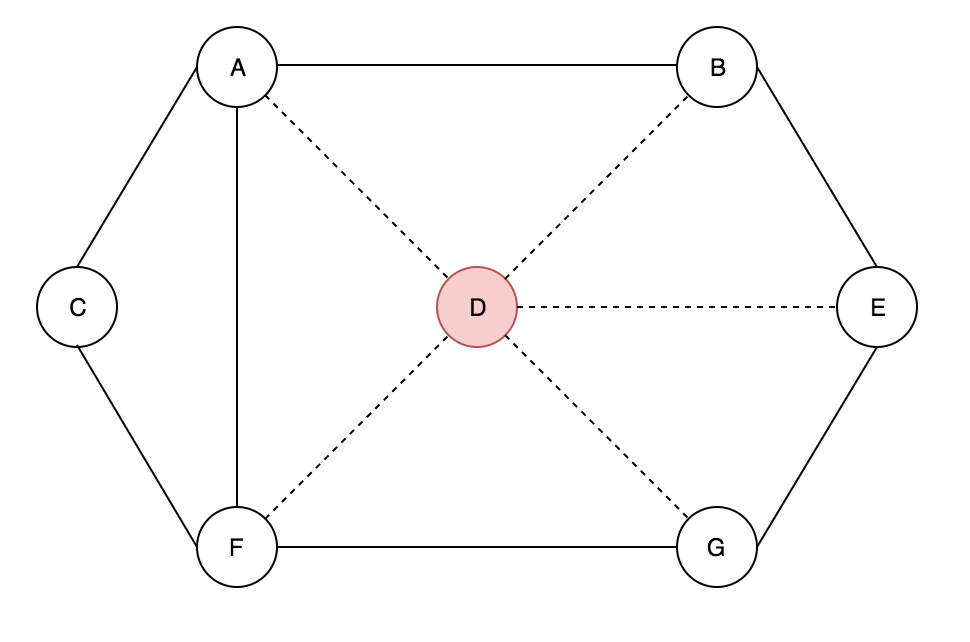


Heap Representation where this is a Max-Heap and the numbers represent the degrees of each vertex. (Line 5)

Line 7: Node D gets extracted from the heap and Max-Heapify is called to maintain the heap property.

Line 8: Going through each Node that is on D’s adjacency list. To find any matching edges.Screen%20Shot%202018-05-13%20at%2010.13.17%20pm.png

Lines 9-12: Whie it is looping through D’s adjacency list, it is checking if an edge exists inside the edgeList. If the edge exists it removes it from the edgeList and decreases the keys of all of the adjacent vertices by 1. Because of this the graph will be missing all the links that was connected to the original node.



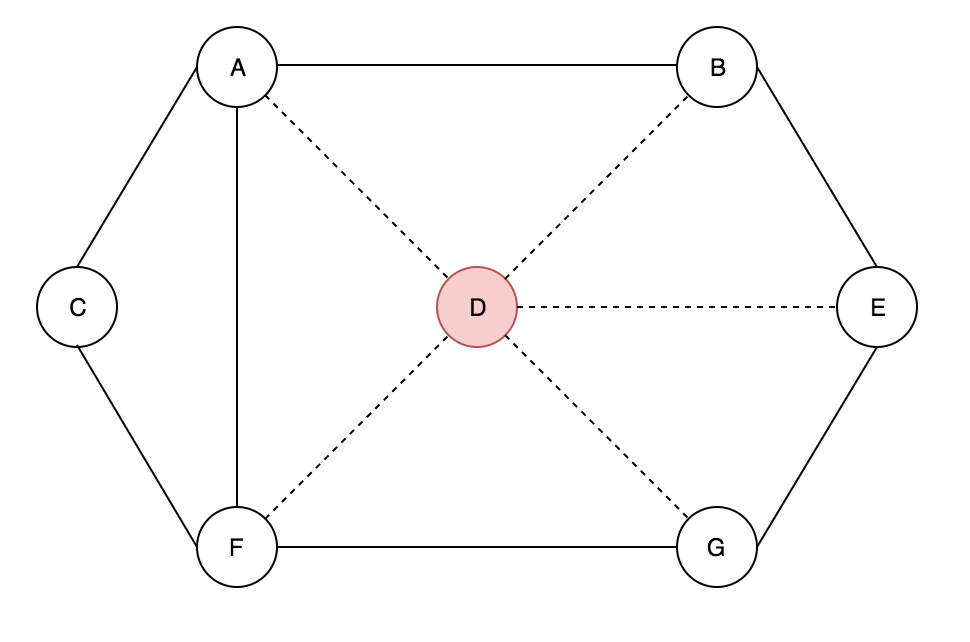
Line 13: The current node that was just extracted gets added onto the vertex cover.

C = {D}

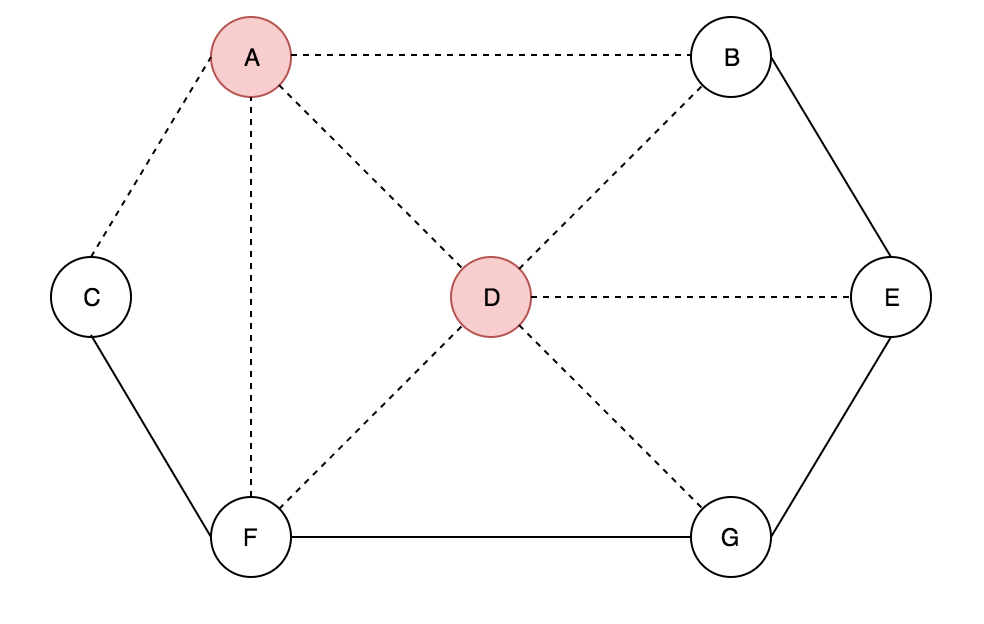
The process show above, is repeated again and again until all of the edges have been covered and the vertex cover will be shown below in the diagram highlighted in red. Note, for cases where the degree is the same number, the node which is in alphabetical order will be chosen.

**FINAL REPRESENTATION:**

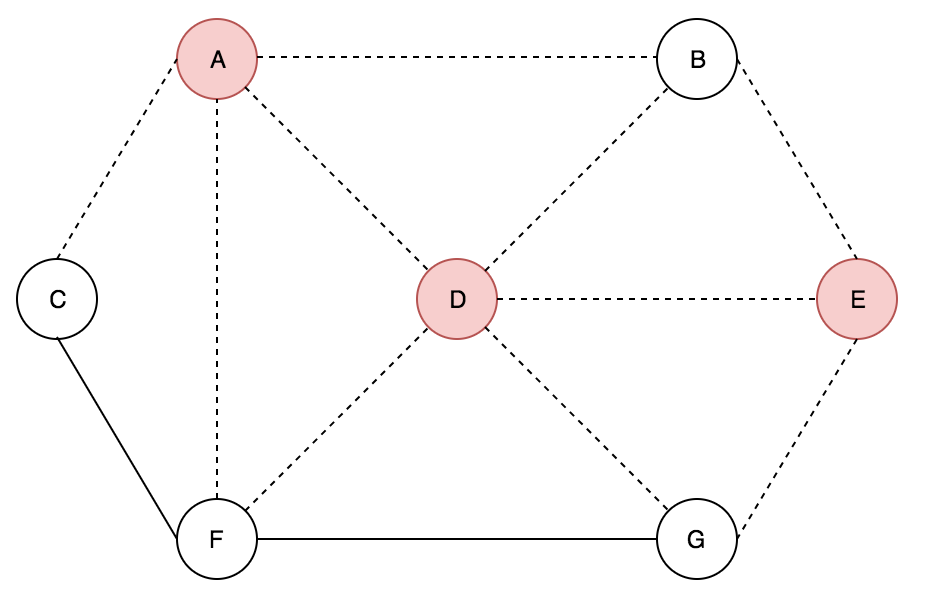
**Vertex Cover = {D}**



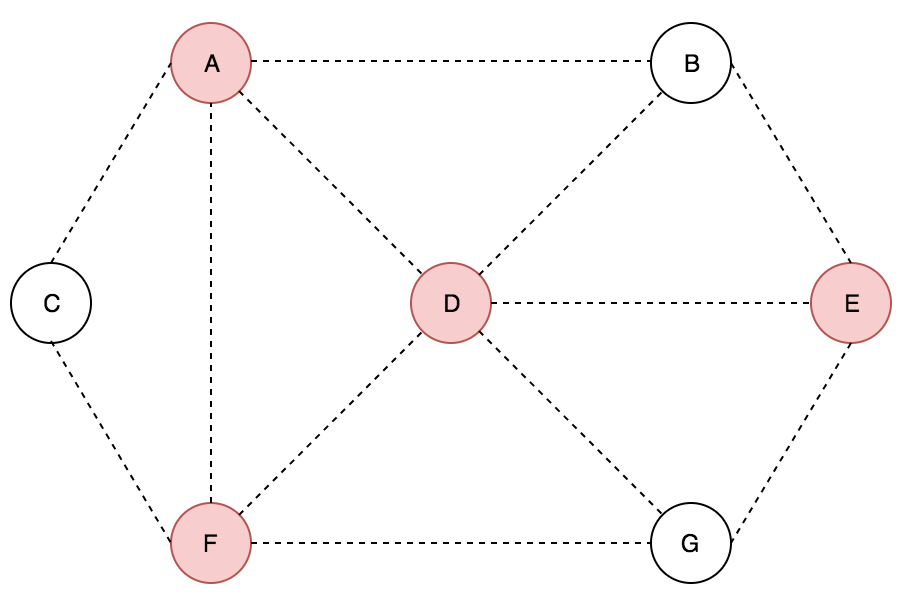
**Vertex Cover = {D, A}**

****

**Vertex Cover = {D, A, E}**

****

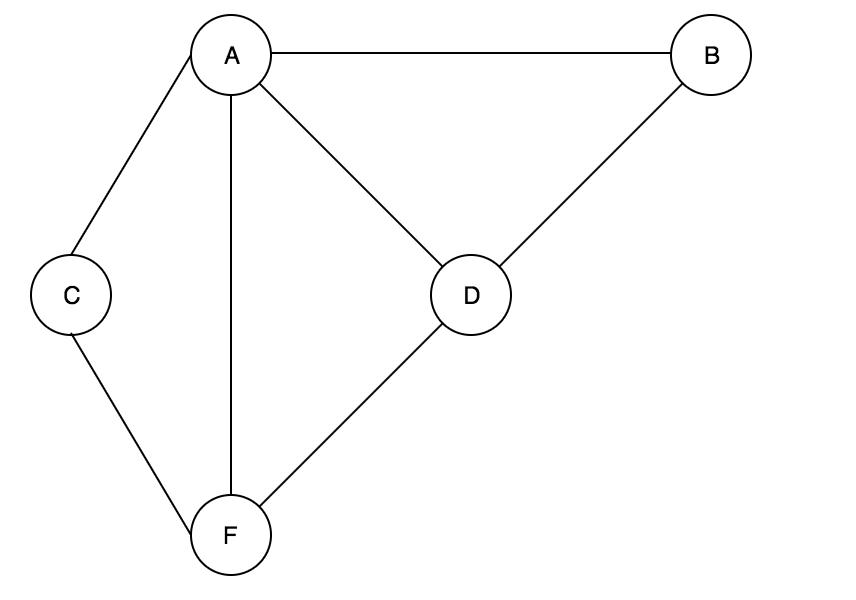
**Vertex Cover = {D, A, E, F}**

****

# Question 2c Working and Non-working examples:

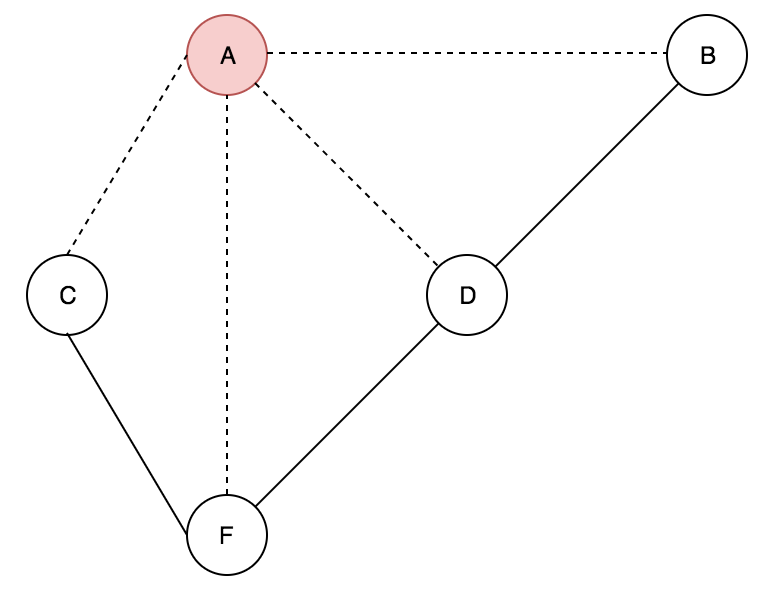
**Working:**

Input Graph:

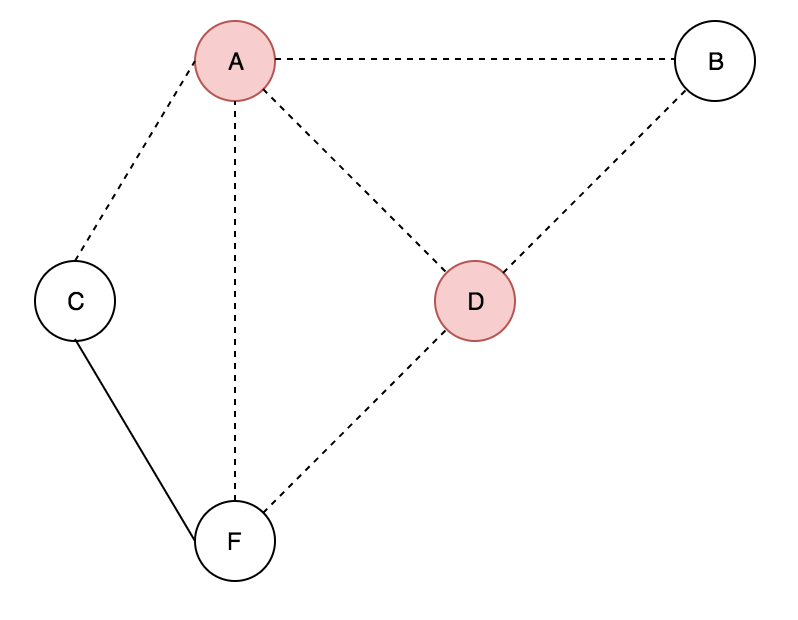


**Solution:**

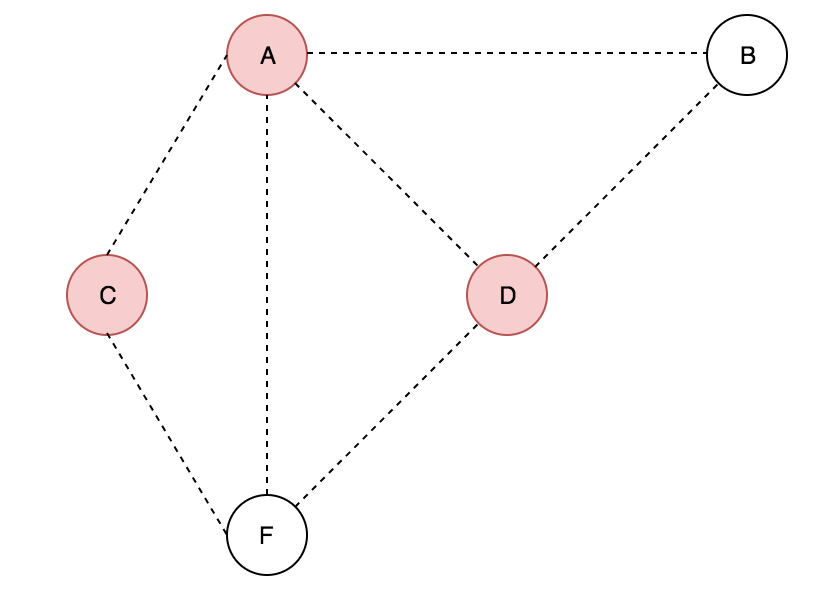
**Vertex Cover = {A}**

****

**Vertex Cover = {A, D}**

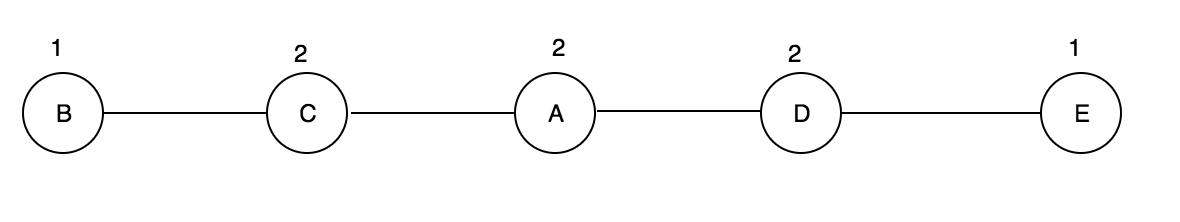
****

**Vertex Cover = {A, D, C}**

****

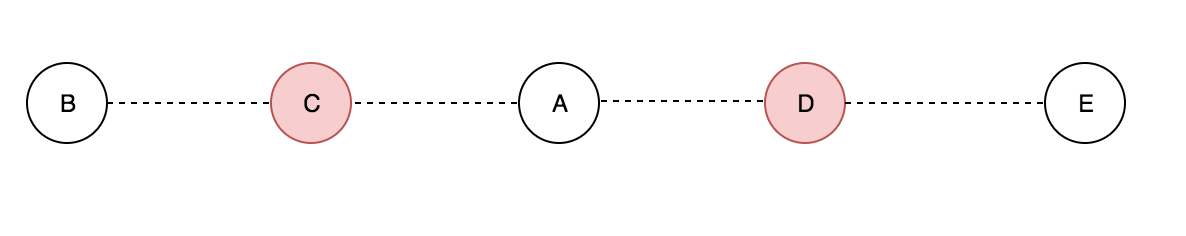
**Non - Working:**

For the following graph, the algorithm will not work.

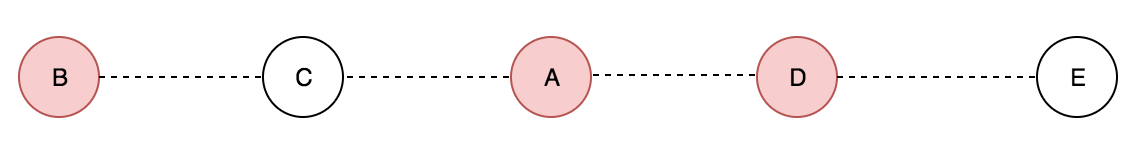


You can easily tell that the minimum vertex cover would be {C, D} that will cover all the edges but according to the algorithm, if the degrees of the maximum node are all the same it will select A first then B then D resulting in 3 nodes being the minimum vertex cover when it can be done in two.

**What the vertex cover should be:**

****

**What the algorithm produces:**

****

Question 3: Yen’s Algorithm Question

**Note: For This Question, I used Yen’s Algorithm that was abstracted from Wikipedia. The link is given here,** [**https://en.wikipedia.org/wiki/Yen%27s\_algorithm**](https://en.wikipedia.org/wiki/Yen%27s_algorithm)

**, The pseudocode will be given below. Assuming Index starts at zero**

function YenKSP(Graph, source, sink, K):  
***1.* // Determine the shortest path from the source to the sink.**  
***2.*** A[0] = Dijkstra(Graph, source, sink);  
**3.** **// Initialize the set to store the potential kth shortest path.**  
***4.*** B = [];  
***5.  
6.*** for k from 1 to K:

***7.*** **// The spur node ranges from the first node to the next to last node in**

***8.* the previous k-shortest path**  
***9.*** for i from 0 to size(A[k − 1]) − 1:  
***10.*** **// Spur node is retrieved from the previous k-shortest path, k − 1.**  
***11.*** spurNode = A[k-1].node(i);  
***12.*** **// The sequence of nodes from the source to the spur node of the**

***13.* // previous k-shortest path**  
***14.*** rootPath = A[k-1].nodes(0, i);  
***15.***

***16.***

***17.***  **// Goes through each path and removes the links that are part of the**

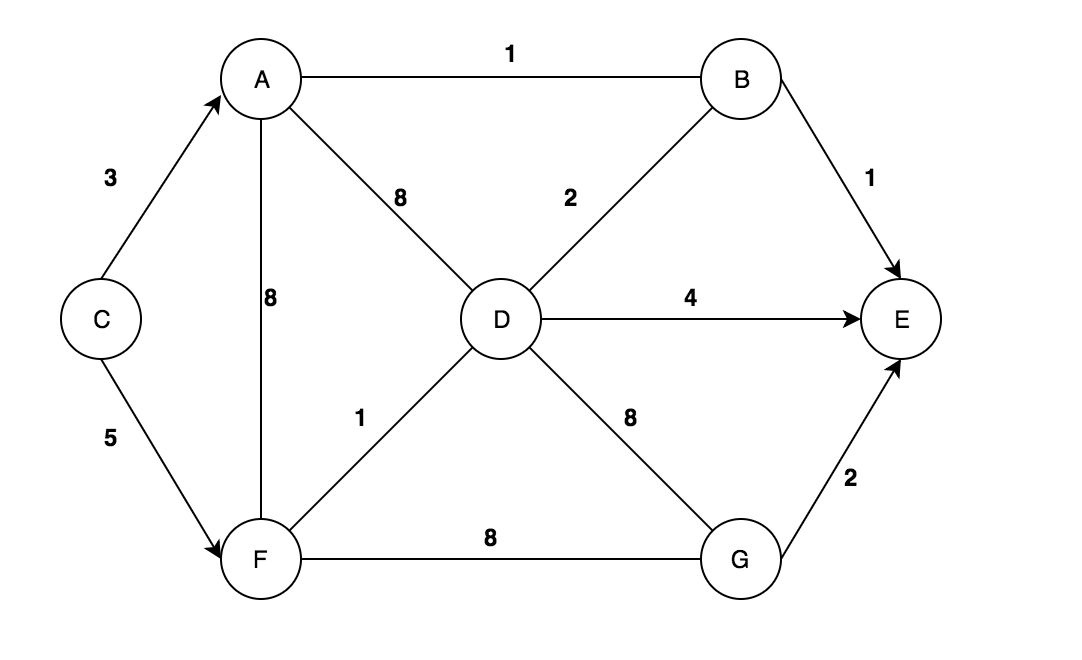
***18.* // previous shortest paths which share the same root path**  
***19.*** for each path p in A:  
***20.*** if rootPath == p.nodes(0, i):  
***21.*** remove p.edge(i,i + 1) from Graph;  
***22.***  
***23.*** for each node rootPathNode in rootPath except spurNode:  
***24.*** remove rootPathNode from Graph;  
***25.***  
***27.*** **// Calculate the spur path from the spur node to the sink.**  
***28.*** spurPath = Dijkstra(Graph, spurNode, sink);  
***29.***  
***30.*** **// Entire path is made up of the root path and spur path**.  
***31.*** totalPath = rootPath + spurPath;  
***32.*** **// Add the potential k-shortest path to the heap.  
*33.*** B.append(totalPath);  
***34.***  
***35.*** **// Add back the edges and nodes that were removed from the graph.**  
***36.*** restore edges to Graph;  
***37.*** restore nodes in rootPath to Graph;  
***38.***

***39.*** **// This handles the case where there are no spurs paths or an left  
*40.*** if B is empty:  
***41.*** break;

***42.******43*.** // Sort the potential k-shortest paths by cost.  
***44.*** B.sort();  
***45.*** // Add the lowest cost path becomes the k-shortest path.  
***46.*** A[k] = B[0];  
***47.*** B.pop();  
***48.***  
***49.*** return A;

Student ID: 18821354

The Graph:



**Results for four shortest paths:**

* 1. C -> A -> B -> E Cost = 5
* 2. C -> F -> D -> B -> E Cost = 9
* 3. C-> A -> B -> D -> E Cost = 10
* 4. C -> F -> D -> E Cost = 10

**Yen’s Algorithm after each iteration:**

How Dijkstra finds the shortest path for first iteration.

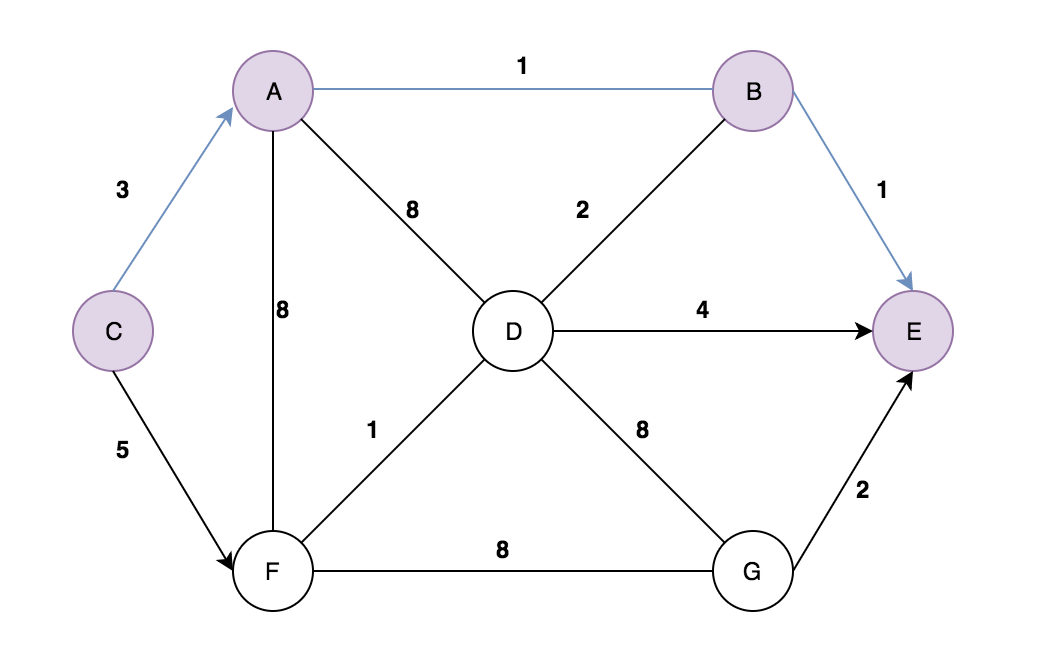
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **A** | **B** | **C** | **D** | **E** | **F** | **G** |
| **C** | 3 |  | 0 |  |  | 5 |  |
| **A** | 3 | 4 | 0 | 11 |  | 5 |  |
| **B** | 3 | 4 | 0 | 6 | 5 | 5 |  |
| **F** | 3 | 4 | 0 | 6 | 5 | 5 | 13 |
| **D** | 3 | 4 | 0 | 6 | 5 | 5 | 13 |
| **E** | 3 | 4 | 0 | 6 | 5 | 5 | 13 |
| **G** | 3 | 4 | 0 | 6 | 5 | 5 | 13 |

Dijkstra’s algorithm gives you shortest paths for from a starting node to all the other nodes while Yen’s is an extension of Dijkstra. Instead, it takes a source and destination node and it gives you the path’s for the 1st shortest, 2nd shortest 3rd shortest etc. for the distance between source and destination nodes. Starting off with the shortest node. Below will be a graph representation on how Yen’s algorithm does this given the pseudocode on page 16 and the graphs shown on each iteration.

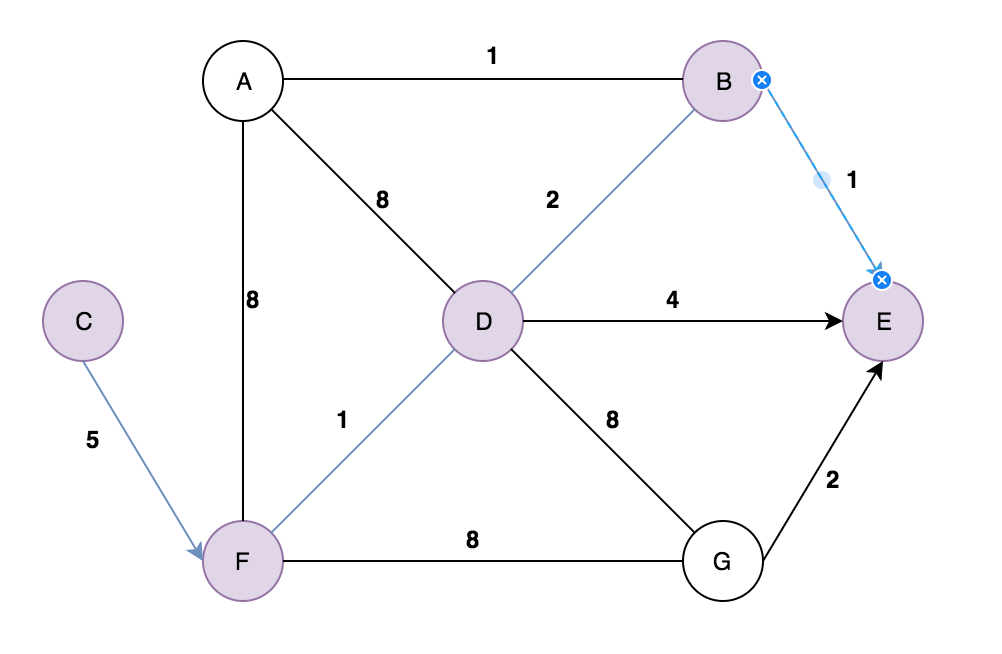
Note: For this part I am assuming indexes start at zero

Dijkstra: First Shortest Path as shown in line 2.

A[0] = CABE

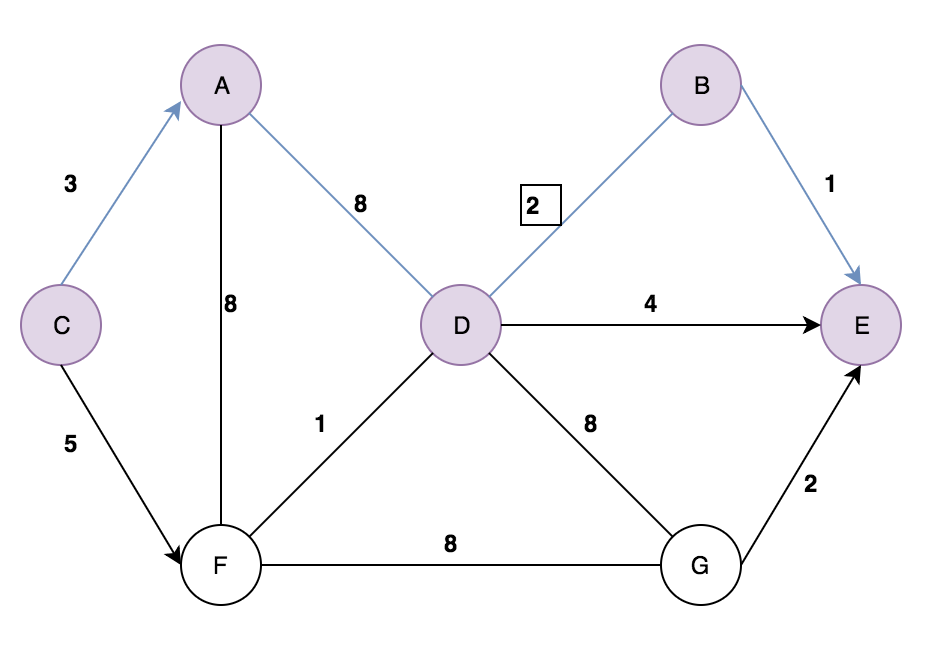


When k = 1 and i = 0;  
spurNode = C // Line 11  
rootPath = C // Line 14  
  
Remove Edge between C and A // Lines 19-21  
  
spurPath = CFDBE // Line 28  
totalPath = CFDBE, Cost = 9 // Line 31  
B = {CFDBE} // Line 33



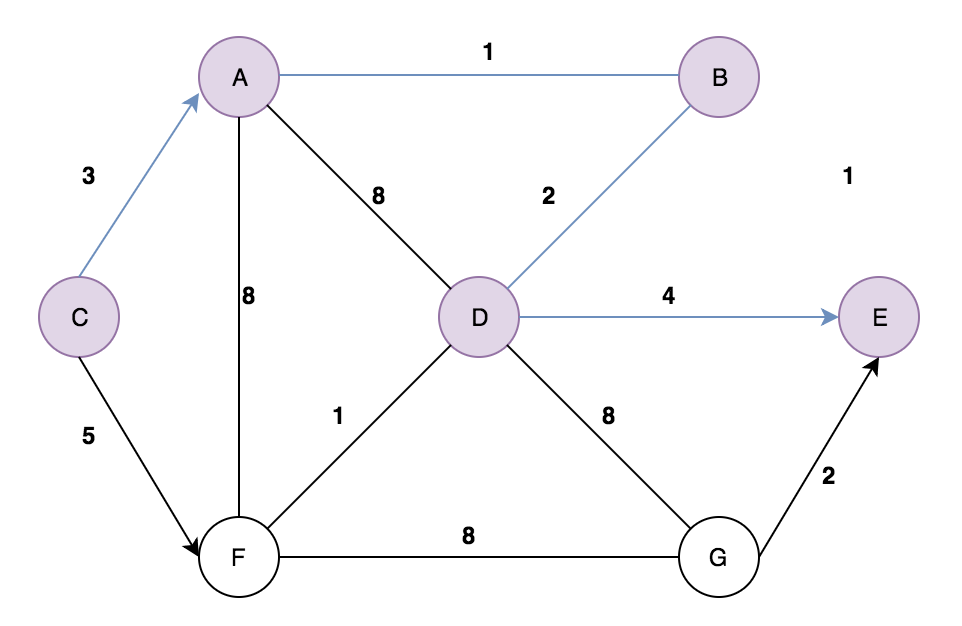
When k = 1 and i = 1;  
spurNode = A // Line 11  
rootPath = CA // Line 14  
  
Remove Edge between A and B // Lines 19-21

Remove Node C // Lines 23-24  
  
spurPath = ADBE // Line 28  
totalPath = CADBE, Cost = 14 // Line 31  
B = {CFDBE, CADBE} // Line 33



When k = 1 and i = 2;  
spurNode = B // Line 11  
rootPath = CAB // Line 14  
  
Remove Edge between B and E // Lines 19-21

Remove Nodes C and A // Lines 23-24  
  
spurPath = BDE // Line 28  
totalPath = CABDE, Cost = 10 // Line 31  
B = {CFDBE, CADBE, CABDE} // Line 33



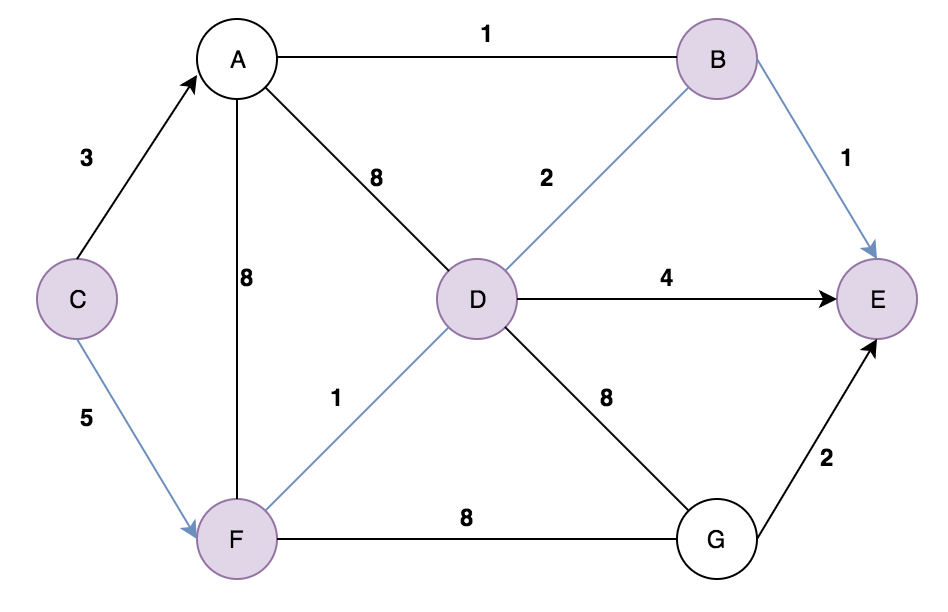
Now, Sort the potential k-shortest paths by costs, then select the first one (2nd Shortest one) as A[k] and pop it off the set. **Lines 44-47**

INPUT : B = {CFDBE, CADBE, CABDE} = {9, 14, 10}

OUTPUT: B = {CFDBE, CABDE, CADBE} = {9, 10, 14}

Dijkstra: Second Shortest path as shown in line 2

A[1] = CFDBE



When k = 2 and i = 0

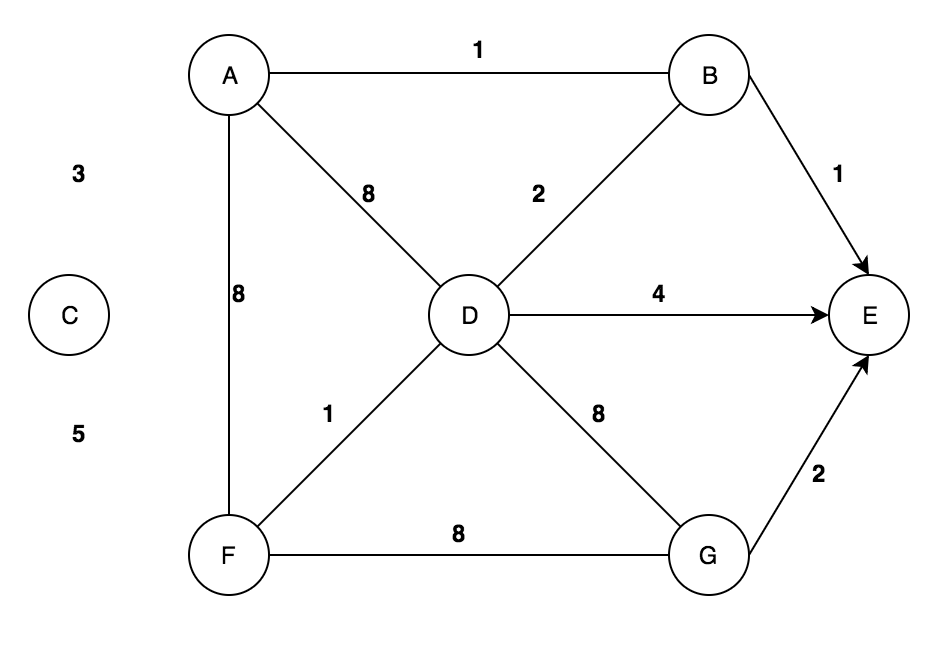
spurNode = C // Line 11  
rootPath = C // Line 14

Remove Edges between C,A and C,F // Line 19-21

spurPath = CANNOT BE FOUND

totalPath = CANNOT BE FOUND

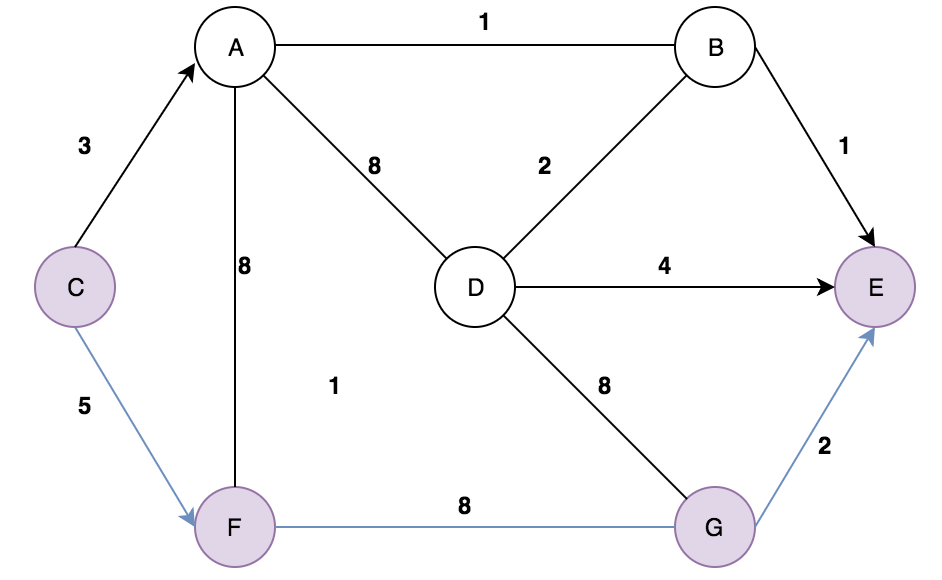
**Since, there are no ways on getting the total path, just move onto the next iteration.**



When k = 2 and i = 1

spurNode = F // Line 11  
rootPath = CF // Line 14  
  
Remove Edge between F and D // Lines 19-21

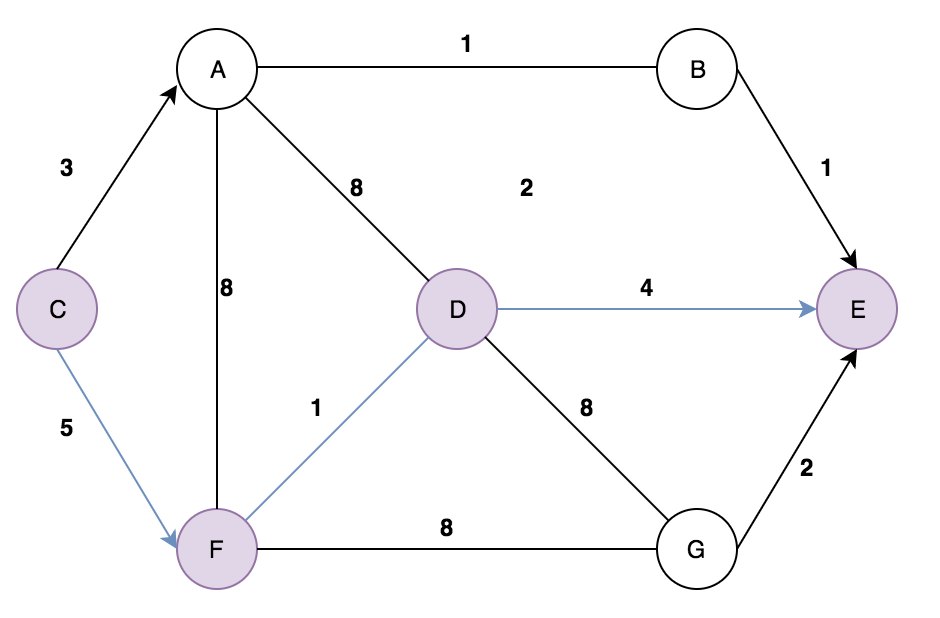
Remove Node C // Lines 23-24  
  
spurPath = FGE // Line 28  
totalPath = CFGE, Cost = 15 // Line 31  
B = {CADBE, CABDE, CFGE} // Line 33



When k = 2 and i = 2

spurNode = D // Line 11  
rootPath = CFD // Line 14  
  
Remove Edge between D and B // Lines 19-21

Remove Node C and F // Lines 23-24  
  
spurPath = DE // Line 28  
totalPath = CFDE, Cost = 10 // Line 31  
B = {CADBE, CABDE, CFGE, CFDE} // Line 33

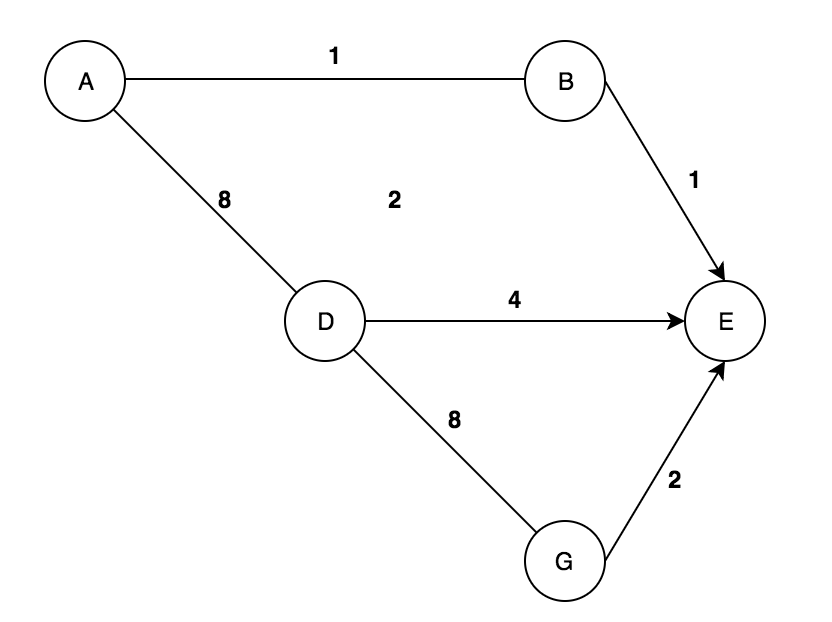


When k = 2 and i = 3

spurNode = B // Line 11  
rootPath = CFDB // Line 14  
  
Remove Edge between B and E // Lines 19-21

Remove Node C and F // Lines 23-24  
  
spurPath = CANNOT BE FOUND  
totalPath = CANNOT BE FOUND  
B = {CADBE, CABDE, CFGE, CFDE} // Line 33

**Since, there are no ways on getting the total path, just move onto the next iteration.**



Now, Sort the potential k-shortest paths by costs, then select the first one (3rd Shortest one) as A[k] and pop it off the set. **Lines 44-47**

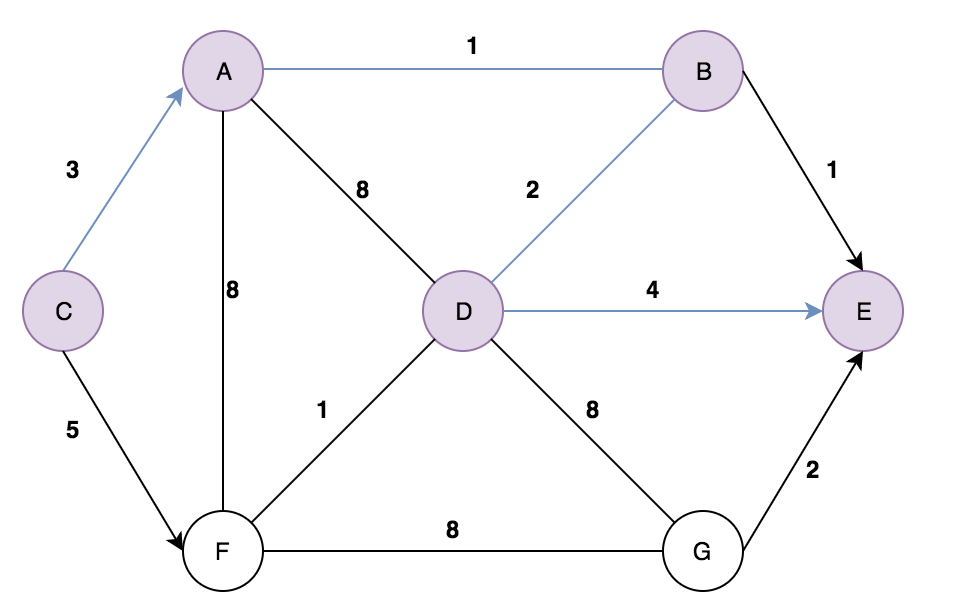
INPUT : B = {CADBE, CABDE, CFGE, CFDE} = {14, 10, 15, 10}

OUTPUT: B = {CABDE, CFDE, CADBE, CFGE} = {10, 10, 14, 15}

Note: Since there is a tie between two 10’s for cost. Just select the one in alphabetical order

Dijkstra: Third Shortest path as shown in line 2

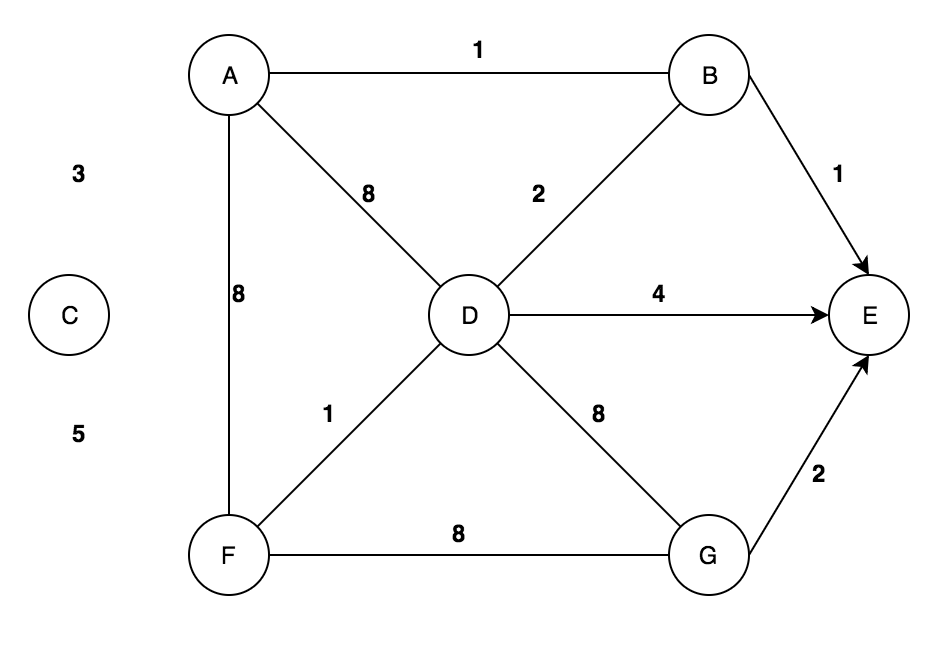
A[2] = CABDE



When k = 3 and i = 0

spurNode = C // Line 11  
rootPath = C // Line 14  
  
Remove Edges between C,A and C,F // Lines 19-21  
  
spurPath = CANNOT BE FOUND  
totalPath = CANNOT BE FOUND

**Since, there are no ways on getting the total path, just move onto the next iteration.**

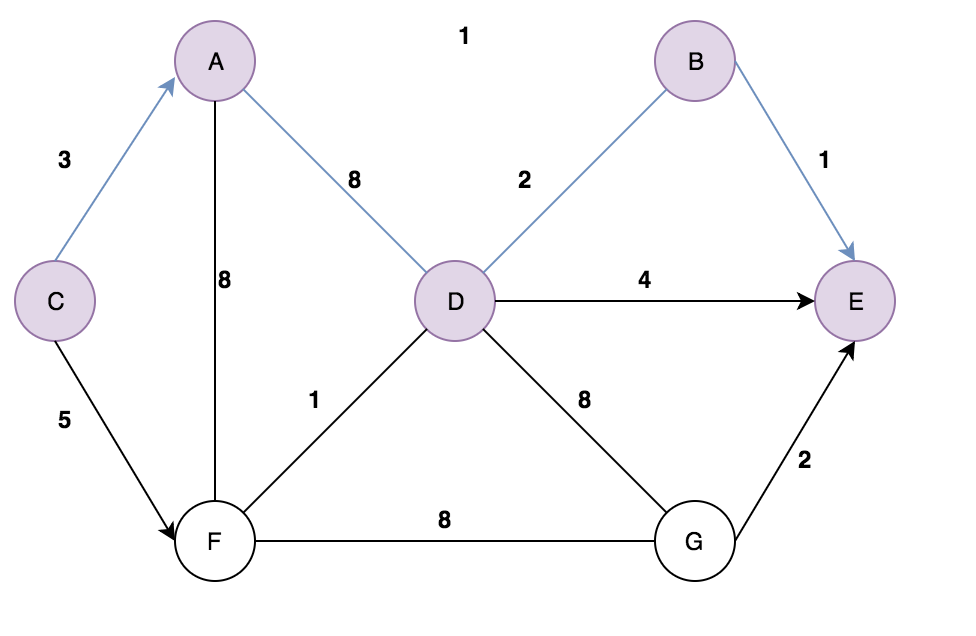


When k = 3 and i = 1

spurNode = A // Line 11  
rootPath = CA // Line 14  
  
Remove Edges between A,B // Lines 19-21

Remove Node C // Lines 23-24  
  
spurPath = ADBE // Line 28  
totalPath = CADBE, Cost = 14 // Line 31  
B = {CFDE, CADBE, CFGE} // Line 33

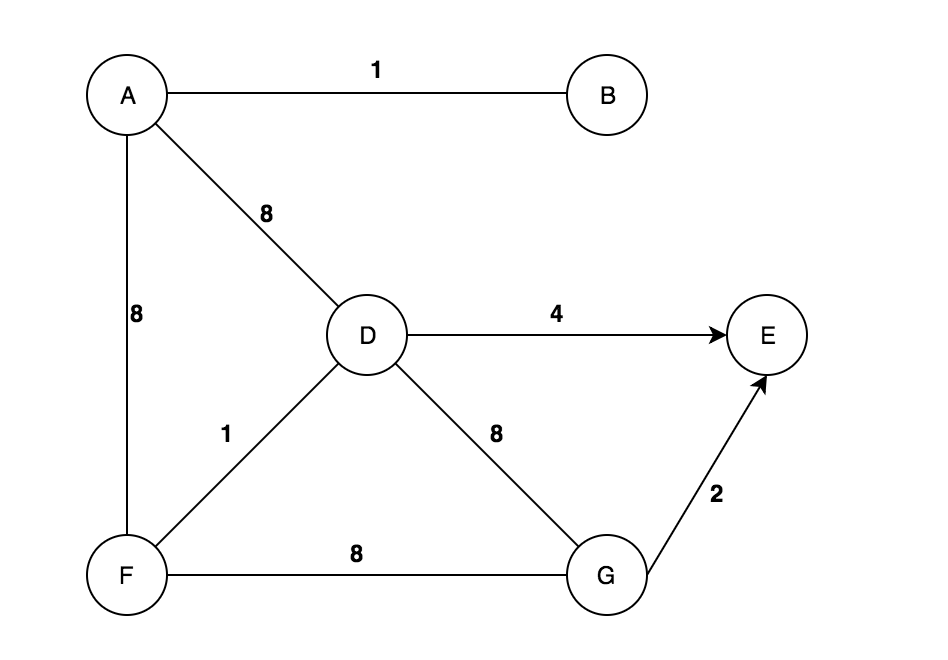
Since, the total path CADBE is already in the set it is just updated with the same value, no duplicates can happen in a set



When k = 3 and i = 2

spurNode = A // Line 11  
rootPath = CA // Line 14  
  
Remove Edges between B,E and B,D // Lines 19-21

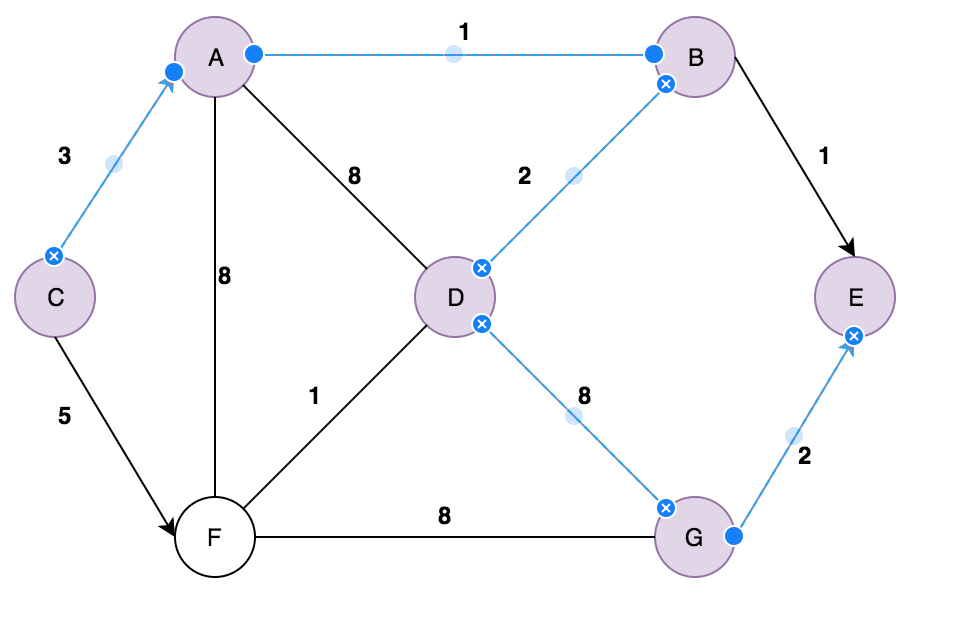
Remove Node C // Lines 23-24  
  
spurPath = CANNOT BE FOUND // Line 28  
totalPath = CANNOT BE FOUND // Line 31  
Because of removing the edges BE and BD you cannot find a totalPath so skip this iteration.



When k = 3 and i = 3

spurNode = D // Line 11  
rootPath = CABD // Line 14  
  
Remove Edges between D,E // Lines 19-21

Remove Node C, A, B // Lines 23-24  
  
spurPath = DGE // Line 28  
totalPath = CABDGE, Cost = 16 // Line 31  
B = {CFDE, CADBE, CFGE, CABDGE} // Line 33

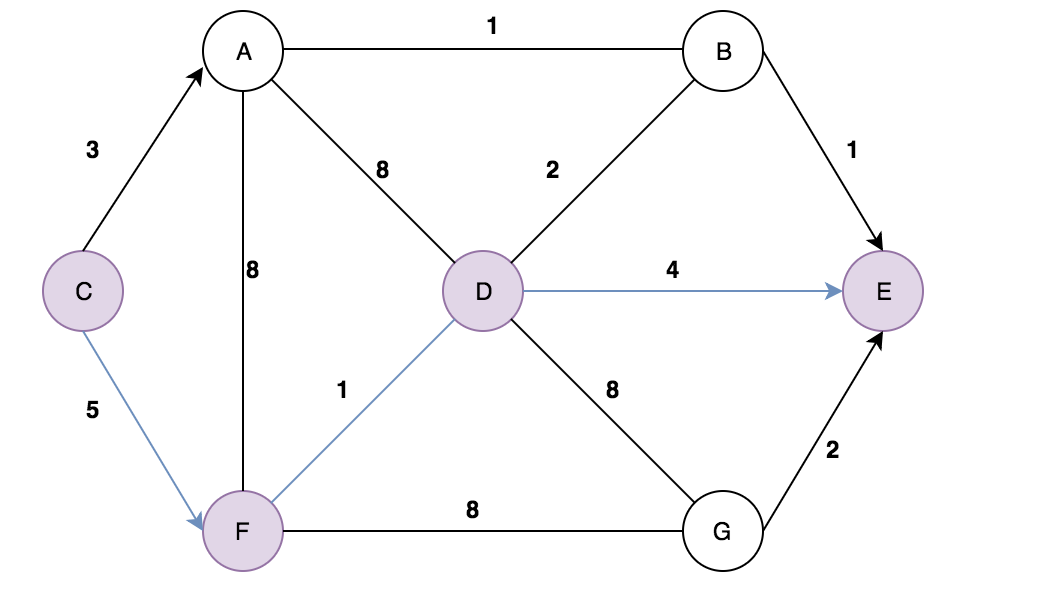


Now, Sort the potential k-shortest paths by costs, then select the first one (4th Shortest one) as A[k] and pop it off the set. **Lines 44-47**

INPUT : B = {CFDE, CADBE, CFGE, CABDGE} = {10, 14, 15, 16}

OUTPUT: B = {CFDE, CADBE, CFGE, CABDGE} = {10, 14, 15, 16}

A[4] = CFDE, then pop it from the set.



When k = 4 and i = 0

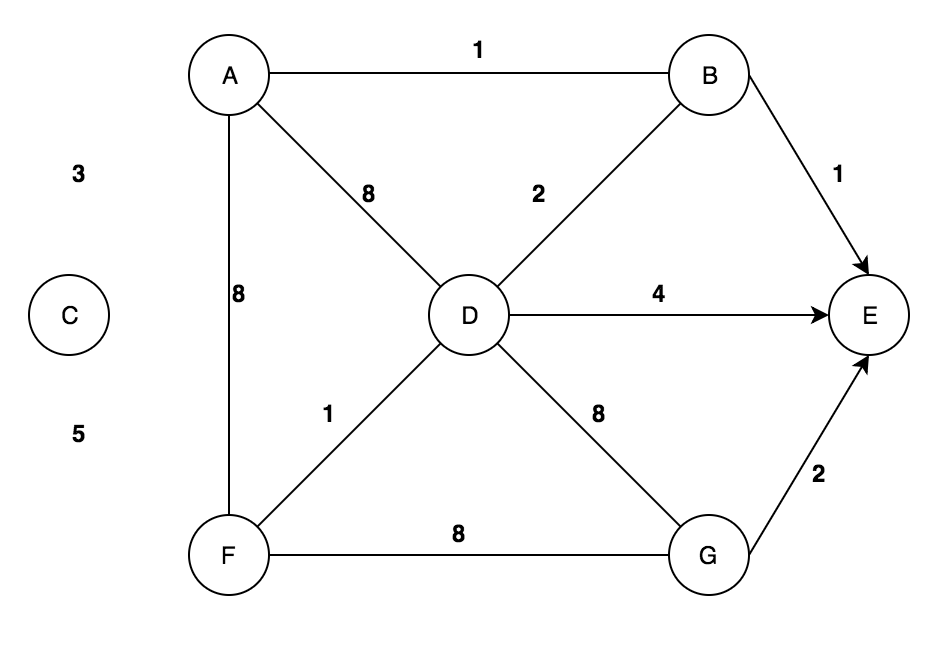
spurNode = C // Line 11  
rootPath = C // Line 14

Remove Edges between C,A and C,F // Line 19-21

spurPath = CANNOT BE FOUND

totalPath = CANNOT BE FOUND

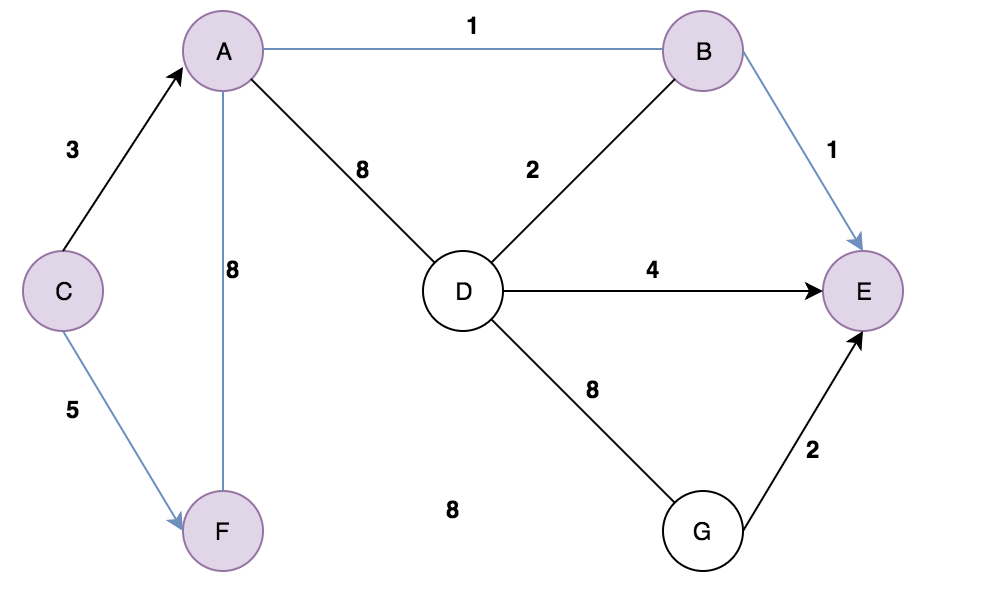
**Since, there are no ways on getting the total path, just move onto the next iteration.**



When k = 4 and i = 1

spurNode = F // Line 11  
rootPath = CF // Line 14  
  
Remove Edges between FG and FD // Lines 19-21

Remove Node C // Lines 23-24  
  
spurPath = FABE // Line 28  
totalPath = CFABE, Cost = 15 // Line 31  
B = {CADBE, CFGE, CABDGE, CFABE} // Line 33

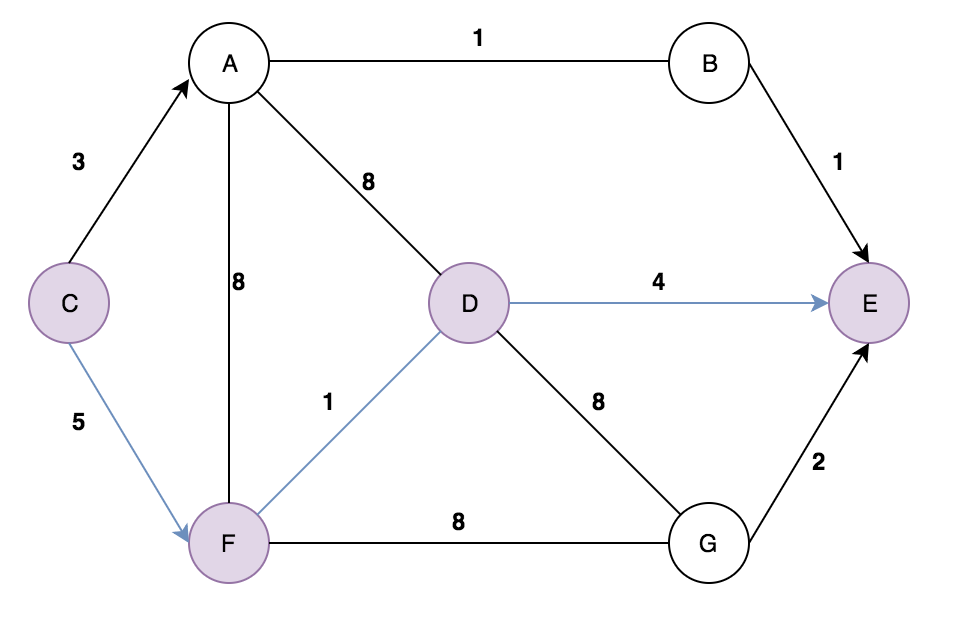


When k = 4 and i = 2

spurNode = D // Line 11  
rootPath = CFD // Line 14  
  
Remove Edges between DB // Lines 19-21

Remove Node C and F // Lines 23-24  
  
spurPath = DE // Line 28  
totalPath = CFDE, Cost = 15 // Line 31  
B = {CADBE, CFGE, CABDGE, CFABE} // Line 33

NOTE: Since this is the same as Dijkstra’s 4th shortest path, it would not be added into the set B.



# Question 3b:

**Discuss the time complexity of Yen’s algorithm. Your complexity derivation must be calculated following each step of the pseudocode of the algorithm.**

**Note: I’m assuming the nodes of the graph is being stored in an Adjacency List and B is being stored as a set.**

function YenKSP(Graph, source, sink, K):  
 A[0] = Dijkstra(Graph, source, sink); O((|E|+|V|)lg|V|)  
 B = {}; O(1)  
   
 for k from 1 to K: O(K)  
 for i from 0 to size(A[k − 1]) − 1: O(K-1)  
 spurNode = A[k-1].node(i); O(1)  
 rootPath = A[k-1].nodes(0, i); O(1)  
   
 for each path p in A: O(K)  
 if rootPath == p.nodes(0, i): O(1)  
 remove p.edge(i,i + 1) from Graph; O(|E|)  
   
 for each node rootPathNode in rootPath except spurNode: O(V)  
 remove rootPathNode from Graph; O(V)  
   
 spurPath = Dijkstra(Graph, spurNode, sink); O((|E|+|V|)lg|V|)  
   
 totalPath = rootPath + spurPath; O(1)  
 B.append(totalPath); O(1)  
   
 restore edges to Graph; O(|E|)  
 restore nodes in rootPath to Graph; O(V)  
   
 if B is empty: O(1)  
 break; O(1)

B.sort(); O(nlogn) // Mergesort  
 A[k] = B[0]; O(1)  
 B.pop(); O(1)  
   
 return A; O(1)

**Time Complexity:**

O( (|E|+|V|) lg|V|) ) + O(K + K\*E + V + (|E|+|V|) lg|V|) + E + V + nlogn)

= O(|E|+|V|lg|V|) \* KE

Overall, the time complexity of this above implementation of Yen’s algorithm, is O(|E| + |V|lg|V|) \* KE because Yen’s algorithm is done the same way as Dijkstra’s except done KE times where K is each iteration of the shortest path (eg. 1st shortest, 2nd shortest etc.) and E is the edges of the graph as represented by the first two for loops in the algorithm. Although there are other operations such as nlogn and V they are all bounded by the outer for loop and since O((|E|+|V|)lg|V|) is the most dominating iteration so