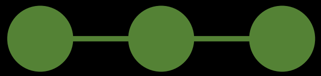
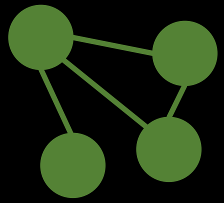
Final Exam Review

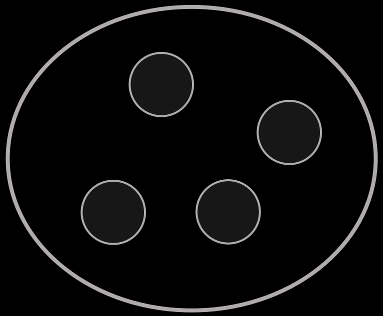
Data Structures & Algorithm

Categories of Data Structures

Linear Ordered	Non-linear Ordered	Not Ordered
Lists	Trees	Sets
Stacks	Graphs	Tables/Maps
Queues		



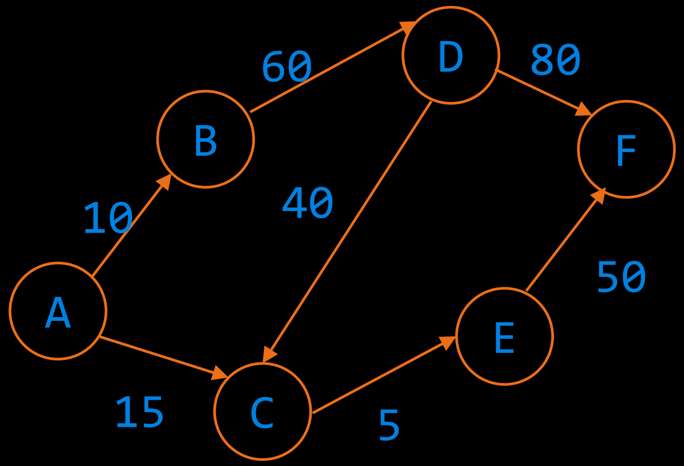




Announcements

- You must take the exam between:
  - 6pm – 11:59pm EST on July 26 [All students except UFOL/UDER]
  - 6pm July 26 to 11:59 pm July 26 [UFOL/UDER students]
- The exam will be over Honorlock and you are allowed one double sided handwritten sheet of notes.
- The exam duration is 2 hours. This means you must start by 10 pm EST or else you will lose time.
- Exam 2 Topics and Expectations Guide: Link
- Exam reviews: Exam 2 Resources

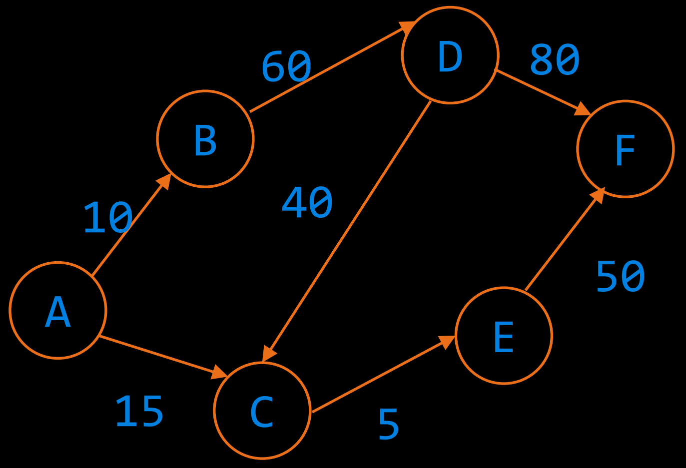
Common Representations



G

- Edge List
- Adjacency Matrix
- Adjacency List

Edge List

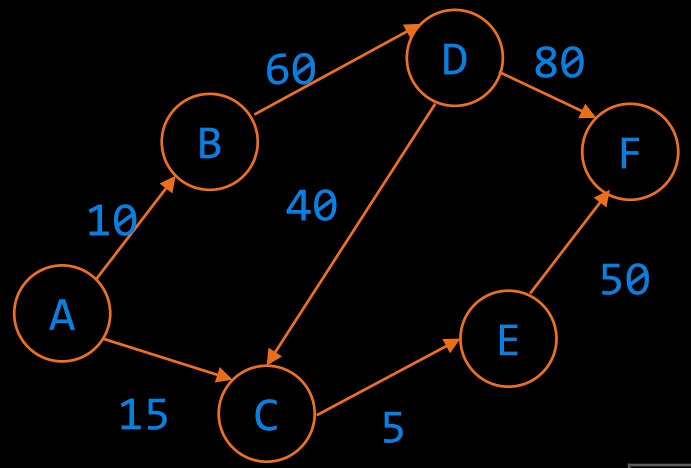


G

A	B	10
A	C	15
B	D	60
D	C	40
D	F	80
E	F	50
C	E	5

G = {(A,B), (A,C), (B,D), (D,C), (D,F), (E,F), (C,E)}

Edge List



G = {(A,B), (A,C), (B,D), (D,C), (D,F), (E,F), (C,E)}

G

A	B	10
A	C	15
B	D	60
D	C	40
D	F	80
E	F	50
C	E	5

Common Operations:

1. Connectedness

Is A connected to B?

~ O(E)

2. Adjacency

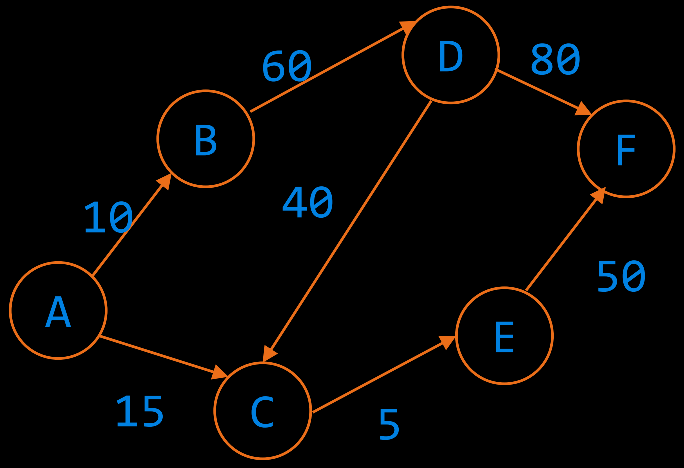
What are A's adjacent nodes?

~ O(E)

O(|E|) ~ O(|V| * |V|)

Space: O(E)

Adjacency Matrix



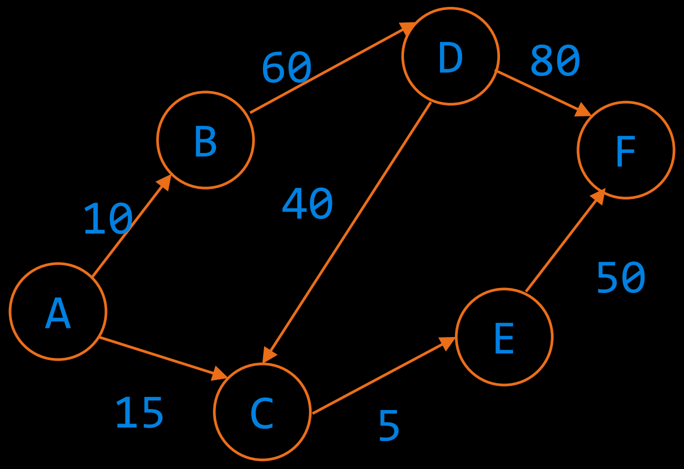
G

	A	B	C	D	E	F
A	0	10	15	0	0	0
B	0	0	0	60	0	0
C	0	0	0	0	5	0
D	0	0	40	0	0	80
E	0	0	0	0	0	50
F	0	0	0	0	0	0

Insertion:

```
G[from][to] = weight; (if there is an edge, "from" -> "to")
G[from][to] = 0;           (otherwise)
```

Adjacency Matrix Implementation



Insertion:

G[from][to] = weight; (if there is an edge, “from” -> “to”)

G[from][to] = 0; (otherwise)

Input
7
A B 10
A C 15
B D 60
D C 40
C E 5
D F 80
E F 50

G

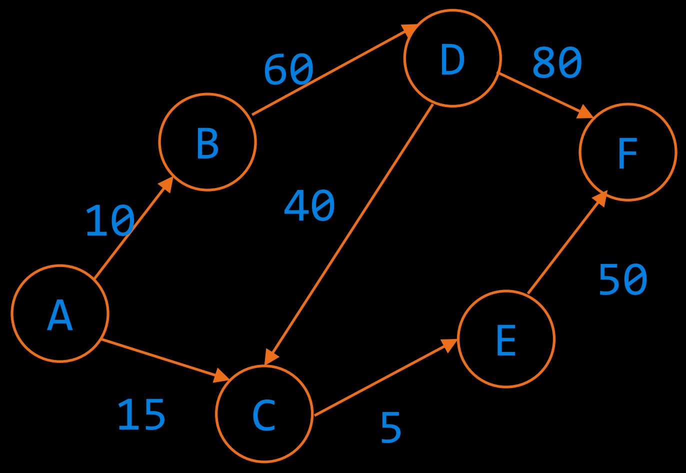
Map
A 0
B 1
C 2
D 3
E 4
F 5

	0	1	2	3	4	5
0	0	10	15	0	0	0
1	0	0	0	60	0	0
2	0	0	0	0	5	0
3	0	0	40	0	0	80
4	0	0	0	0	0	50
5	0	0	0	0	0	0

```
#include <iostream>
#include<map>
#define VERTICES 6
using namespace std;
int main()
{
        int no_lines, wt, j=0;
        string from, to;
        int graph [VERTICES][VERTICES] = {0};
        map<string, int> mapper;
        cin >> no_lines;
        for(int i = 0; i < no_lines; i++)
        {
                cin >> from >> to >> wt;
                if (mapper.find(from) == mapper.end())
                        mapper[from] = j++;
                if (mapper.find(to) == mapper.end())
                        mapper[to] = j++;
                graph[mapper[from]][mapper[to]] = wt;
        }
        return 0;
}
```

https://www.onlinegdb.com/Hy8M0CnsS

Adjacency Matrix



G

Map	
A	0
B	1
C	2
D	3
E	4
F	5

	0	1	2	3	4	5
0	0	10	15	0	0	0
1	0	0	0	60	0	0
2	0	0	0	0	5	0
3	0	0	40	0	0	80
4	0	0	0	0	0	50
5	0	0	0	0	0	0

Common Operations:

1. Connectedness

   Is A connected to B?

   G[“A”][“B”] ~ O(1)

2. Adjacency

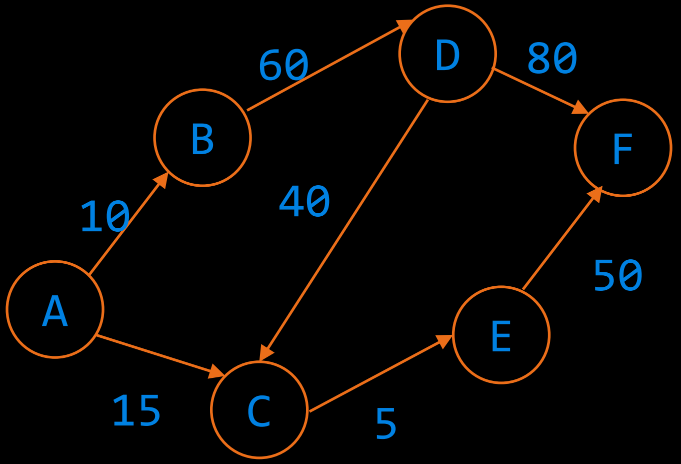
   What are A’s adjacent nodes?

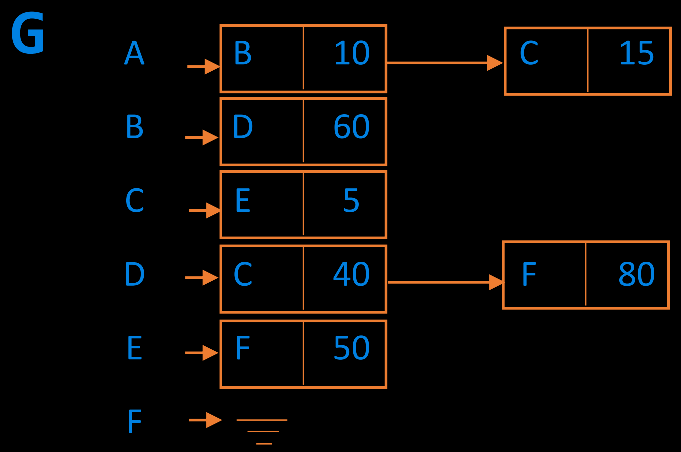
   for each element x in G[“A”]
       if x ! = 0

   ~ O(|V|)

Space: O(|V| * |V|)

Adjacency List





Sparse Graph:
Edges ~ Vertices

Common Operations:

1. Connectedness

   Is A connected to B?

   for each element x in G[“A”]
   if x ! = ‘B’
   ~ O(outdegree|V|)

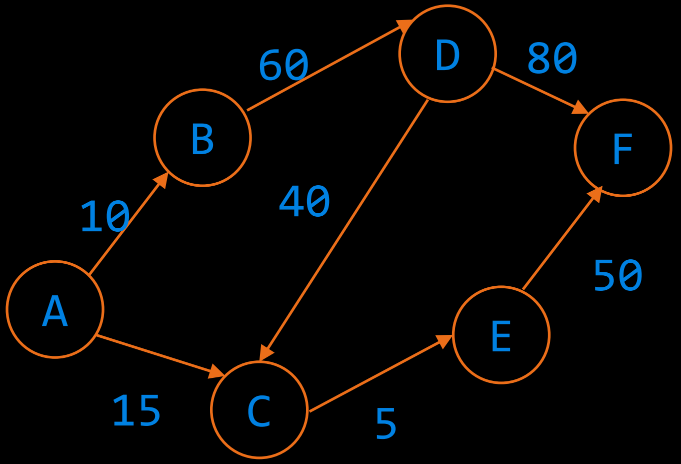
2. Adjacency

   What are A’s adjacent nodes?

   G[“A”] ~ O(outdegree|V|)

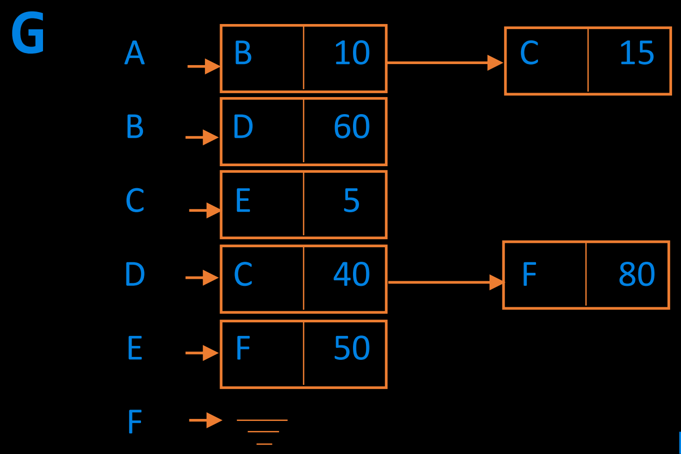
Space: **O(|V| + |E|)**

Adjacency List Implementation



Input

7
A B 10
A C 15
B D 60
D C 40
C E 5
D F 80
E F 50



Insertion:

If to or from vertex not present add vertex

Otherwise add edge at the end of the list

```
#include <iostream>
#include<map>
#include<vector>
#include<iterator>
using namespace std;

int main()
{
        int no_lines;
        string from, to, wt;
        map<string, vector<pair<string,int>>> graph;
        cin >> no_lines;
        for(int i = 0; i < no_lines; i++)
        {
                cin >> from >> to >> wt;
                graph[from].push_back(make_pair(to, stoi(wt)));
                if (graph.find(to)==graph.end())
                        graph[to] = {};
        }
}
```

https://onlinegdb.com/HkJq9iFaI

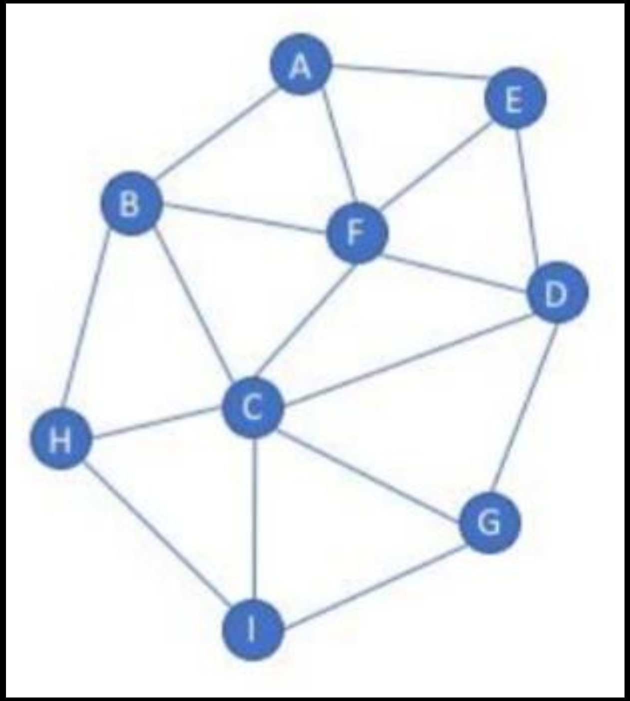
Graph Implementation

	Edge List	Adjacency Matrix	Adjacency List
Time Complexity: Connectedness	O(E)	O(1)	O(outdegree(V))
Time Complexity: Adjacency	O(E)	O(V)	O(outdegree(V))
Space Complexity	O(E)	O(V*V)	O(V+E)

Alt Text for the Graph on Next Slide

Vertex: Neighbors of Vertex (Edges pointing from a vertex to the neighbor)

A: B, E, F
B: A, C, F, H
C: B, D, F, G, H, I
D: C, E, F, G
E: A, D, F
F: A, B, C, D, E
G: C, D, I
H: B, C, I
I: C, G, H



Graph - BFS

- Which of the following are valid breadth first search traversals for this graph?

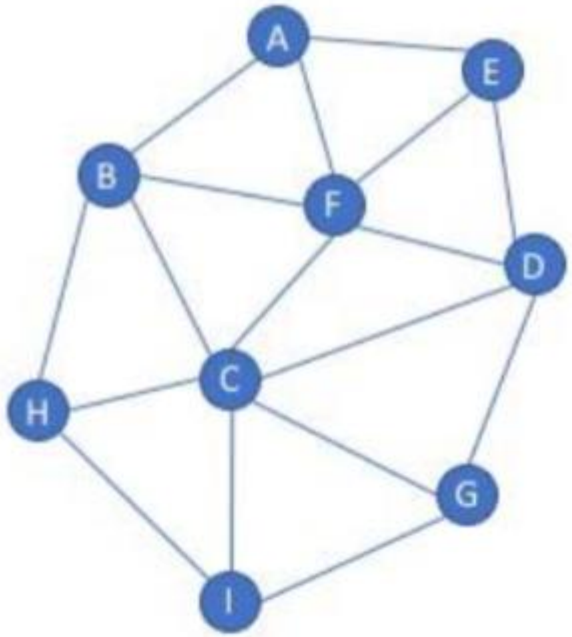
a) A F B E D C H G I

b) I C H G B F D A E

c) D C F E G H I B A

d) E A F D B H C I G

e) F A E D C B G I H



Graph - BFS

- Which of the following are valid breadth first search traversals for this graph?

a) A F B E D C H G I

b) I C H G B F D A E

c) D C F E G H I B A

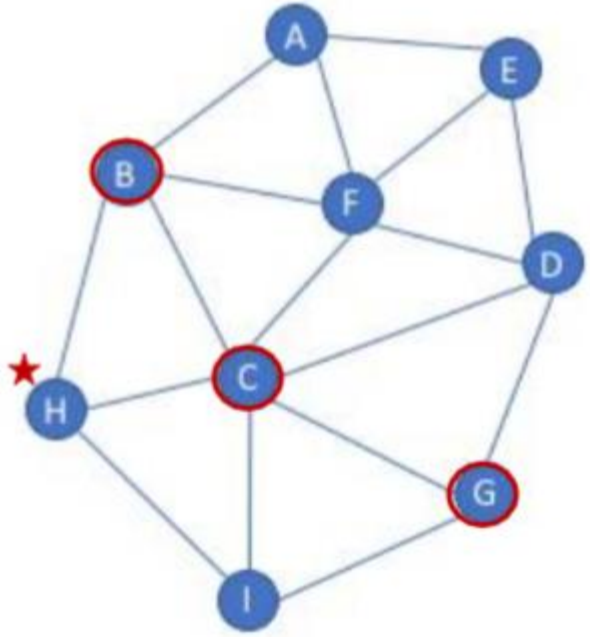
d) E A F D B H C I G

e) F A E D C B G I H

All the options except for d

Why not d?

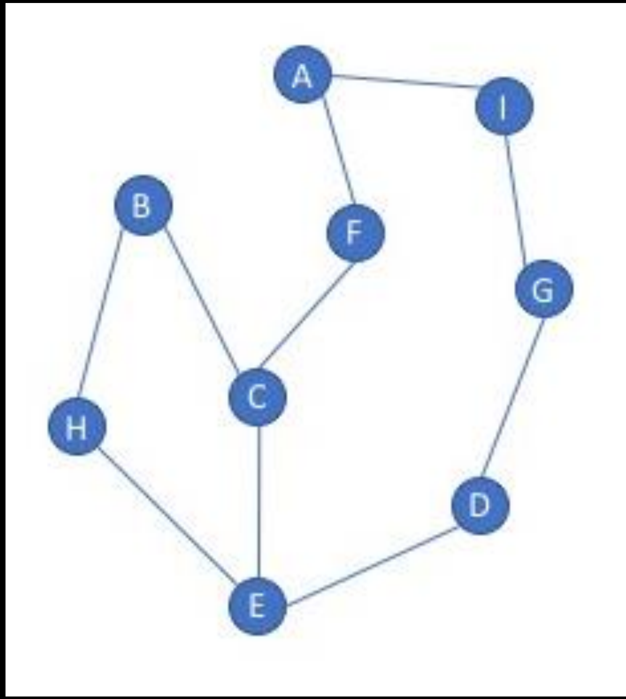
** H is visited before C and G

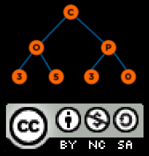


Alt Text for the Graph on Next Slide

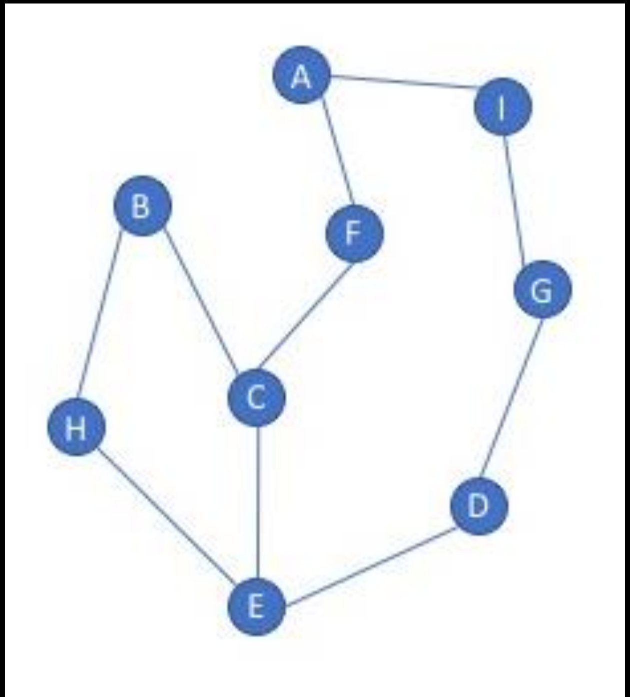
Vertex: Neighbors of Vertex (Edges pointing from a vertex to the neighbor)

A: F, I
B: C, H
C: B, E, F
D: E, G
E: C, D, H
F: A, C
G: D, I
H: B, E
I: A, G



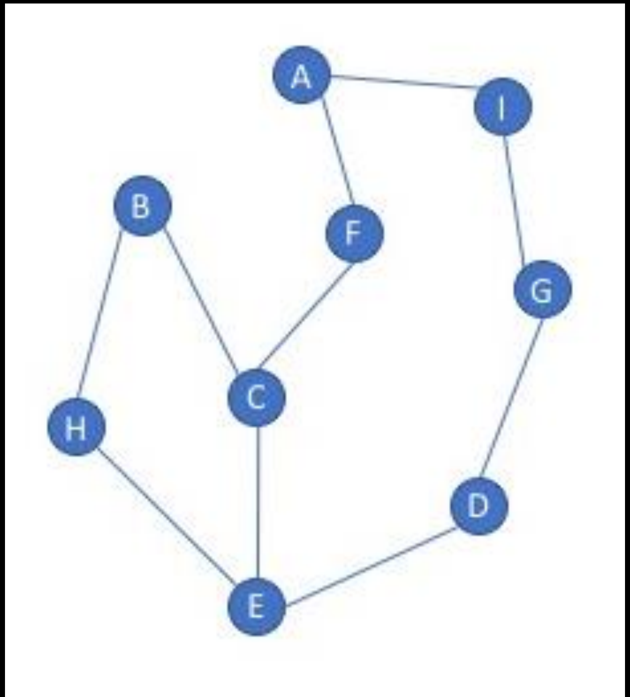


Valid DFS: Which DFS are valid?



- H E C B D G I A F
- C E H B D G I A F
- A F C E H B I G D
- D E C B H F A I G

Valid DFS: Which DFS are valid?



- H E C B D G I A F
- C E H B D G I A F
- A F C E H B I G D
- D E C B H F A I G

Applied Traversal

A connected component of a graph is a collection of vertices in which any two vertices in a component have a path between them.Given an unweighted and undirected graph represented as an adjacency list, write a function using pseudocode or C++ code which will return the number of vertices in the largest component of the graph. You do not need to write main method for reading and creating a graph. Assume that the graph is passed into your function which has the following signature:

```
unsigned int largest_component(unordered_map<int, vector<int>>& adjListGraph){
        // code here
}
```

Example input:

```
0: 2, 3
1: 4
2: 0, 6
3: 0
4: 1, 5
5: 4
6: 2
7:
```

Example output: 4

Explanation: There are three connected components: (0,2,3,6); (1,4,5); and (7). Of these, the largest has 4 vertices.

Also, state the Big O complexity of the solution in the worst case.

Applied Traversal

```cpp
#include <unordered_map>
#include <vector>
#include <unordered_set>
using namespace std;

unsigned int dfs(int node, unordered_map<int, vector<int>>& adjListGraph, unordered_set<int>& visited) {
    visited.insert(node);
    unsigned int size = 1;  // Start with the current node

    for (int neighbor : adjListGraph[node]) {
        if (visited.find(neighbor) == visited.end()) {
            size += dfs(neighbor, adjListGraph, visited);
        }
    }
    return size;
}

unsigned int largest_component(unordered_map<int, vector<int>>& adjListGraph) {
    unordered_set<int> visited;
    unsigned int maxComponentSize = 0;

    for (const auto entry : adjListGraph) {
        int node = entry.first;
        if (visited.find(node) == visited.end()) {
            unsigned int componentSize = dfs(node, adjListGraph, visited);
            if (componentSize > maxComponentSize) {
                maxComponentSize = componentSize;
            }
        }
    }

    return maxComponentSize;
}
```

BFS vs DFS

```
string source = "A";
std::set<string> visited;
std::queue<string> q;

visited.insert(source);
q.push(source);
cout<<"BFS: ";

while(!q.empty())
{
	string u = q.front();
	cout << u;
	q.pop();
	vector<string> neighbors = graph[u];
	for(string v: neighbors)
	{
		if(visited.count(v)==0)
		{
			visited.insert(v);
			q.push(v);
		}
	}
}
```

```
string source = "A";
std::set<string> visited;
std::stack<string> s;

visited.insert(source);
s.push(source);
cout<<“DFS: ";

while(!q.empty())
{
	string u = s.top();
	cout << u;
	s.pop();
	vector<string> neighbors = graph[u];
	for(string v: neighbors)
	{
		if(visited.count(v)==0)
		{
			visited.insert(v);
			s.push(v);
		}
	}
}
```

Theoretical Complexity: O(V+E)

BFS Pseudocode

- Write pseudocode/code for implementing the **Breadth First Search Algorithm** of a graph, G that takes a source vertex S as input. (8).
- Also, state the Big O complexity of the traversal in the worst case (2).

Graph Algorithm Mix n Match

- Finds the shortest paths in a weighted graph
- Find the minimum cost connected network
- Scheduling algorithm, list steps in a process
- Finds the shortest path in an unweighted graph

Prim’s or Kruskals
BFS
DFS
Topological Sort
Dijkstra’s Algorithm

Graph Algorithm Mix n Match

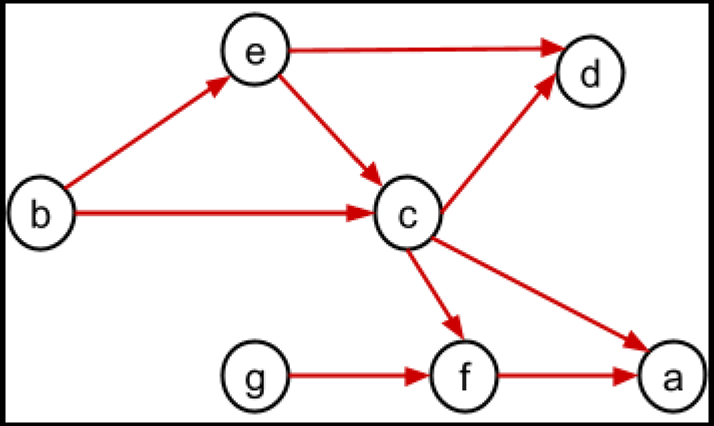
- Finds the shortest paths in a weighted graph
- Find the minimum cost connected network
- Scheduling algorithm, list steps in a process
- Finds the shortest path in an unweighted graph

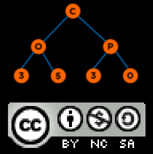
Prim's or Kruskals
BFS
DFS
Topological Sort
Dijkstra's Algorithm

Alt Text for the Graph on Next Slide

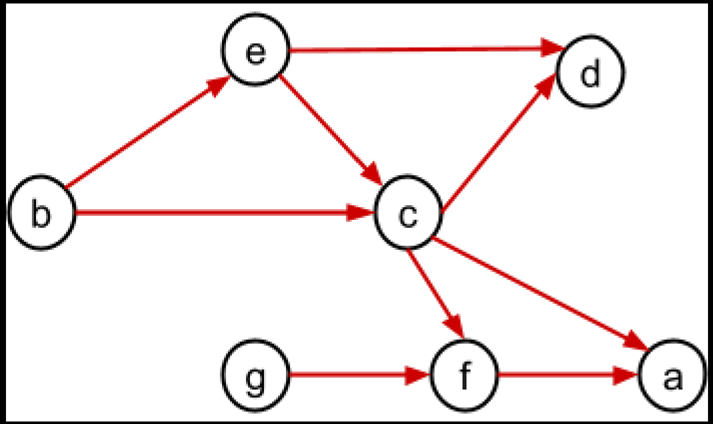
Vertex: Neighbors of Vertex (Edges pointing from a vertex to the neighbor)

A: -
B: C, E
C: A, D, F
D: -
E: C, D
F: A
G: F



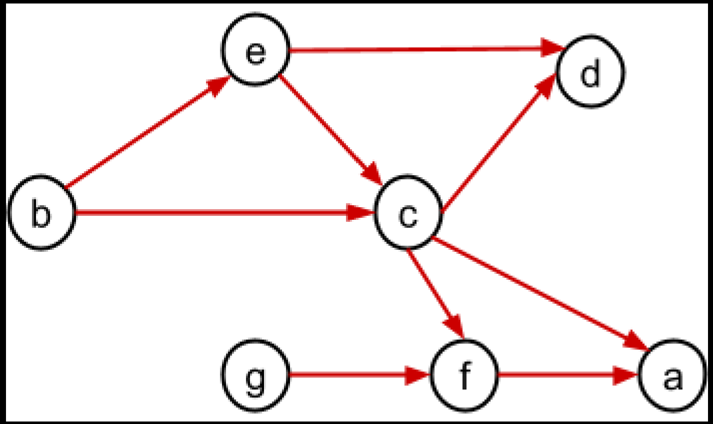


Which of the choices below represent a valid topological sort ordering of this graph?



- b, e, c, g, f, a, d
- b, a, c, g, f, e, d
- b, g, f, c, e, a, d
- b, e, c, g, a, f, d
- b, g, e, c, d, f, a
- b, f, c, g, a, e, d

Which of the choices below represent a valid topological sort ordering of this graph?



- b, e, c, g, f, a, d
- b, a, c, g, f, e, d
- b, g, f, c, e, a, d
- b, e, c, g, a, f, d
- b, g, e, c, d, f, a
- b, f, c, g, a, e, d

What does this code do?

```
#include <set>
#include <stack>
using namespace std;

bool doSomething(const Graph& graph, int src, int dest)
{
    set<int> visited;
    stack<int> s;
    visited.insert(src);
    s.push(src);
    while(!s.empty())
    {
        int u = s.top();
        s.pop();
        for(auto v: graph.adjList[u])
        {
            if(v == dest)
                return true;
            if ((visited.find(v) == visited.end())) {
                visited.insert(v);
                s.push(v);
            }
        }
    }
    return false;
}
```

What does this code do?

```
#include <set>
#include <stack>
using namespace std;

bool doSomething(const Graph& graph, int src, int dest)
{
    set<int> visited;
    stack<int> s;
    visited.insert(src);
    s.push(src);
    while(!s.empty())
    {
        int u = s.top();
        s.pop();
        for(auto v: graph.adjList[u])
        {
            if(v == dest)
                return true;
            if ((visited.find(v) == visited.end())) {
                visited.insert(v);
                s.push(v);
            }
        }
    }
    return false;
}
```

Returns whether a given vertex is reachable from another vertex using DFS

Scenario

A county government maintains a network of roads. The county government has tabulated the cost of maintaining each road. They need to minimize the cost of road maintenance but ensure that all places in the county are accessible.

Which graph algorithm that we discussed in class could they use to solve this problem? What are the vertices, what are the edges, what are the edge values?

Scenario

A county government maintains a network of roads. The county government has tabulated the cost of maintaining each road. They need to minimize the cost of road maintenance but ensure that all places in the county are accessible.

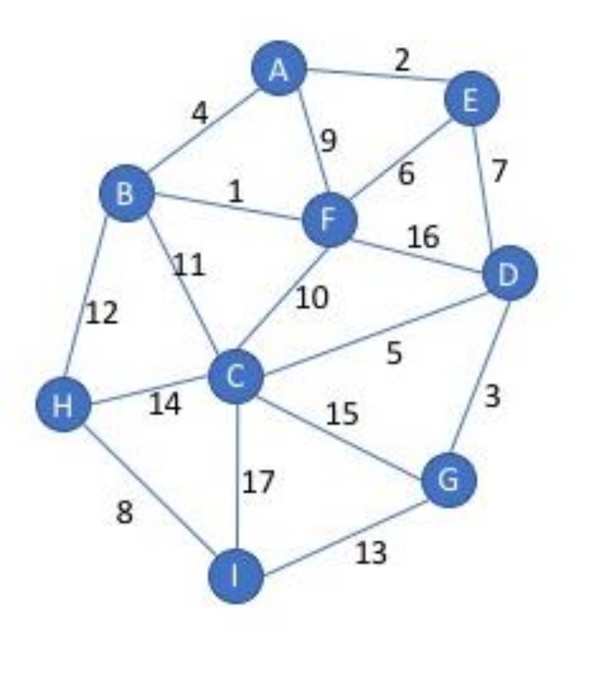
Which graph algorithm that we discussed in class could they use to solve this problem? What are the vertices, what are the edges, what are the edge values?

- Prim's or Kruskals algorithm for minimum spanning tree.
- Roads are edges.
- Ends of roads are vertices.
- Edge weights are cost for maintaining roads.

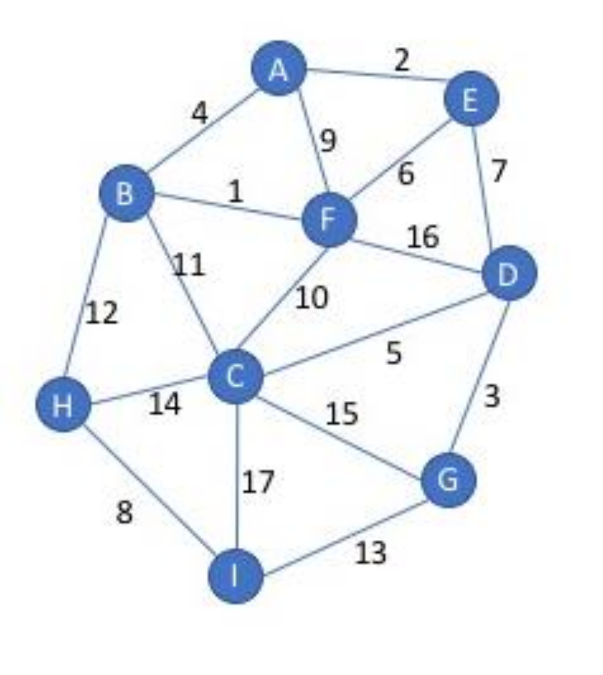
Alt Text for the Graph on Next Slide

Vertex: <Neighbors of Vertex (Edges pointing from a vertex to the neighbor), edge weight>

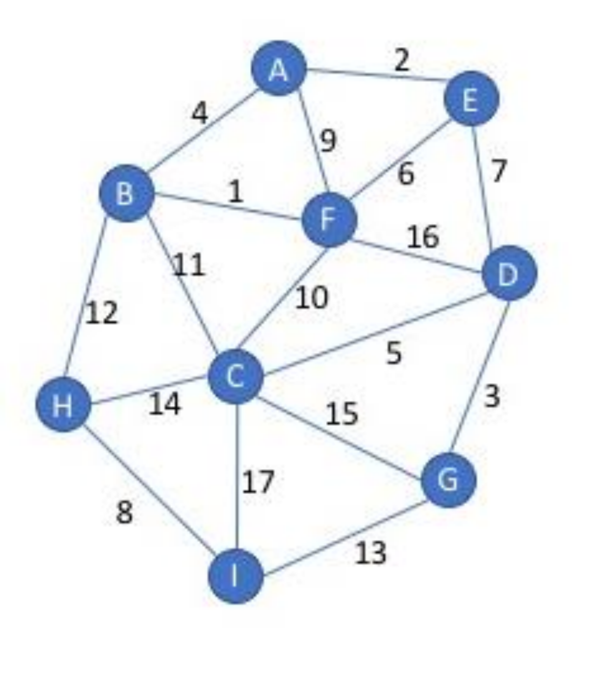
A: <B, 4>, <E, 2>, <F, 9>
B: <A, 4>, <C, 11>, <F, 1>, <H, 12>
C: <B, 11>, <D, 7>, <F, 10>, <G, 15>, <H, 14>, <I, 17>
D: <C, 5>, <E, 7>, <F, 16>, <G, 3>
E: <A, 2>, <D, 7>, <F, 6>
F: <A, 9>, <B, 1>, <C, 10>, <D, 16>, <E, 6>
G: <C, 15>, <D, 3>, <I, 13>
H: <B, 12>, <C, 14>, <I, 8>
I: <C, 17>, <G, 13>, <H, 8>



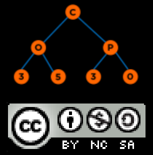
MST using Prims starting from “I”



MST using Prims starting from “I”



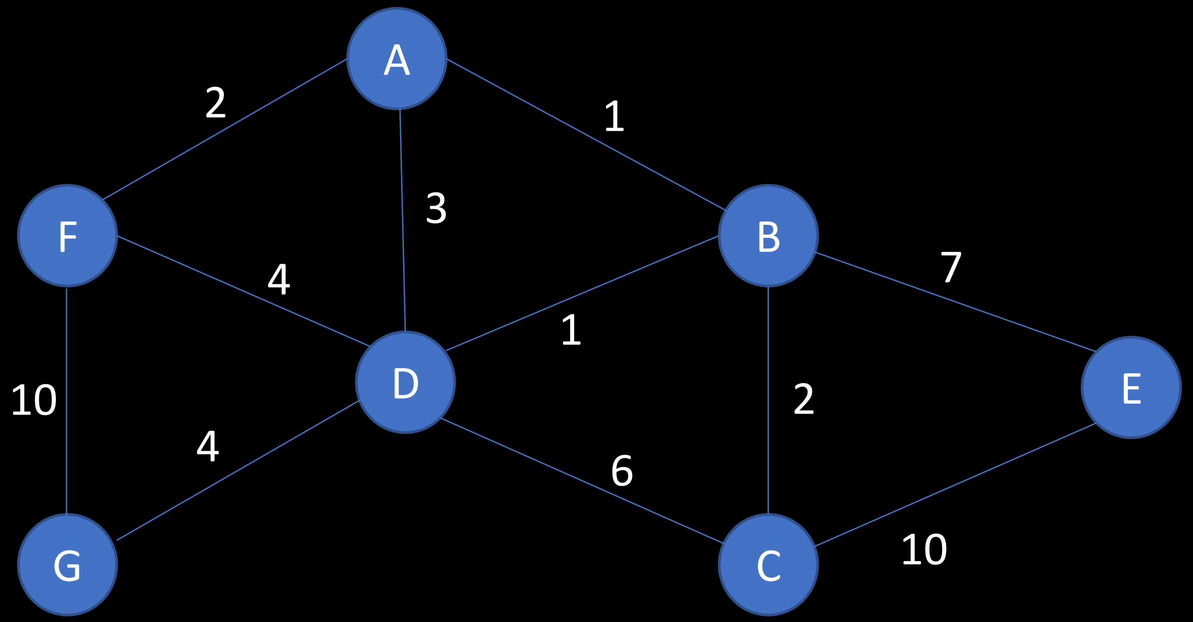
I H B F A E D G C



Alt Text for the Graph on Next Slide

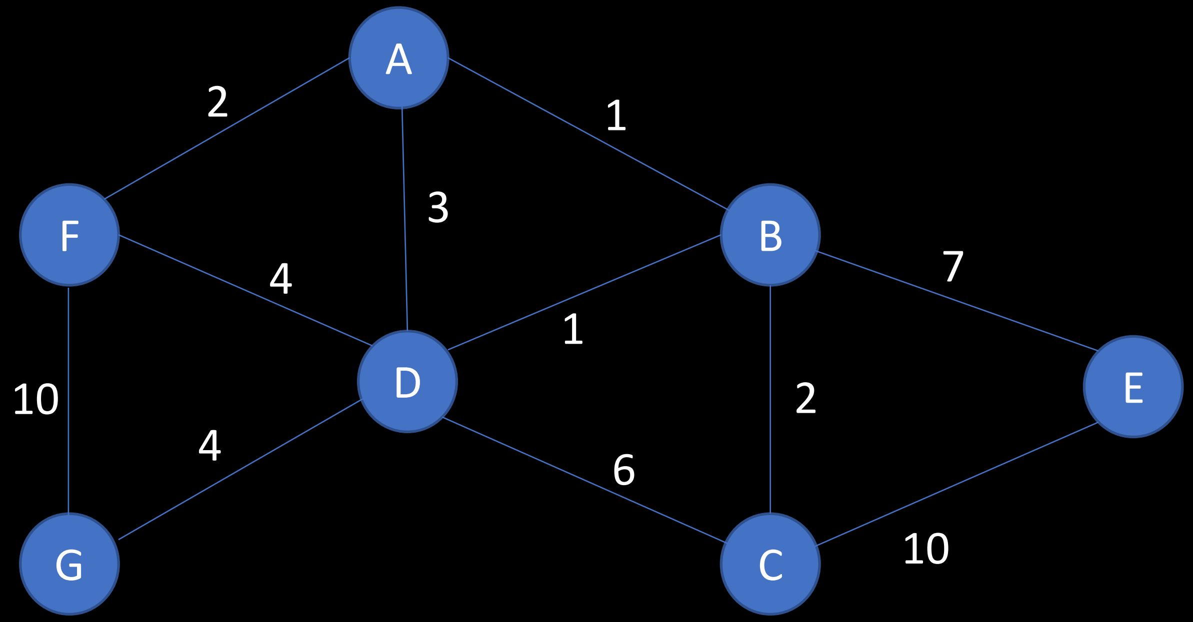
Vertex: <Neighbors of Vertex (Edges pointing from a vertex to the neighbor), edge weight>

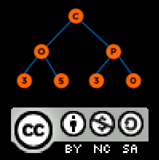
A: <B, 1>, <D, 3>, <F, 2>
B: <A, 1>, <C, 2>, <D, 1>, <E, 7>
C: <B, 2>, <D, 6>, <E, 10>
D: <A, 3>, <B, 1>, <C, 6>, <F, 4>, <G, 4>
E: <B, 7>, <C, 10>
F: <A, 2>, <D, 4>, <G, 10>
G: <D, 4>, <F, 10>



Dijkstra with A as source

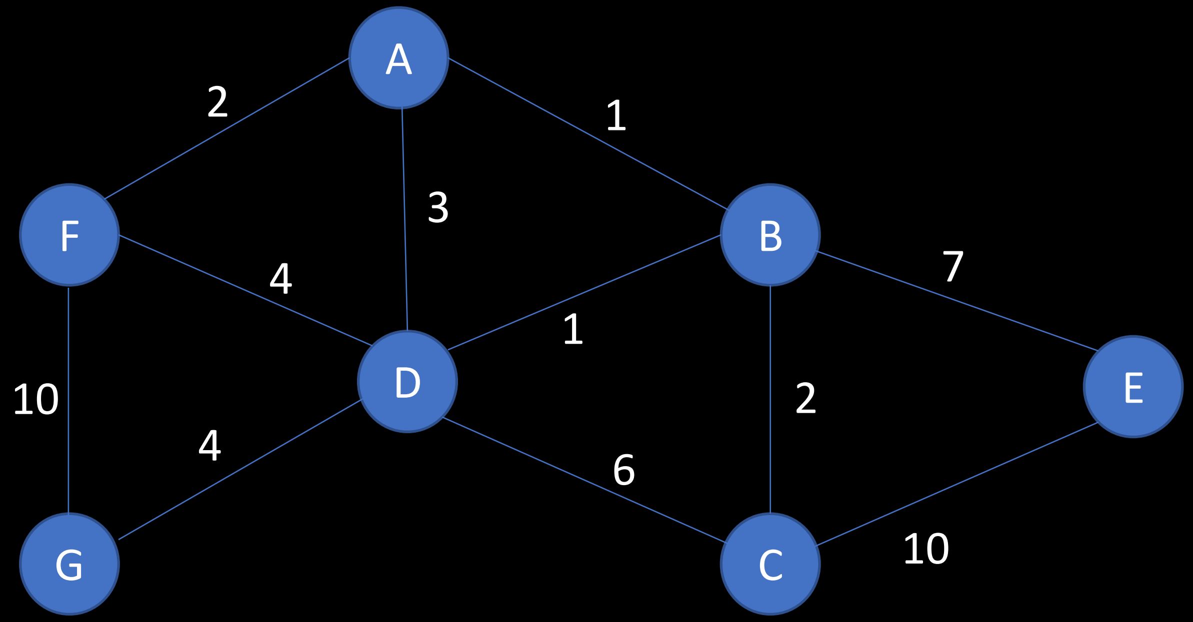
v	D(v)	P(v)
A		
B		
C		
D		
E		
F		
G		





Dijkstra with A as source

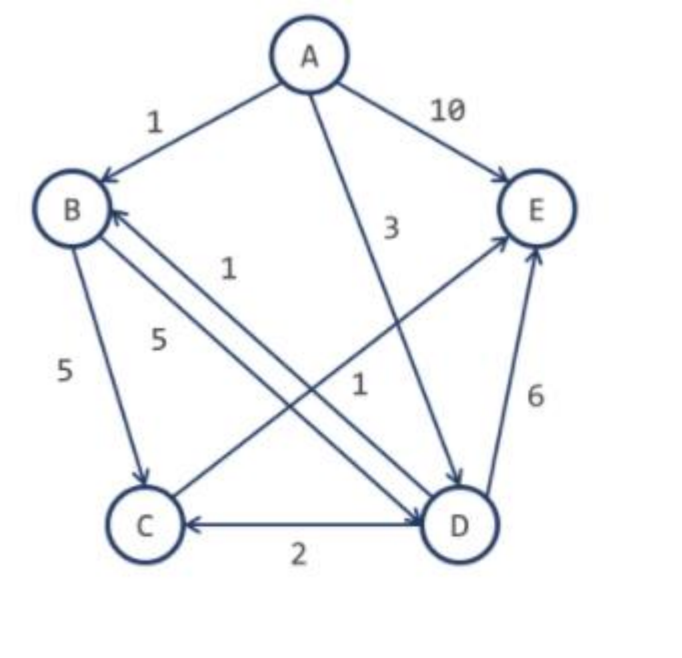
v	D(v)	P(v)
A	0	NA
B	1	A
C	3	B
D	2	B
E	8	B
F	2	A
G	6	D



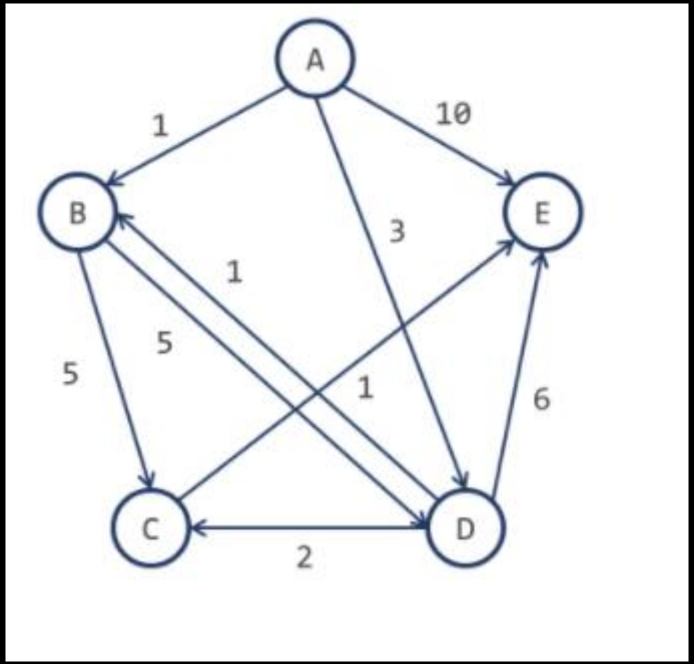
Alt Text for the Graph on Next Slide

Vertex: <Neighbors of Vertex (Edges pointing from a vertex to the neighbor), edge weight>

A: <B, 1>, <D, 3>, <E, 10>
B: <C, 5>, <D, 5>
C: <E, 1>
D: <B, 1>, <C, 2>, <E, 6>
E: -

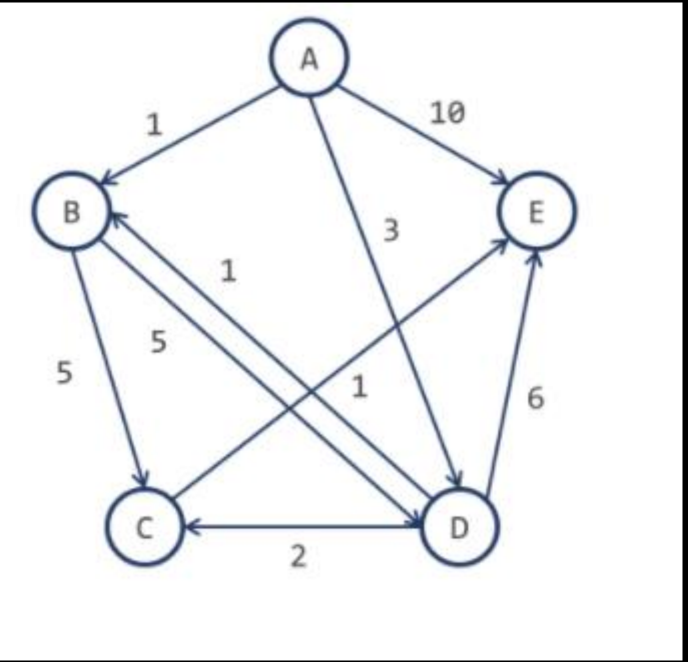


Dijkstra with A as source



V	d(V)	p(V)
B		
C		
D		
E		

Dijkstra with A as source



V	d(V)	p(V)
B	1	A
C	5	D
D	3	A
E	6	C

Questions