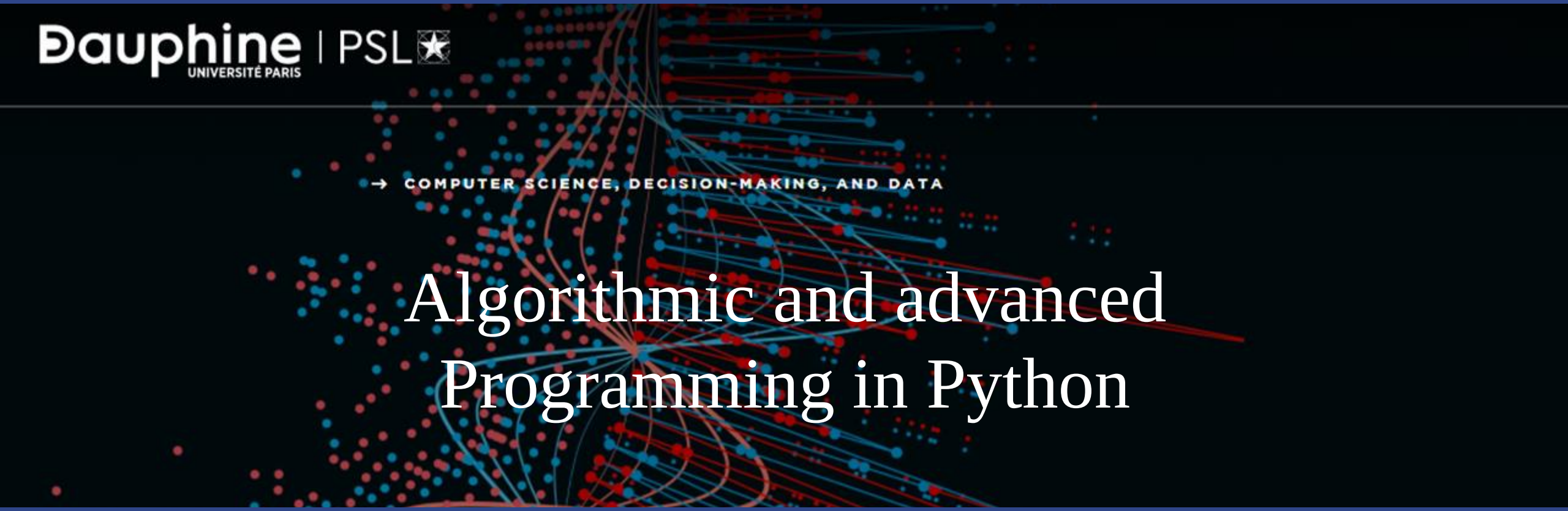


Algorithmic and advanced Programming in Python

Eric Benhamou eric.benhamou@dauphine.eu
Remy Belmonte remy.belmonte@dauphine.eu

Masterclass 7

Outline

1. Introduction to Graph Algorithms
2. Graph Representation
3. Graph Traversals
4. Topological Sort
5. Shortest Path Algorithms: Dystra

Reminder of the objective of this course

- People often learn about data structures out of context
- But in this course you will learn foundational concepts by building a real application with python and Flask

- To learn the ins and outs of the essential data structure, experiencing in practice has proved to be a much more powerful way to learn data structures

Reminder of previous session

- In Master class 6, we discuss about advanced binary trees
- Question: can you give the name of advanced binary trees and the main intuitions?

Introduction

- In the real world, many problems are represented in terms of objects and connections between them. For example, in an airline route map, we might be interested in questions like:
  - “What’s the fastest way to go from Paris to New York?”
  - “What is the cheapest way to go from Paris to New York?”
- To answer these questions we need information about connections (airline routes) between objects (towns). Graphs are data structures used for solving these kinds of problems.

Applications

- As part of this chapter, you will learn several ways to traverse graphs and how you can do useful things while traversing the graph in some order.

- We will also talk about shortest paths algorithms and see minimum spanning trees in LAB, which are used to plan road, telephone and computer networks and also find applications in clustering and approximate algorithms.

Glossary

- **Graph:** A graph G is simply a way of encoding pairwise relationships among a set of objects: it consists of a collection V of **nodes** and a collection E of **edges**, each of which "joins" two of the nodes.

- We thus represent an edge e in E as a two-element subset of V: e = {u, v} for some u, v in V, where we call u and v the ends of e.

- **Edges** in a graph indicate a symmetric relationship between their ends. Often we want to encode asymmetric relationships, and for this, we use the closely related notion of a **directed graph**.

Directed Graph

- A **directed** graph G’ consists of a set of nodes V and a set of directed edges E’. Each e’ in E’ is an ordered pair (u, v); in other words, the roles of u and v are not **interchangeable**, and we call u the tail of the edge and v the head. We will also say that edge e’ leaves node u and enters node v.

Undirected graph

- When we want to emphasize that the graph we are considering is **not directed**, we will call it an **undirected graph.**
- **By default, however, the term "graph" will mean an undirected graph**.

- It is also worth mentioning two warnings in our use of graph terminology.
  - First, although an edge e in an undirected graph should properly be written as a set of nodes {u, u}, one will more often see it written in the notation used for ordered pairs: e = (u, v).
  - Second, a node in a graph is also frequently called a **vertex**; in this context, **the two words have exactly the same meaning**.

Directed edge

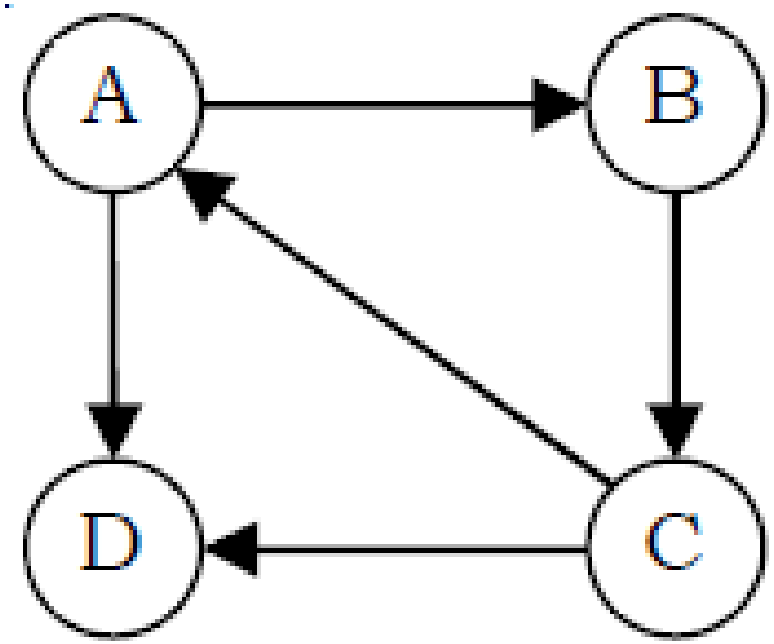
- Vertices and edges are positions and store elements
- Definitions that we use:
- Directed edge:
  - Ordered pair of vertices (u,v)
  - First vertex u is the origin
  - Second vertex v is the destination
  - Example: one-way road traffic

Undirected edge

- Unordered pair of vertices (u,v)
- Example: railway lines

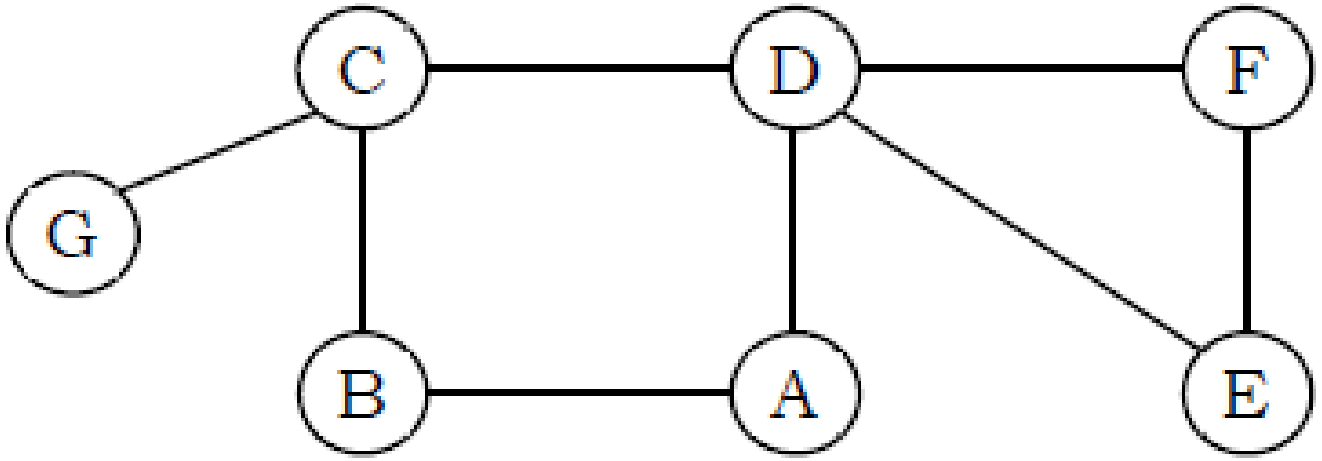
Directed graph

- All the edges are directed
- Example: route network



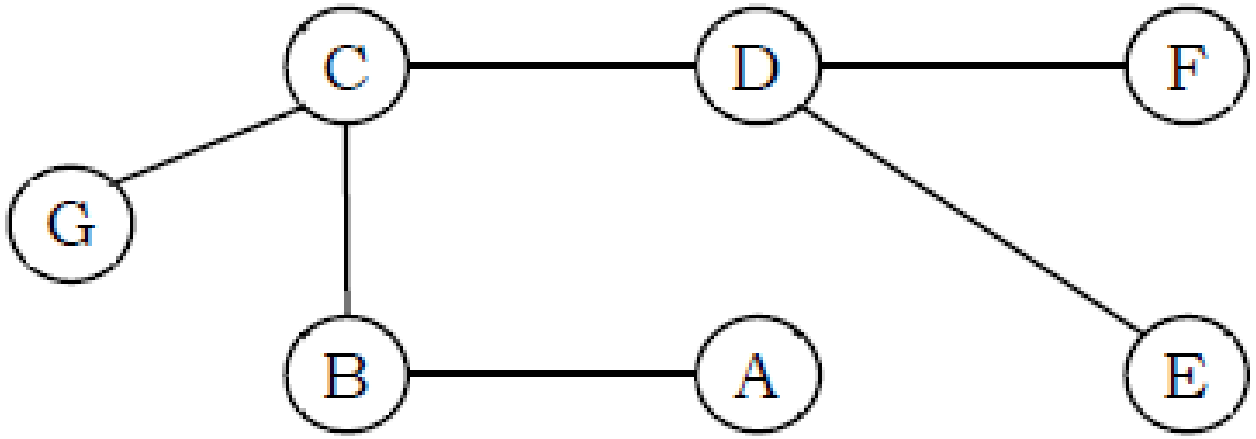
Undirected graph

- All the edges are undirected
- Example: flight network



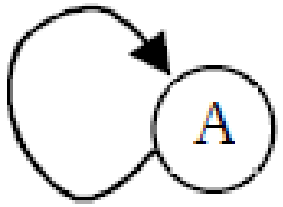
Connection with tree

- When an edge connects two vertices, the vertices are said to be **adjacent** to each other and the edge is incident on both vertices.
- **A graph with no cycles is called a tree.** *We will define properly a cycly in a few slides.* A tree is an acyclic connected graph.

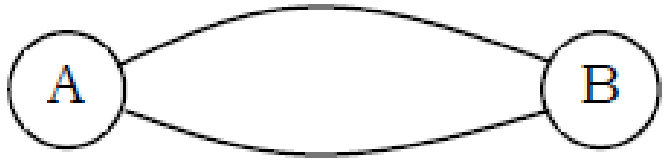


Other terms

- A self loop is an edge that connects a vertex to itself.



- Two edges are parallel if they connect the same pair of vertices.



- The degree of a vertex is the number of edges incident on it.
- A subgraph is a subset of a graph's edges (with associated vertices) that form a graph.

Path in undirected graph

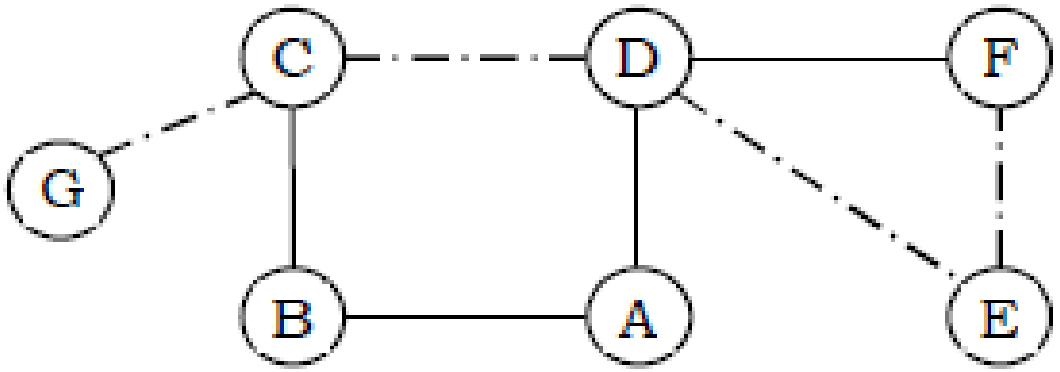
- One of the fundamental operations in a graph is that of traversing a sequence of nodes connected by edges.
- We define a **path** in an undirected graph G = (V, E) to be a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k$ with the property that each consecutive pair v_i, v_{i+1} is joined by an edge in G. P is often called a path from v_1, to v_k or a v_1 - v_k path.
- A path is called **simple if** all its vertices are distinct from one another.
- A **cycle** is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which k > 2 and the first *k-1* nodes are all distinct, and $v_1 = v_k$. In other words, the sequence of nodes "cycles back" to where it began.

Path in directed graph

- All of these definitions carry over naturally to **directed graphs**, with the following change: in a directed path or cycle, each pair of consecutive nodes has the property that (v_{i-1}, v_i) is an edge.
- In other words, the sequence of nodes in the path or cycle must respect the directionality of edges.

Visually

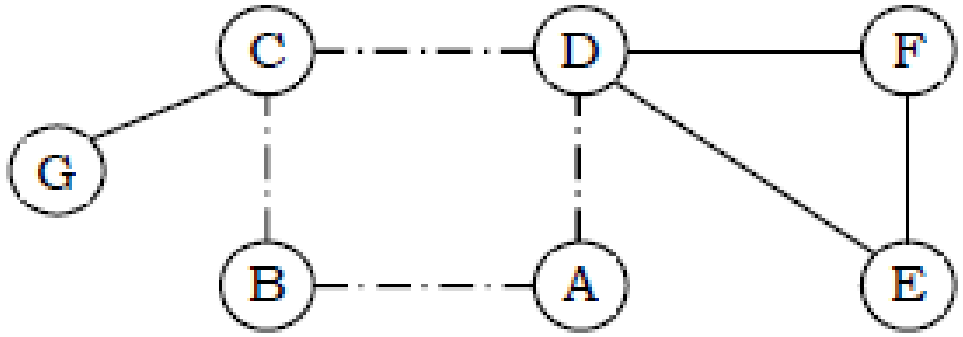
- A **path** in a graph is a sequence of adjacent vertices. Simple path is a path with no repeated vertices. In the graph below, the dotted lines represent a path from G to E.



- Question: can you give another path from G to E?

Cycle

- A cycle is a path where the first and last vertices are the same. A simple cycle is a cycle with no repeated vertices or edges (except the first and last vertices).



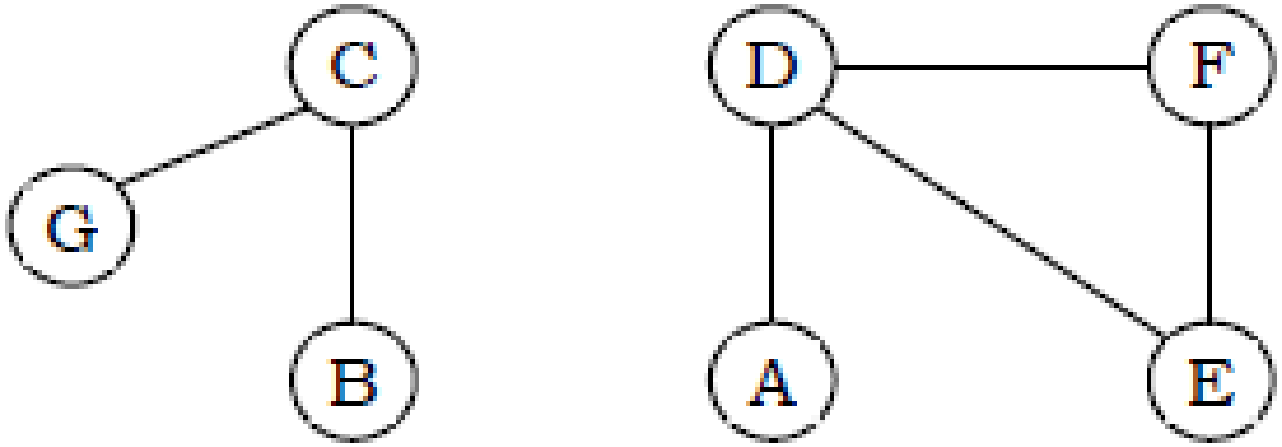
- Question: can you cite cycles in the graph above?

Connected graph

- We say that an undirected graph is **connected** if, for every pair of nodes u and v, there is a path from u to v. Choosing how to define connectivity of a **directed** graph is a bit more subtle, since it's possible for u to have a path to v while v has no path to u.
- We say that a directed graph is **strongly connected** if, for every two nodes u and v, there is a path from u to v and a path from v to u.

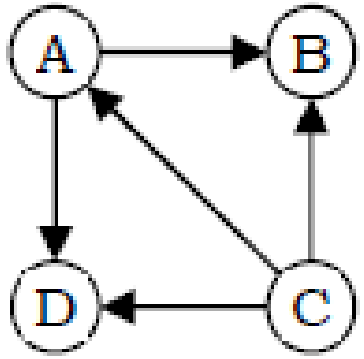
- We say that **one vertex is connected to another** if there is a path that contains both of them.
- A graph is connected if there is a path from *every* vertex to every other vertex.
- If a graph is not connected then it consists of a **set of connected components**.

Exemple

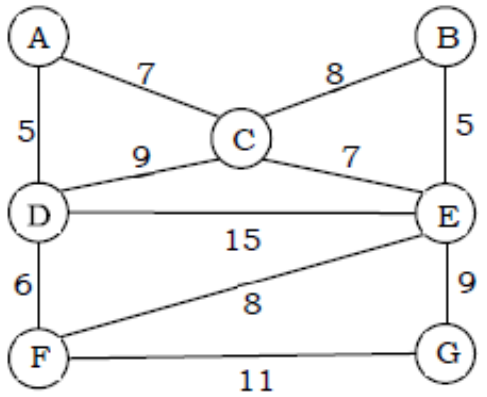


DAG

- A ***directed acyclic graph*** **[DAG]** is a directed graph with no cycles.



- In *weighted graphs* integers (*weights*) are assigned to each edge to represeant (distances or costs).

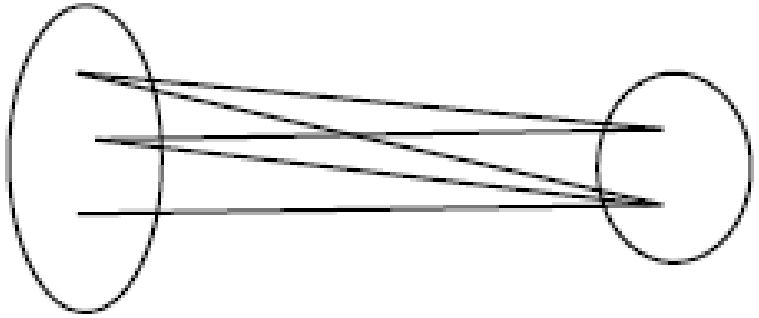


Short path

- In addition to simply knowing about the existence of a path between some pair of nodes u and v, we may also want to know whether there is a **short path**. Thus we define the distance between two nodes u and v to be **the minimum number of edges** in a u-v path.
- A forest is a disjoint set of trees.
- A spanning tree of a connected graph is a subgraph that contains all of that graph's vertices and is a single tree. A spanning forest of a graph is the union of spanning trees of its connected components.

Bipartite graph

- A bipartite graph is a graph whose vertices can be divided into two sets such that all edges connect a vertex in one set with a vertex in the other set.



Complete graphs

- Graphs with all edges present are called ***complete* graphs**.

- Graphs with relatively few edges (generally if it edges < $|V|$ log $|V|$) are called ***sparse graphs***.
- Graphs with relatively few of the possible edges missing are called ***dense* graphs.**
- Directed weighted graphs are sometimes called ***network***.
- We will denote the number of vertices in a given graph by $|V|$, and the number of edges by $|E|$. Note that E can range anywhere from 0 to $|V|$ ($|V|$ -1)/2, (in undirected graph). This is because each node can connect to every other node.

What are the applications of Graphs?

- Question: can you think of applications of Graphs?

What are the applications of Graphs?

- Representing relationships between components in electronic circuits
- Transportation networks: Highway network, Flight network
- Computer networks: Local area network, Internet, Web
- Databases: For representing ER (Entity Relationship) diagrams in databases, for representing dependency of tables in databases
- Machine learning (Graphical models)
- Causality representation
- Hidden relationship (Bayesian Graphs… like HMM, Kalman filter, etc..)

Graph Representation

- As in other ADTs, to manipulate graphs we need to represent them in some useful form. There are several ways to represent graphs, each with its advantages and disadvantages. Some situations, or algorithms that we want to run with graphs as input, call for one representation, and others call for a different representation.
- Here, we'll see three ways to represent graphs.
  - Adjacency Matrix
  - Adjacency List
  - Adjacency Set

Adjacency Matrix

- Graph Declaration for Adjacency Matrix: First, let us look at the components of the graph data structure. To represent graphs, we need the number of vertices, the number of edges and also their interconnections. So, the graph can be declared as:

Corresponding Code

```python
class Vertex:
    def __init__(self, node):
        self.id = node
        # Mark all nodes unvisited
        self.visited = False

    def addNeighbor(self, neighbor, G):
        G.addEdge(self.id, neighbor)

    def getConnections(self, G):
        return G.adjMatrix[self.id]

    def getVertexID(self):
        return self.id

    def setVertexID(self, id):
        self.id = id

    def setVisited(self):
        self.visited = True

    def __str__(self):
        return str(self.id)
```

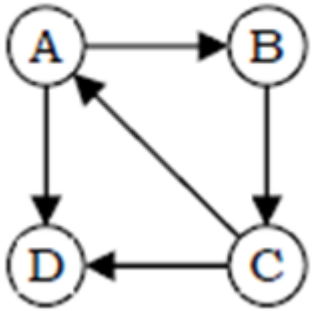
```python
class Graph:
    def __init__(self, numVertices, cost=0):
        self.adjMatrix = [[-1] * numVertices for _ in range(numVertices)]
        self.numVertices = numVertices
        self.vertices = []
        for i in range(numVertices):
            newVertex = Vertex(i)
            self.vertices.append(newVertex)
```

Description

- The adjacency matrix of a graph is a square matrix of size $V \times V$. The V is the number of vertices of the graph G. The values of matrix are boolean. Let us assume the matrix is $adjMatrix$. The value $adjMatrix[u, v]$ is set to 1 if there is an edge from vertex u to vertex v and 0 otherwise. In the matrix, each edge is represented by two bits for undirected graphs. That means, an edge from u to v is represented by 1 value in both $adjMatrix$[u, v] and $adjMatrix[u, v]$. To save time, we can process only half of this symmetric matrix. Also, we can assume that there is an "edge" from each vertex to itself. So, $adjMatrix$[u, u] is set to 1 for all vertices.

Example

- If the graph is a directed graph then we need to mark only one entry in the adjacency matrix. As an example, consider the directed graph below.



- The adjacency matrix for this graph can be given as:

	A	B	C	D
A	0	1	0	1
B	0	0	1	0
C	1.41	0	0	1
D	0	0	0	0

Implementation

- Now, let us concentrate on the implementation. To read a graph, one way is to first read the vertex names and then read pairs of vertex names (edges). The code below reads an undirected graph.

Code

```python
class Vertex:
    def __init__(self, node):
        self.id = node
        # Mark all nodes unvisited
        self.visited = False

    def addNeighbor(self, neighbor, G):
        G.addEdge(self.id, neighbor)

    def getConnections(self, G):
        return G.adjMatrix[self.id]

    def getVertexID(self):
        return self.id

    def setVertexID(self, id):
        self.id = id

    def setVisited(self):
        self.visited = True

    def __str__(self):
        return str(self.id)
```

Code follows

```
class Graph:
    def __init__(self, numVertices, directed=False):
        self.adjMatrix = [[0] * numVertices for _ in range(numVertices)]
        self.numVertices = numVertices
        self.vertices = []
        for i in range(numVertices):
            newVertex = Vertex(i)
            self.vertices.append(newVertex)
        self.directed = directed
        self.vtx = 0

    def addVertex(self, id):
        if self.vtx < self.numVertices:
            self.vertices[self.vtx].setVertexID(id)
            self.vtx += 1

    def getVertex(self, n):
        for vertxin in range(0, self.numVertices):
            if n == self.vertices[vertxin].getVertexID():
                return vertxin
        return -1

    def addEdge(self, frm, to, cost=0):
        if self.getVertex(frm) != -1 and self.getVertex(to) != -1:
            self.adjMatrix[self.getVertex(frm)][self.getVertex(to)] = cost
            if not self.directed:
                self.adjMatrix[self.getVertex(to)][self.getVertex(frm)] = cost

    def getVertices(self):
```

Code follows

```
def getVertices(self):
    vertices = []
    for vertxin in range(0, self.numVertices):
        vertices.append(self.vertices[vertxin].getVertexID())
    return vertices

def printMatrix(self):
    for u in range(0, self.numVertices):
        row = []
        for v in range(0, self.numVertices):
            row.append(self.adjMatrix[u][v])
        print('row', u, ':', row)

def getEdges(self):
    edges = []
    for u in range(0, self.numVertices):
        for v in range(0, self.numVertices):
            if self.adjMatrix[u][v] != -1:
                uid = self.vertices[u].getVertexID()
                vid = self.vertices[v].getVertexID()
                edges.append((uid, vid, self.adjMatrix[u][v]))
    return edges
```

Code follows

```python
if __name__ == '__main__':
    for directed in [False, True]:
        G = Graph(4, directed=directed)
        G.addVertex('a')
        G.addVertex('b')
        G.addVertex('c')
        G.addVertex('d')
        G.addEdge('a', 'b', 1)
        G.addEdge('a', 'd', 1)
        G.addEdge('b', 'c', 1)
        G.addEdge('c', 'a', round(2**0.5,2))
        G.addEdge('c', 'd', 1)
        print('Data for ' + ('Directed' if directed else 'Undirected') + ' Graph:')
        print(G.getEdges())
        print(' ')
        print('Matrix ')
        G.printMatrix()
        print(' ')
```

- Question: What is the output?

Output

```
Data for Undirected Graph:
[('a', 'a', 0), ('a', 'b', 1), ('a', 'c', 1.41), ('a', 'd', 1), ('b', 'a',
1), ('b', 'b', 0), ('b', 'c', 1), ('b', 'd', 0), ('c', 'a', 1.41), ('c',
'b', 1), ('c', 'c', 0), ('c', 'd', 1), ('d', 'a', 1), ('d', 'b', 0), ('d',
'c', 1), ('d', 'd', 0)]

Matrix
row 0 : [0, 1, 1.41, 1]
row 1 : [1, 0, 1, 0]
row 2 : [1.41, 1, 0, 1]
row 3 : [1, 0, 1, 0]

Data for Directed Graph:
[('a', 'a', 0), ('a', 'b', 1), ('a', 'c', 0), ('a', 'd', 1), ('b', 'a', 0),
('b', 'b', 0), ('b', 'c', 1), ('b', 'd', 0), ('c', 'a', 1.41), ('c', 'b',
0), ('c', 'c', 0), ('c', 'd', 1), ('d', 'a', 0), ('d', 'b', 0), ('d', 'c',
0), ('d', 'd', 0)]

Matrix
row 0 : [0, 1, 0, 1]
row 1 : [0, 0, 1, 0]
row 2 : [1.41, 0, 0, 1]
row 3 : [0, 0, 0, 0]
```

When to choose adjacency matrix representation?

- Question: when is the adjacency matrix representation good?

When to choose adjacency matrix representation?

- Question: when is the adjacency matrix representation good?
- The adjacency matrix representation is good if the graphs are dense. The matrix requires $O(V^2)$ bits of storage and $O(V^2)$ time for initialization.
- If the number of edges is proportional to V^2, then there is no problem because V^2 steps are required to read the edges.
- If the graph is sparse, the initialization of the matrix dominates the running time of the algorithm as it takes takes $O(V^2)$.

Downside

- The downsides of adjacency matrices are that enumerating the outgoing edges from a vertex takes O(n) time even if there aren't very many, and the O(V^2) space cost is high for sparse graphs, those with much fewer than V^2 edges. The adjacency matrix representation takes O(V^2) amount of space while it is computed. When graph has maximum number of edges or minimum number of edges, in both cases the required space will be same.

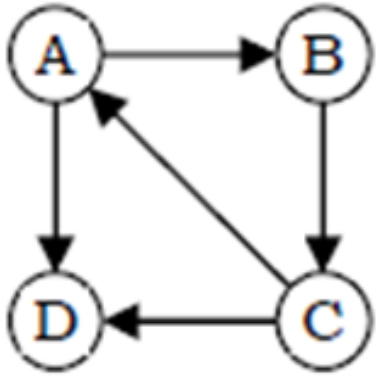
- Question: which other data structure could we use to represent graphs?

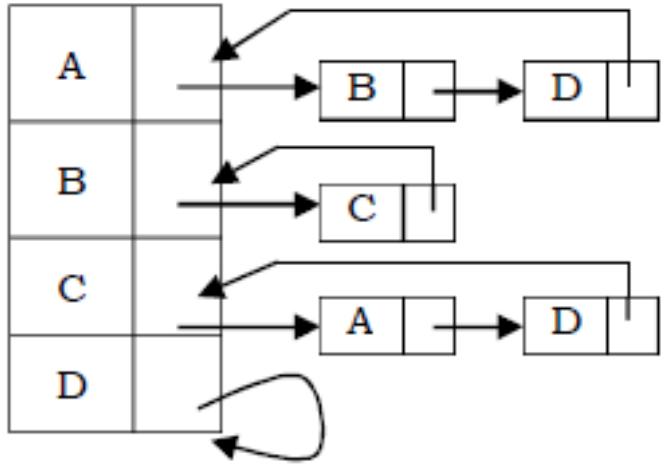
Adjacency List

- Graph Declaration for Adjacency List
- In this representation all the vertices connected to a vertex v are listed on an adjacency list for that vertex v. This can be easily implemented with linked lists. That means, for each vertex v we use a linked list and list nodes represents the connections between v and other vertices to which v has an edge.

Description

- Considering the same example as that of the adjacency matrix, the adjacency list representation can be given as:





- Since vertex A has an edge for B and D, we have added them in the adjacency list for A. The same is the case with other vertices as well.

Corresponding code

```python
class Vertex:
    def __init__(self, node):
        self.id = node
        self.adjacent = {}
        # Set distance to infinity for all nodes
        self.distance = float('inf')
        # Mark all nodes unvisited
        self.visited = False
        # Predecessor
        self.previous = None

    def addNeighbor(self, neighbor, weight=0):
        self.adjacent[neighbor] = weight

    def getConnections(self):
        return self.adjacent.keys()

    def getVertexID(self):
        return self.id

    def getWeight(self, neighbor):
        return self.adjacent[neighbor]

    def setDistance(self, dist):
        self.distance = dist
```

Code (follows)

```
    def getDistance(self):
        return self.distance

    def setPrevious(self, prev):
        self.previous = prev

    def setVisited(self):
        self.visited = True

    def __str__(self):
        return str(self.id) + ' adjacent: ' + str([x.id for x in self.adjacent])
```

Code (follows)

```
class Graph:
    def __init__(self, directed=False):
        self.vertDictionary = {}
        self.numVertices = 0
        self.directed = directed

    def __iter__(self):
        return iter(self.vertDictionary.values())

    def addVertex(self, node):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(node)
        self.vertDictionary[node] = newVertex
        return newVertex

    def getVertex(self, n):
        if n in self.vertDictionary:
            return self.vertDictionary[n]
        else:
            return None
```

Code (follows)

```
    def addEdge(self, frm, to, cost=0):
        if frm not in self.vertDictionary:
            self.addVertex(frm)
        if to not in self.vertDictionary:
            self.addVertex(to)

        self.vertDictionary[frm].addNeighbor(self.vertDictionary[to], cost)
        # For undirected graph add the opposite edge
        if not self.directed:
            self.vertDictionary[to].addNeighbor(self.vertDictionary[frm], cost)

    def getVertices(self):
        return self.vertDictionary.keys()

    def setPrevious(self, current):
        self.previous = current

    def getPrevious(self, current):
        return self.previous

    def getEdges(self):
        edges = []
        for v in G:
            for w in v.getConnections():
                vid = v.getVertexID()
                wid = w.getVertexID()
                edges.append((vid, wid, v.getWeight(w)))
        return edges
```

Code follows

```
if __name__ == '__main__':
    for directed in [False, True]:
        G = Graph(directed=directed)
        G.addVertex('a')
        G.addVertex('b')
        G.addVertex('c')
        G.addVertex('d')
        G.addEdge('a', 'b', 1)
        G.addEdge('a', 'd', 1)
        G.addEdge('b', 'c', 1)
        G.addEdge('c', 'a', 2**0.5)
        G.addEdge('c', 'd', 1)
        print('Data for ' + ('Directed' if directed else 'Undirected') + ' Graph:')
        print(G.getEdges())
        print(' ')
```

- Question: What is the output?

Output

```
Data for Undirected Graph:
[('a', 'b', 1), ('a', 'd', 1), ('a', 'c', 1.4142135623730951),
('b', 'a', 1), ('b', 'c', 1), ('c', 'b', 1), ('c', 'a',
1.4142135623730951), ('c', 'd', 1), ('d', 'a', 1), ('d', 'c', 1)]

Data for Directed Graph:
[('a', 'b', 1), ('a', 'd', 1), ('b', 'c', 1), ('c', 'a',
1.4142135623730951), ('c', 'd', 1)]
```

Disadvantages of Adjacency Lists

- Question: what are the disadvantages of Adjacency Lists ?

Disadvantages of Adjacency Lists

- Question: what are the disadvantages of Adjacency Lists ?

- Using adjacency list representation we cannot perform some operations efficiently.
- As an example, consider the case of **deleting a node**. In adjacency list representation, it is not enough if we simply delete a node from the list representation. We also need to the node in all adjacency lists. For each node on the adjacency list of that node specifies another vertex. We need to search other nodes linked list also for deleting it. This problem can be solved by linking the two list nodes that correspond to a particular edge and making the adjacency lists doubly linked. But all these extra links are risky to process.

- Question: could you think of a data structure to address this issue?

Adjacency Set, Adjancy Map

- It is very much similar to adjacency list but instead of using Linked lists, Disjoint Sets are used.
- In line with adjacency list and sets representation, we can use use maps for storing the edges information of the graphs.

Comparison of Graph Representations

- Directed and undirected graphs are represented with the same structures. For directed graphs, everything is the same, except that each edge is represented just once. An edge from x to y is represented by a 1 value in $Adj[x][y]$ in the adjacency matrix, or by adding y on x $'s$ adjacency list. For weighted graphs, everything is the same, except fill the adjacency matrix with weights instead of boolean values.

Representation	Space	Checking edge between v and w?	Iterate over edges incident to v?
List of edges	E	E	E
Adj Matrix	V^2	1	V
Adj List	$E+V$	$Degree(v)$	$Degree(v)$
Adj Set	$E+V$	$log(Degree(v))$	$Degree(v)$

Graph Traversals

- To solve problems on graphs, we need a mechanism for traversing the graphs. Graph traversal algorithms are also called *graph search* algorithms. Like trees traversal algorithms (Inorder, Preorder, Postorder and Level-Order traversals), graph search algorithms can be thought of as starting at some source vertex in a graph and "searching" the graph by going through the edges and marking the vertices. Now, we will discuss two such algorithms for traversing the graphs.
  - Depth First Search [DFS]
  - Breadth First Search [BFS]

- Question: what about cycles?

Handling cycle?

- A graph can contain cycles, which may bring you to the same node again while traversing the graph. To avoid processing of same node again, use a boolean array which marks the node after it is processed. While visiting the nodes in the layer of a graph, store them in a manner such that you can traverse the corresponding child nodes in a similar order.

Depth First Search [DFS]

- Depth-first search (DFS) is a method for exploring a tree or graph. In a DFS, you go as deep as possible down one path before backing up and trying a different one.

- DFS algorithm works in a manner **similar to preorder traversal** of the trees.

- Question: which data structure should it use.?

Depth First Search [DFS] data structure

- Like preorder traversal, internally this algorithm also uses stack.

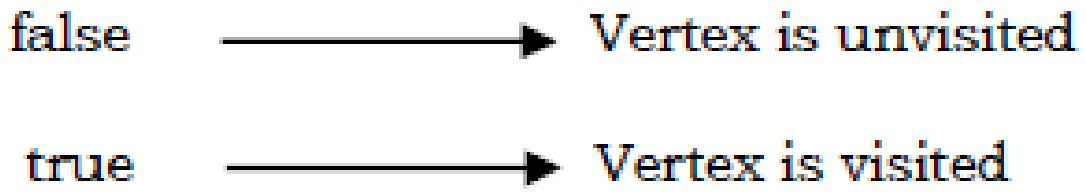
- Let us consider the following example. Suppose a person is trapped inside a maze. To come out from that maze, the person visits each path and each intersection (in the worst case). Let us say the person uses two colors of paint to mark the intersections already passed. When discovering a new intersection, it is marked grey, and he continues to go deeper.
- After reaching a “dead end” the person knows that there is no more unexplored path from the grey intersection, which now is completed, and he marks it with black. This “dead end” is either an intersection which has already been marked grey or black, or simply a path that does not lead to an intersection.

Intuition

- The intersections of the maze are the vertices and the paths between the intersections are the edges of the graph. The process of returning from the “dead end” is called *backtracking*. We are trying to go away from the starting vertex into the graph as deep as possible, until we have to backtrack to the preceding grey vertex. In DFS algorithm, we encounter the following types of edges.

DFS intuition

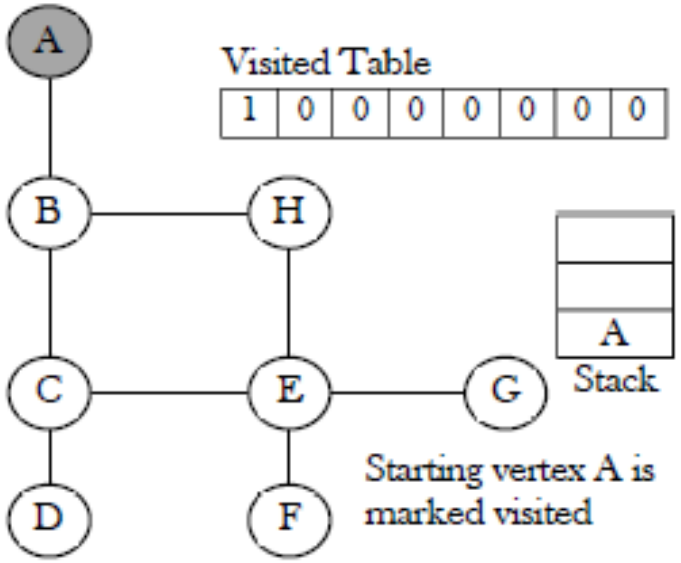
- For most algorithms Boolean classification, unvisited/visited is enough (for three color implementation refer to problems section). That means, for some problems we need to use three colors, but for our discussion two colors are enough.



DFS intuition

- Initially all vertices are marked unvisited (false). The DFS algorithm starts at a vertex u in the graph. By starting at vertex u it considers the edges from u to other vertices. If the edge leads to an already visited vertex, then backtrack to current vertex u. If an edge leads to an unvisited vertex, then go to that vertex and start processing from that vertex. That means the new vertex becomes the current vertex. Follow this process until we reach the dead-end. At this point start $backtracking$. The process terminates when backtracking leads back to the start vertex.

Illustration



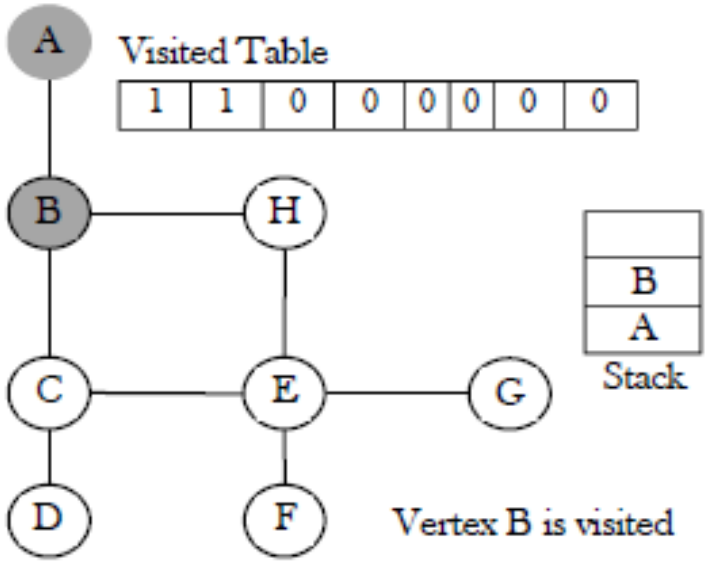
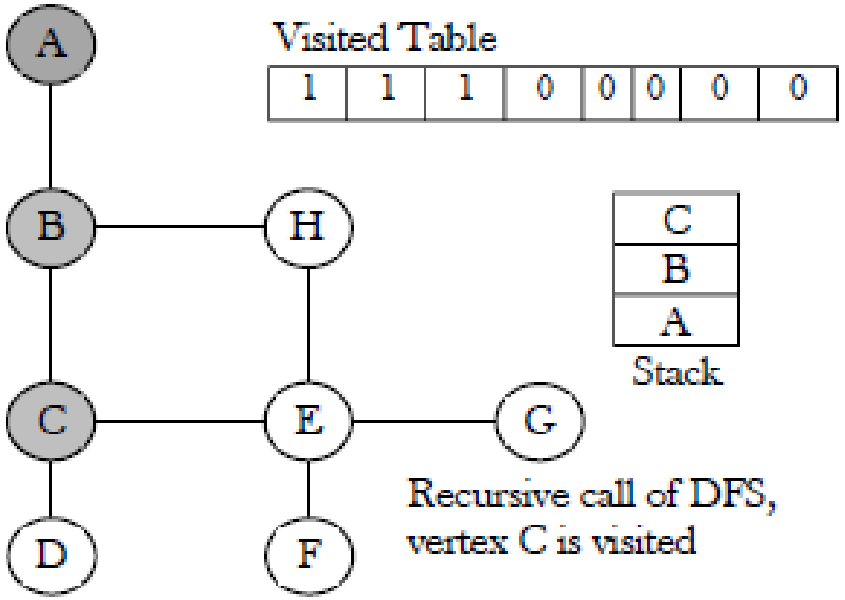


Illustration follows



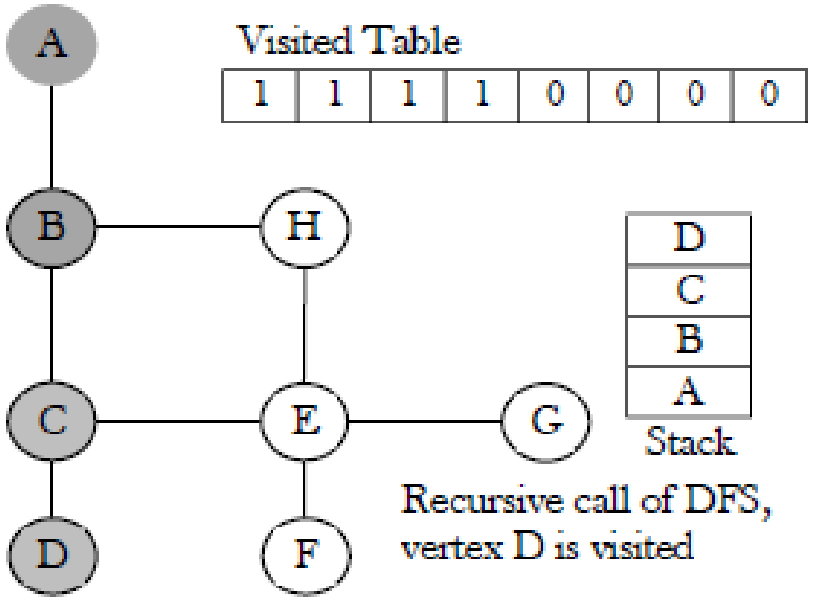
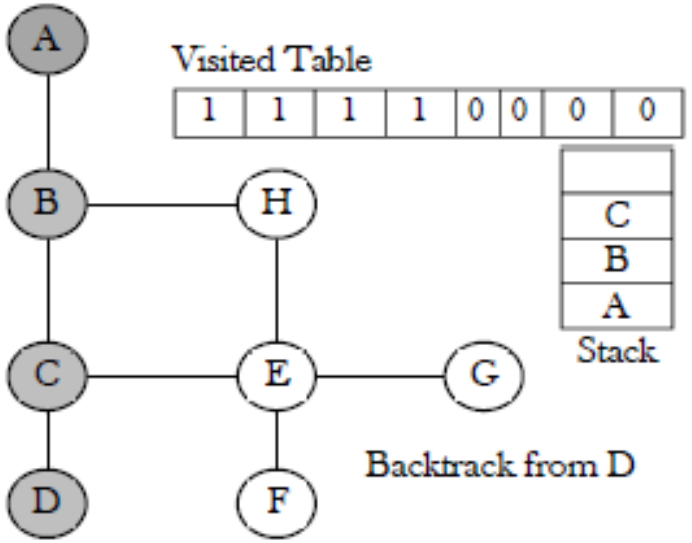


Illustration follows



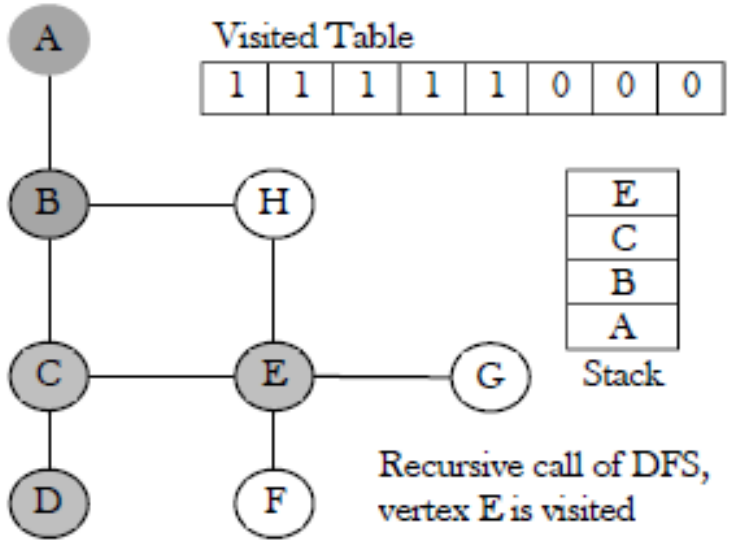
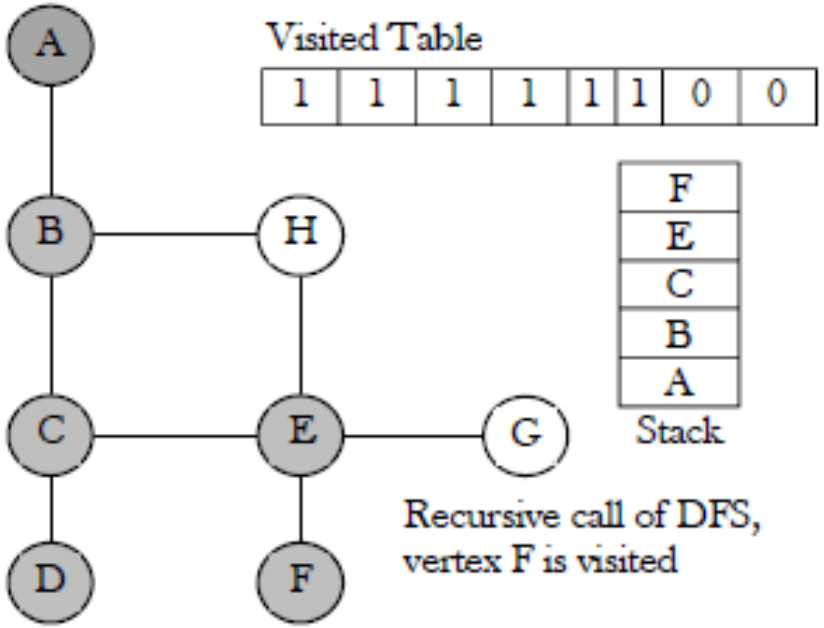


Illustration follows



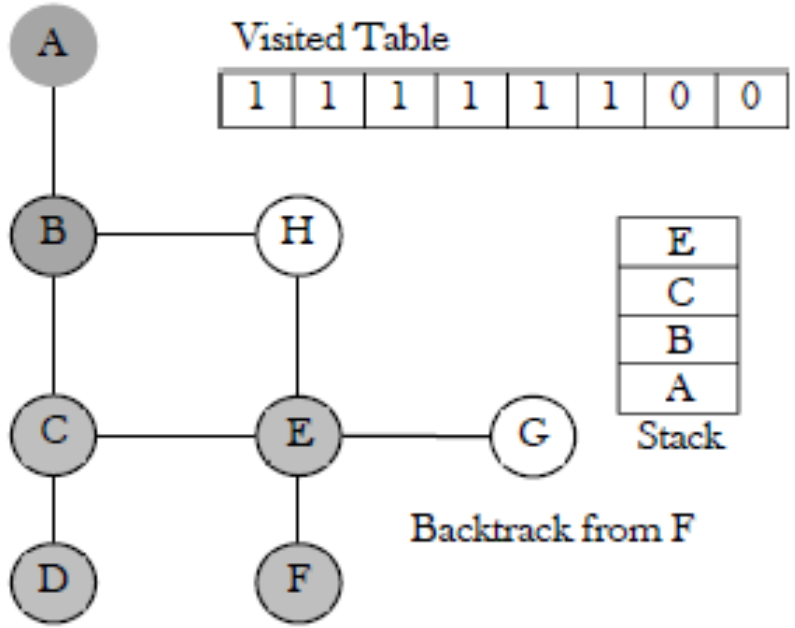


Illustration follows

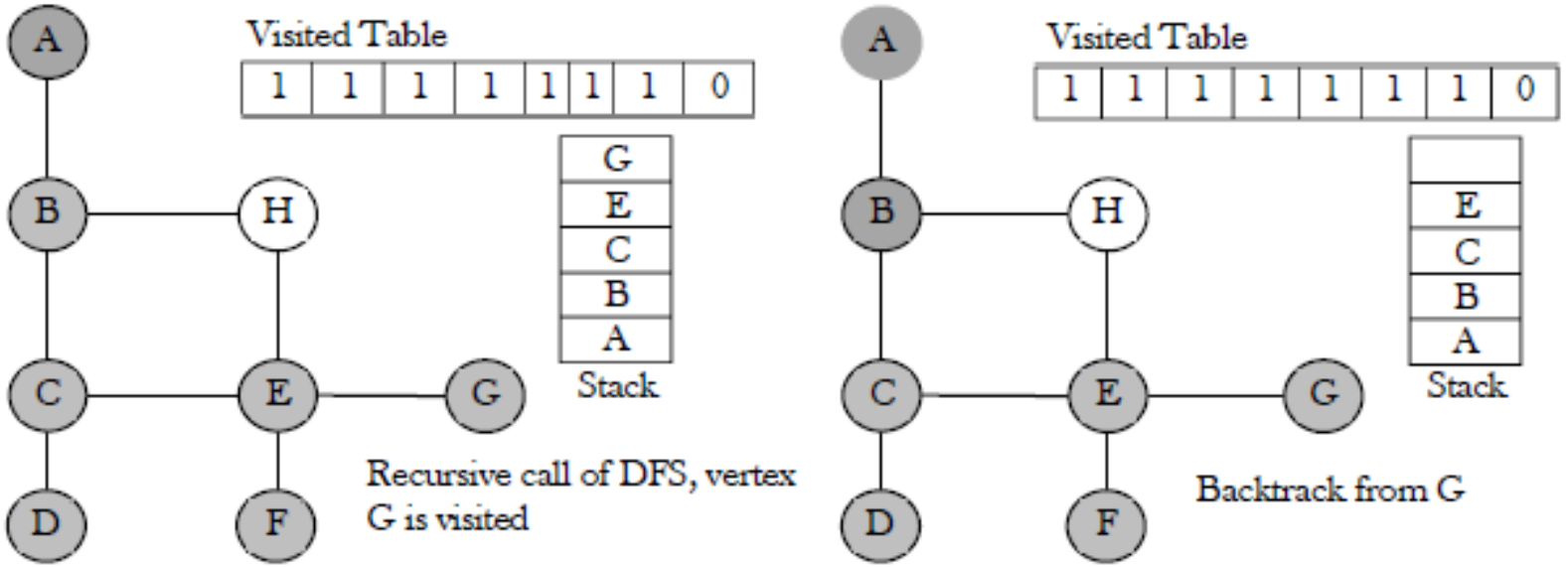


Illustration follows

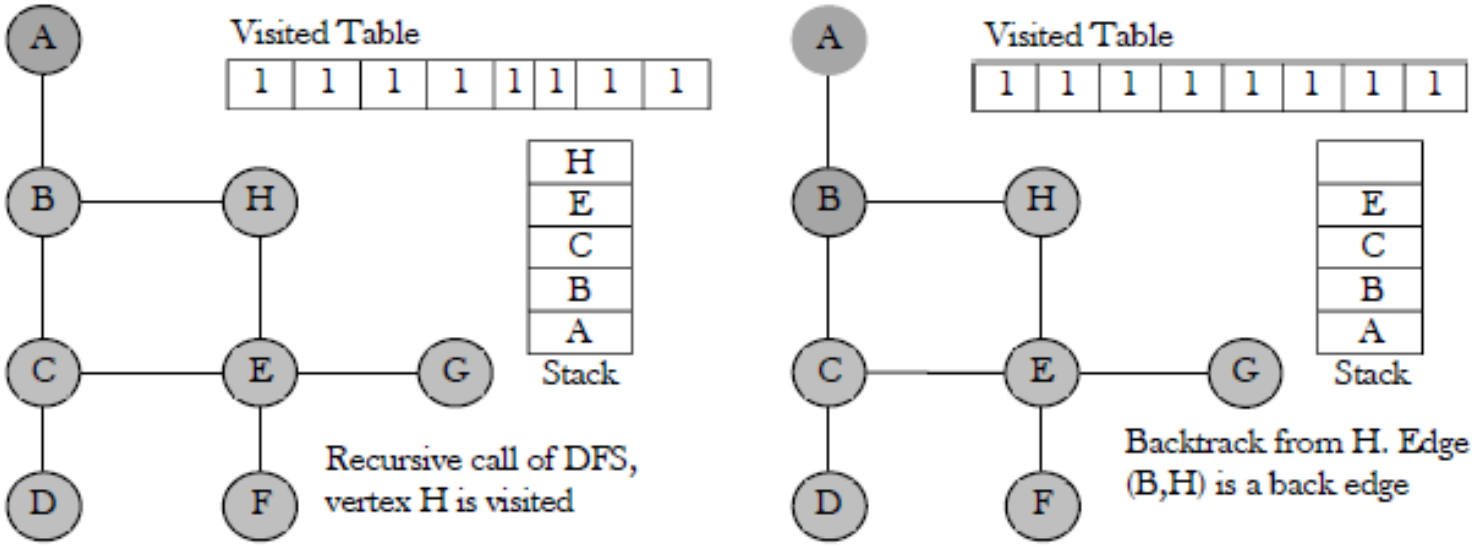
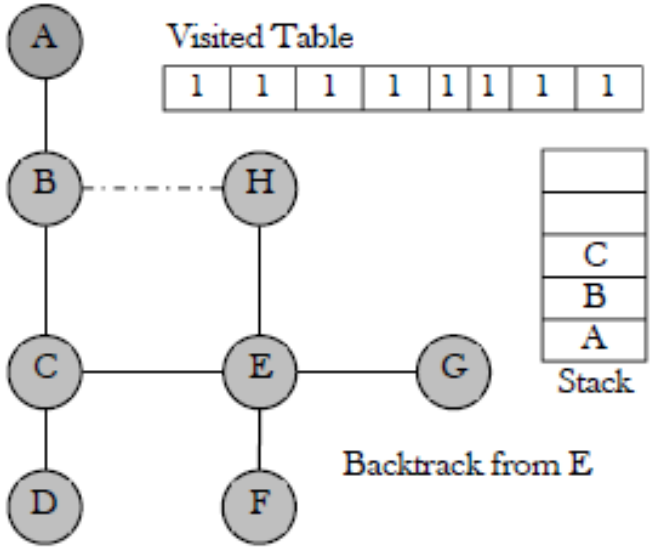


Illustration follows



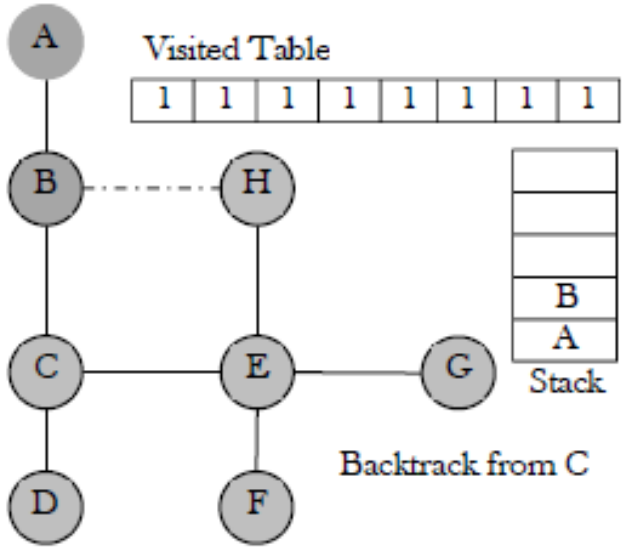
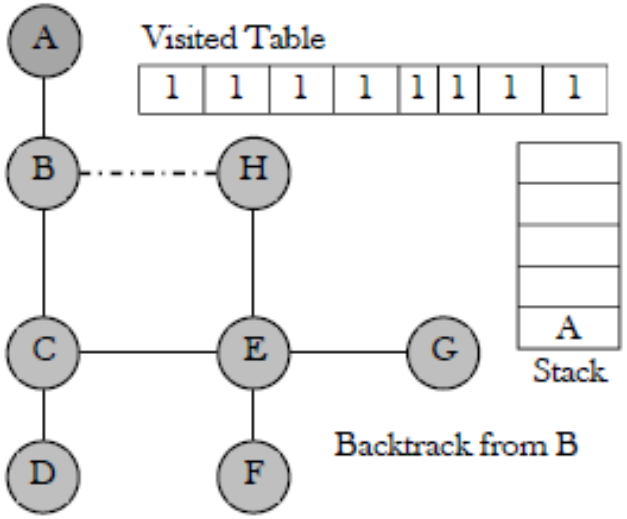
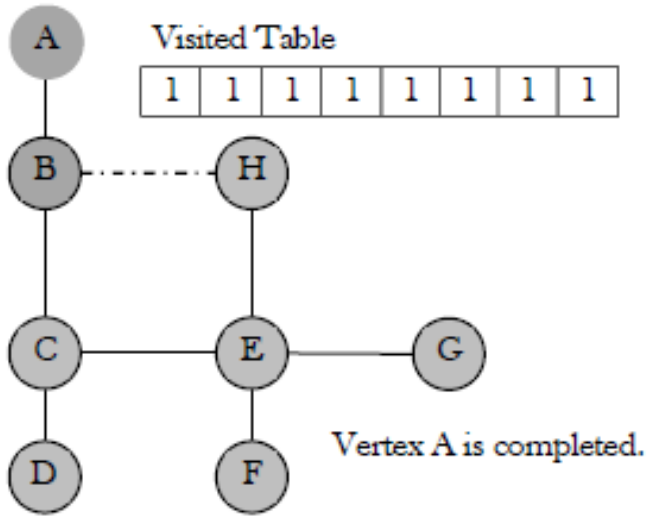


Illustration follows





Advantages/disadvantages?

- Question: any idea?

Advantages/disadvantages?

- **Advantages:**
  - Depth-first search on a binary tree generally requires less memory than breadth-first.
  - Depth-first search can be easily implemented with recursion.
- **Disadvantages:**
  - A DFS doesn't necessarily find the shortest path to a node, while breadth-first search does.

- Question: any idea of applications?

Applications of DFS

- Topological sorting
- Finding connected components
- Finding articulation points (cut vertices) of the graph
- Finding strongly connected components
- Solving puzzles such as mazes

Implementation

- See the LAB

Breadth First Search [BFS]

- Breadth-first search (BFS) is a method for exploring a tree or graph. In a BFS, you first explore all the nodes one step away, then all the nodes two steps away, etc. Breadth-first search is like throwing a stone in the center of a pond. The nodes you explore "ripple out" from the starting point.

- The BFS algorithm works similar to $level - order$ traversal of the trees. Like $level - order$ traversal, BFS also uses queues. In fact, $level - order$ traversal got inspired from BFS.

Breadth First Search [BFS] follows

- BFS works level by level. Initially, BFS starts at a given vertex, which is at level 0. In the first stage it visits all vertices at level 1 (that means, vertices whose distance is 1 from the start vertex of the graph). In the second stage, it visits all vertices at the second level. These new vertices are the ones which are adjacent to level 1 vertices. BFS continues this process until all the levels of the graph are completed.

- Question: any idea of the data structure?

Breadth First Search [BFS]

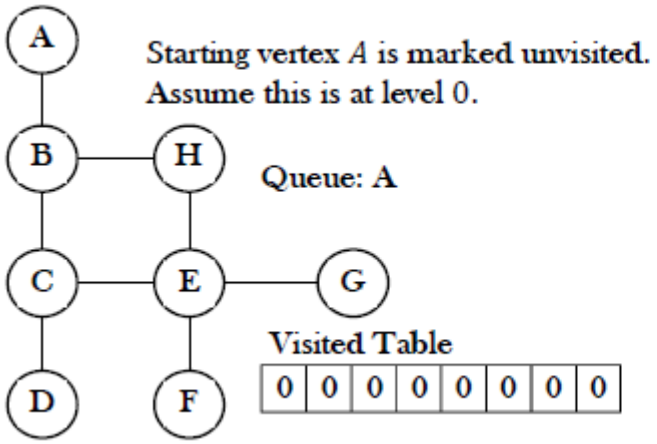
- To make this process easy, use a **queue** to store the node and mark it as 'visited' once all its neighbors (vertices that are directly connected to it) are marked or added to the queue.
- The queue follows the First In First Out (FIFO) queuing method, and therefore, the neighbors of the node will be visited in the order in which they were inserted in the node i.e. the node that was inserted first will be visited first, and so on.
- Generally *queue* data structure is used for storing the vertices of a level.

BFS rules

- BFS employs the following rules:
  - Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
  - If no adjacent vertex is found, remove the first vertex from the queue.
  - Repeat step 1 and step 2 until the queue is empty.

- As similar to DFS, assume that initially all vertices are marked $unvisited$ ($false$). Vertices that have been processed and removed from the queue are marked $visited$ ($true$). We use a queue to represent the visited set as it will keep the vertices in the order of when they were first visited. As an example, let us consider the same graph as that of the DFS example.

Illustration

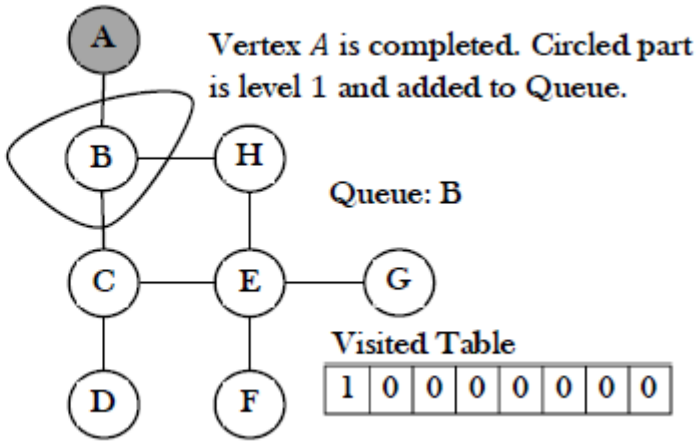


Starting vertex *A* is marked unvisited.
Assume this is at level 0.

Queue: A

Visited Table

0	0	0	0	0	0	0	0



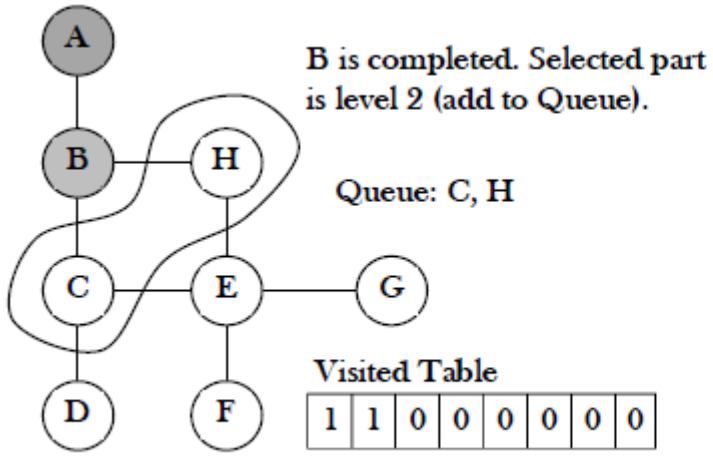
Vertex *A* is completed. Circled part
is level 1 and added to Queue.

Queue: B

Visited Table

1	0	0	0	0	0	0	0

Illustrations follows

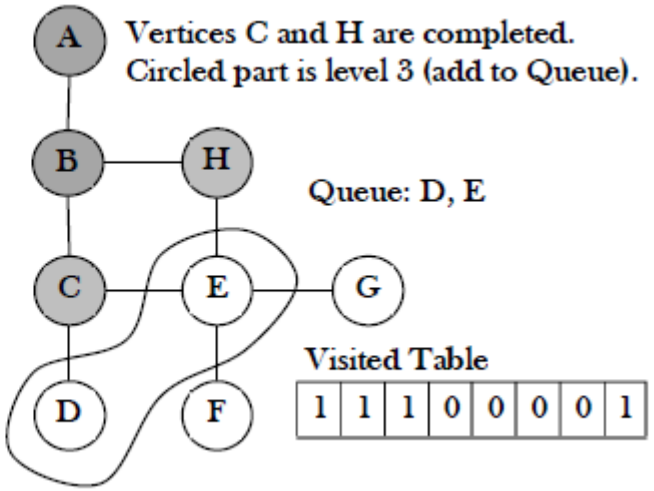


B is completed. Selected part is level 2 (add to Queue).

Queue: C, H

Visited Table

1	1	0	0	0	0	0	0



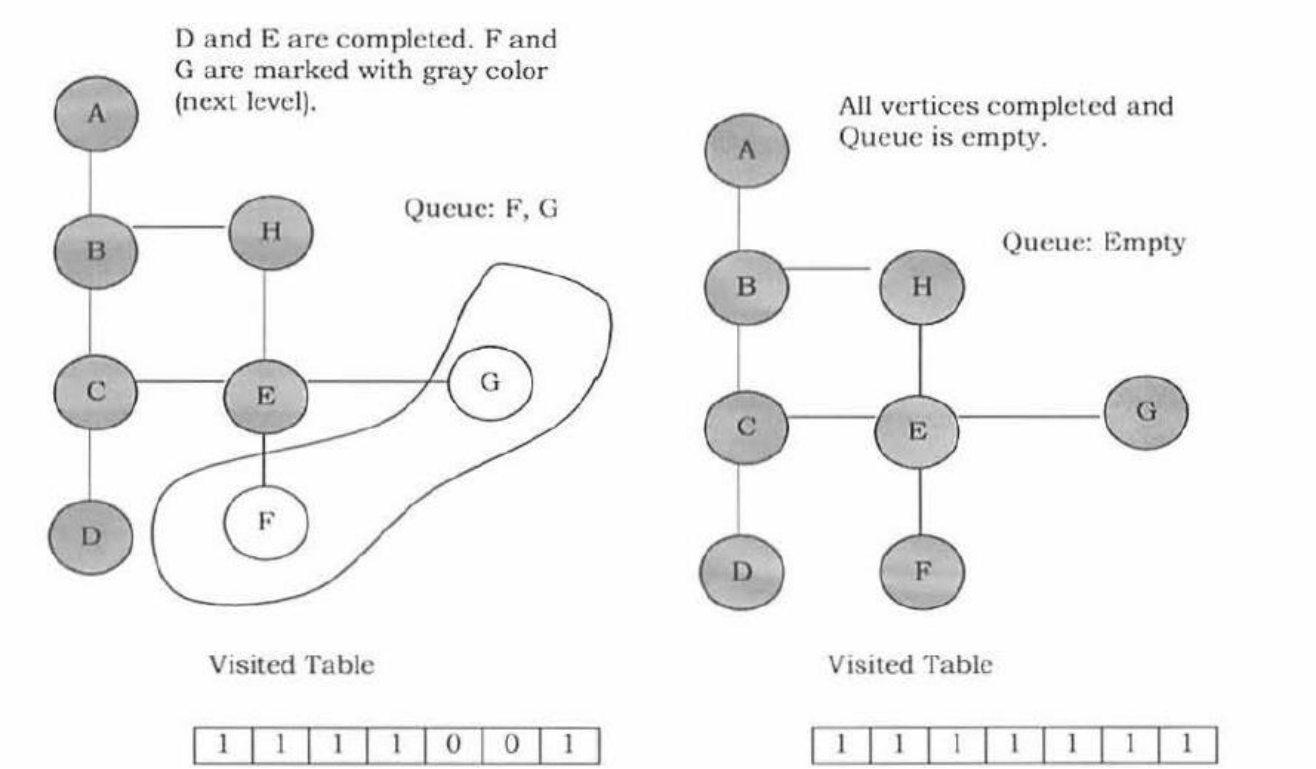
Vertices C and H are completed. Circled part is level 3 (add to Queue).

Queue: D, E

Visited Table

1	1	1	0	0	0	0	1

Illustrations follows



Advantages/Disadvantages

- **Advantages**: A BFS will find the shortest path between the starting point and any other reachable node. A depth-first search will not necessarily find the shortest path.
- **Disadvantages:** A BFS on a binary tree generally requires more memory than a DFS.

- **Applications of BFS:**
  - Finding all connected components in a graph
  - Finding all nodes within one connected component
  - Finding the shortest path between two nodes
  - Testing a graph for bipartiteness

Implementation

- See Lab

- **Question: any ideas of the difference between DFS and BFS?**

Comparing DFS and BFS

- Comparing BFS and DFS, the big advantage of **DFS is that it has much lower memory requirements than BFS** because it's not required to store all of the child pointers at each level. Depending on the data and what we are looking for, either DFS or BFS can be advantageous.
- For example, in a family tree if we are looking for someone who's still alive and if we assume that person would be at the bottom of the tree, then DFS is a better choice. BFS would take a very long time to reach that last level.
- The DFS algorithm finds the goal faster. Now, if we were looking for a family member who died a very long time ago, then that person would be closer to the top of the tree. In this case, BFS finds faster than DFS. So, the advantages of either vary depending on the data and what we are looking for.

Comparing DFS and BFS

- DFS is related to preorder traversal of a tree. Like *preorder* traversal, DFS visits each node before its children. The BFS algorithm **works similar to *level − order* traversal of the trees.**

- **Question: so what is better?**

Answer!

- **If someone asks whether DFS is better or BFS is better, the answer depends on the type of the problem that we are trying to solve.**
- BFS visits each level one at a time, and if we know the solution we are searching for is at a low depth, then BFS is good. DFS is a better choice if the solution is at maximum depth. The below table shows the differences between DFS and BFS in terms of their applications.

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	Yes	Yes
Shortest paths		Yes
Minimal use of memory space	Yes	

Topological Sort

- Assume that we need to schedule a series of tasks, such as classes or construction jobs, where we cannot start one task until after its prerequisites are completed. We wish to organize the tasks into a linear order that allows us to complete them one at a time without violating any prerequisites.
- We can model the problem using a DAG.

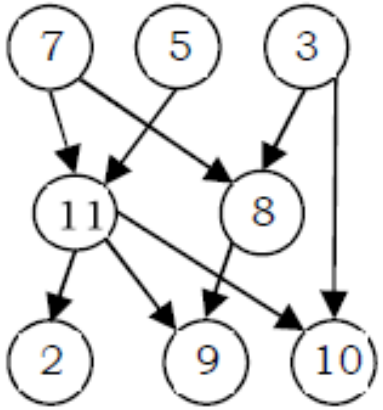
- The graph is directed because one task is a prerequisite of another -- the vertices have a directed relationship. It is acyclic because a cycle would indicate a conflicting series of prerequisites that could not be completed without violating at least one prerequisite. The process of laying out the vertices of a DAG in a linear order to meet the prerequisite rules is called a topological sort.

Topological sort

- *Topological sort* is an ordering of vertices in a directed acyclic graph [DAG] in which each node comes before all nodes to which it has outgoing edges. As an example, consider the course prerequisite structure at universities. A directed $edge$ (v, w) indicates that course v must be completed before course w. Topological ordering for this example is the sequence which does not violate the prerequisite requirement.
- Every DAG may have one or more topological orderings. Topological sort is not possible if the graph has a cycle, since for two vertices v and w on the cycle, v precedes w and w precedes v.

Topological sort property

- Topological sort has an interesting property. All pairs of consecutive vertices in the sorted order are connected by edges; then these edges form a directed **Hamiltonian path** [refer to LAB] in the DAG.
- If a Hamiltonian path exists, the topological sort order is unique. If a topological sort does not form a Hamiltonian path, DAG can have two or more topological orderings. In the graph below: 7, 5, 3, 11, 8, 2, 9, 10 and 3, 5, 7, 8, 11, 2, 9, 10 are both topological orderings.



Implemtation

- We can implement topological sort using a queue.
- First visit all edges, counting the number of edges that lead to each vertex (i.e., count the number of prerequisites for each vertex). Initially, *indegree* is computed for all vertices, starting with the vertices which are having indegree 0. That means consider the vertices which do not have any prerequisite. To keep track of vertices with indegree zero we can use a queue.

Implementation intuition

- All vertices with no prerequisites (indegree 0) are placed on the queue. We then begin processing the queue. While the queue is not empty, a vertex v is removed, and all edges adjacent to v have their indegrees decremented. A vertex is put on the queue as soon as its indegree falls to 0. The topological ordering is the order in which the vertices deQueue. If the queue becomes empty without printing all of the vertices, then the graph contains a cycle (i.e., there is no possible ordering for the tasks that does not violate some prerequisite).

Implementation intuition follows

- The first problem when attempting to create a topographic sort of a graph in any programming language is figuring out how to represent a graph. we can chose a map with an int as a key to simplify the implementation. Each vertex u is represented with a key in the map, each vertex that adjacent to u—v—is stored as a slice in the map referenced by the key u.

- To see in LAB

Applications of Topological Sorting

- Representing course prerequisites
- Detecting deadlocks
- Pipeline of computing jobs
- Checking for symbolic link loop
- Evaluating formulae in spreadsheet

Shortest Path Algorithms

- Shortest path algorithms are a family of algorithms designed to solve the shortest path problem. The shortest path problem is something most people have some intuitive familiarity with: given two points, A and B, what is the shortest path between them? Given a graph $G = (V, E)$ and a distinguished vertex s, we need to find the shortest path from s to every other vertex in G. There are variations in the shortest path algorithms which depends on the type of the input graph and are given below.

Variations of Shortest Path Algorithms

- If the edges have weights, the graph is called a weighted graph. Sometimes these edges are bidirectional and the graph is called undirected. Sometimes there can be even be cycles in the graph. Each of these subtle differences are what makes one algorithm work better than another for certain graph type.

- There are also different types of shortest path algorithms. Maybe you need to find the shortest path between point A and B, but maybe you need to shortest path between point A and all other points in the graph.

Shortest path in unweighted graph
Shortest path in weighted graph
Shortest path in weighted graph with negative edges

Applications of Shortest Path Algorithms

- Shortest path algorithms have many applications. As noted earlier, mapping software like Google or Apple maps makes use of shortest path algorithms. They are also important for road network, operations, and logistics research. Shortest path algorithms are also very important for computer networks, like the Internet.
- Any software that helps you choose a route uses some form of a shortest path algorithm. Google Maps, for instance, has you put in a starting point and an ending point and will solve the shortest path problem for you.
  - Finding fastest way to go from one place to another
  - Finding cheapest way to fly/send data from one city to another

Types of Shortest Path Algorithms

- *Single source shortest path problem*: In a Single Source Shortest Paths Problem, we are given a Graph G = (V, E), we want to find the shortest path from a given source vertex s ∈ V to every vertex v ∈ V.
- *Single destination shortest path problem*: Find the shortest path to a given destination vertex t from every vertex v. By shift the direction of each edge in the graph, we can shorten this problem to a single - source problem.

Types of Shortest Path Algorithms

- *Single pair shortest path problem*: Find the shortest path from u to v for given vertices u and v. If we determine the single - source problem with source vertex u, we clarify this problem also. Furthermore, no algorithms for this problem are known that run asymptotically faster than the best single - source algorithms in the worst case.
- *All pairs shortest path problem*: Find the shortest path from u to v for every pair of vertices u and v. Running a single - source algorithm once from each vertex can clarify this problem; but it can generally be solved faster, and its structure is of interest in the own right.

Comparison

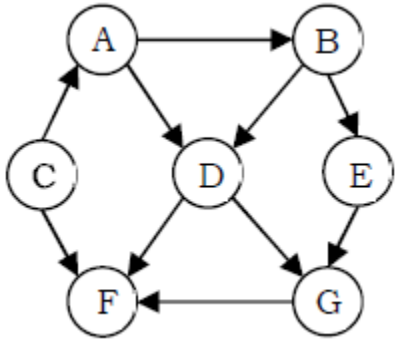
- *All pairs shortest path problem*: Find the shortest path from u to v for every pair of vertices u and v. Running a single - source algorithm once from each vertex can clarify this problem; but it can generally be solved faster, and its structure is of interest in the own right.

Shortest Path in Unweighted Graph

- Let *s* be the input vertex from which we want to find the shortest path to all other vertices. Unweighted graph is a special case of the weighted shortest-path problem, with all edges a weight of 1. The algorithm is similar to BFS and we need to use the following data structures:
- A distance table with three columns (each row corresponds to a vertex):
  - Distance from source vertex.
  - Path - contains the name of the vertex through which we get the shortest distance.
- A queue is used to implement breadth-first search. It contains vertices whose distance from the source node has been computed and their adjacent vertices are to be examined.

Example

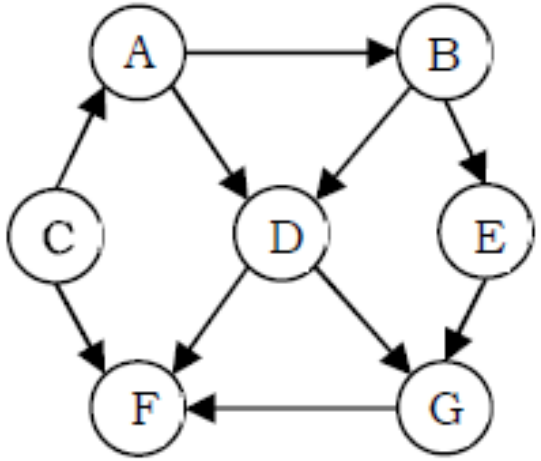
- As an example, consider the following graph and its adjacency list representation.



- Question: give the adjacency list?

Adjacency list

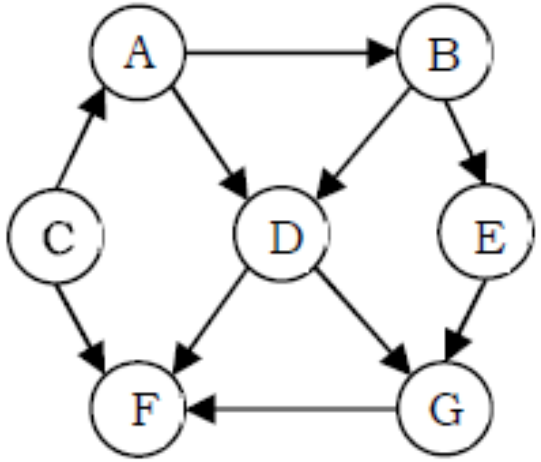
- The adjacency list for this graph is:
- $\boldsymbol{A}$: $B \rightarrow D$
- $\boldsymbol{B}$: $D \rightarrow E$
- $\boldsymbol{C}$: $A \rightarrow F$
- $\boldsymbol{D}$: $F \rightarrow G$
- $\boldsymbol{E}$: G
- $\boldsymbol{F}$: $-$
- $\boldsymbol{G}$: F



Intuition by example path from C

```
step 0
C  to  A --> -1
C  to  B --> -1
C  to  C --> 0
C  to  D --> -1
C  to  E --> -1
C  to  F --> -1
C  to  G --> -1
size= 1
C adjacent: ['A', 'F']
```

```
step 1
C  to  A --> 1
C  to  B --> -1
C  to  C --> 0
C  to  D --> -1
C  to  E --> -1
C  to  F --> 1
C  to  G --> -1
size= 2
A adjacent: ['B', 'D']
F adjacent: []
```



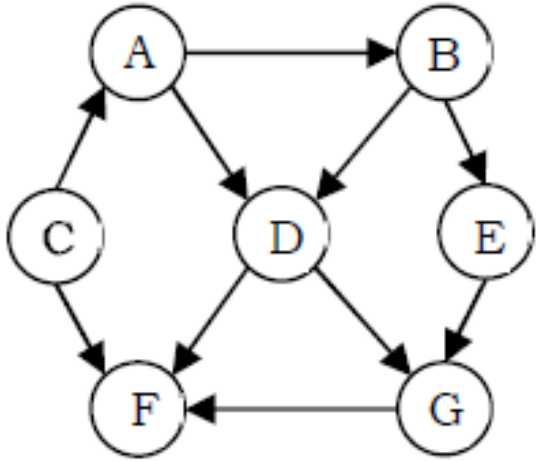

```
step 2
C  to  A --> 1
C  to  B --> 2
C  to  C --> 0
C  to  D --> 2
C  to  E --> -1
C  to  F --> 1
C  to  G --> -1
size= 3
F adjacent: []
B adjacent: ['D', 'E']
D adjacent: ['F', 'G']
```

```
step 3
C  to  A --> 1
C  to  B --> 2
C  to  C --> 0
C  to  D --> 2
C  to  E --> -1
C  to  F --> 1
C  to  G --> -1
size= 2
B adjacent: ['D', 'E']
D adjacent: ['F', 'G']
```

Example path from C

```
step 4
C  to  A --> 1
C  to  B --> 2
C  to  C --> 0
C  to  D --> 2
C  to  E --> 3
C  to  F --> 1
C  to  G --> -1
size= 2
D adjacent: ['F', 'G']
E adjacent: ['G']
```

```
step 5
C  to  A --> 1
C  to  B --> 2
C  to  C --> 0
C  to  D --> 2
C  to  E --> 3
C  to  F --> 1
C  to  G --> 3
size= 2
E adjacent: ['G']
G adjacent: ['F']
```




```
step 6
C  to  A --> 1
C  to  B --> 2
C  to  C --> 0
C  to  D --> 2
C  to  E --> 3
C  to  F --> 1
C  to  G --> 3
size= 1
G adjacent: ['F']
```

```
step 7
C  to  A --> 1
C  to  B --> 2
C  to  C --> 0
C  to  D --> 2
C  to  E --> 3
C  to  F --> 1
C  to  G --> 3
size= 0
```

Then iterate over adjacent node

```
def UnweightedShortestPath(G, s):
  source = G.getVertex(s)
  source.setDistance(0)
  source.setPrevious(None)
  vertQueue = Queue()
  vertQueue.enQueue(source)
  while (vertQueue.size > 0):
    currentVert = vertQueue.deQueue()
    for nbr in currentVert.getConnections():
      if nbr.getDistance() == -1:
        nbr.setDistance(currentVert.getDistance() + 1)
        nbr.setPrevious(currentVert)
        vertQueue.enQueue(nbr)
  for v in G.vertDictionary.values():
    print(source.getVertexID(), " to ", v.getVertexID(), "-->", v.getDistance())
```

Output

```
Graph data:
[('A', 'B', 0), ('A', 'D', 0), ('B', 'D', 0), ('B', 'E', 0), ('C', 'A', 0), ('C', 'F', 0), ('D', 'F',
0), ('D', 'G', 0), ('E', 'G', 0), ('G', 'F', 0)]
C  to  A --> 1
C  to  B --> 2
C  to  C --> 0
C  to  D --> 2
C  to  E --> 3
C  to  F --> 1
C  to  G --> 3
```

Dijkstra’s Algorithm

- A famous solution for the shortest path problem was developed by $Dijkstra$. $Dijkstra's$ algorithm is a generalization of the BFS algorithm. The regular BFS algorithm cannot solve the shortest path problem as it cannot guarantee that the vertex at the front of the queue is the vertex closest to source s.
- Dijkstra's algorithm makes use of breadth-first search (which is not a single source shortest path algorithm) to solve the single-source problem. It does place one constraint on the graph: there can be no negative weight edges. Dijkstra's algorithm is also sometimes used to solve the all-pairs shortest path problem by simply running it on all vertices in V. Again, this requires all edge weights to be positive.

Intuition

- Before going to code let us understand how the algorithm works. As in unweighted shortest path algorithm, here too we use the distance table. The algorithm works by keeping the shortest distance of vertex v from the source in the $Distance$ table. The value $Distance[v]$ holds the distance from s to v. The shortest distance of the source to itself is zero. The $Distance$ table for all other vertices is set to $math.MaxInt$ 64 to indicate that those vertices are not already processed.

Table

Vertex	Distance[v]	Previous vertex which gave Distance[v]
A	math.MaxInt64	-1
B	math.MaxInt64	-1
C	0	-
D	math.MaxInt64	-1
E	math.MaxInt64	-1
F	math.MaxInt64	-1
G	math.MaxInt64	-1

Algorithm

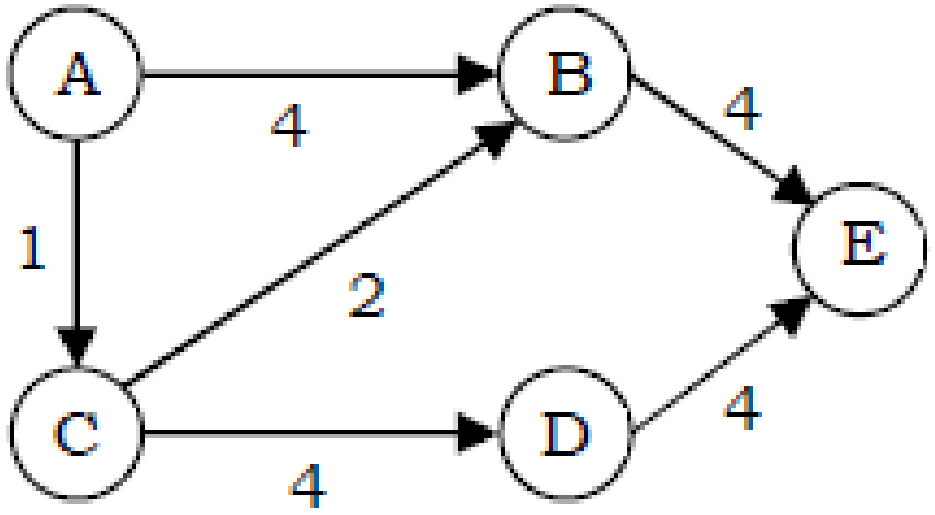
- After the algorithm finishes, the $Distance$ table will have the shortest distance from source s to each other vertex v.

- To simplify the understanding of $Dijkstra's$ algorithm, let us assume that the given vertices are maintained in two sets. Initially the first set contains only the source element and the second set contains all the remaining elements. After the k^{th} iteration, the first set contains k vertices which are closest to the source. These k vertices are the ones for which we have already computed the shortest distances from source.

Intuition with an example

- The $Dijkstra's$ algorithm can be better understood through an example, which will explain each step that is taken and how $Distance$ is calculated. The weighted graph below has 5 vertices from $A - E$.
- The value between the two vertices is known as the edge cost between two vertices. For example, the edge cost between A and C is 1. Dijkstra's algorithm can be used to find the shortest path from source A to the remaining vertices in the graph.

Initial graph



Dystra follows

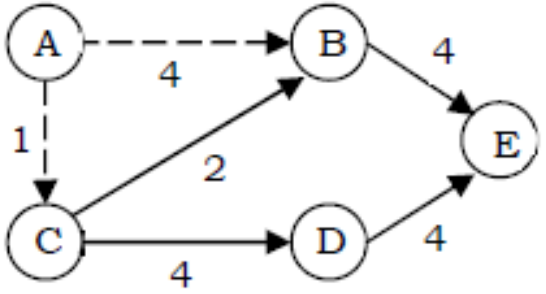
Initially the *Distance* table is:

Vertex	Distance[v]	Previous vertex which gave Distance[v]
A	0	-1
B	math.MaxInt64	-1
C	math.MaxInt64	-1
D	math.MaxInt64	-1
E	math.MaxInt64	-1

Dystra follows

After the first step, from vertex A, we can reach B and C. So, in the *Distance* table we update the reachability of B and C with their costs and the same is shown below.

A	0	-
B	4	A
C	1	A
D	math.MaxInt64	-1
E	math.MaxInt64	-1

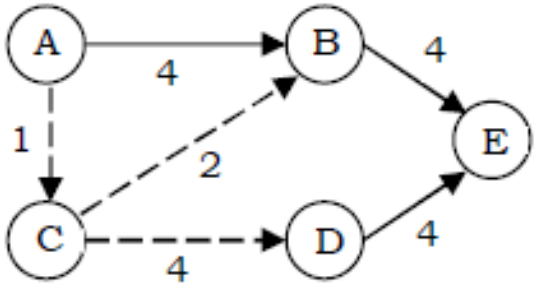


Shortest path from B, C from A

Dystra follows

Now, let us select the minimum distance among all. The minimum distance vertex is C. That means, we have to reach other vertices from these two vertices (A and C). For example, B can be reached from A and also from C. In this case we have to select the one which gives the lowest cost. Since reaching B through C is giving the minimum cost $(1 + 2)$, we update the *Distance* table for vertex B with cost 3 and the vertex from which we got this cost as C.

A	0	-
B	3	C
C	1	A
D	5	C
E	math.MaxInt64	-1

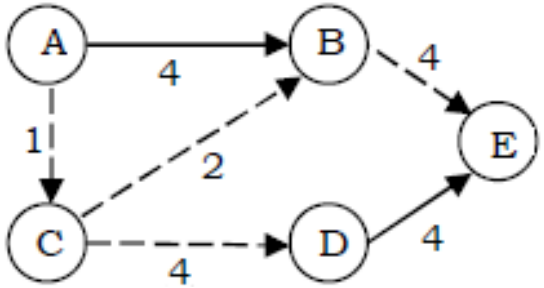


Shortest path to B, D using C as intermediate vertex

Dystra follows

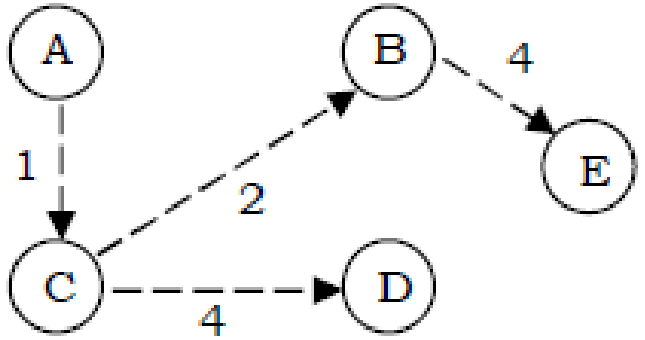
The only vertex remaining is E. To reach E, we have to see all the paths through which we can reach E and select the one which gives the minimum cost. We can see that if we use B as the intermediate vertex through C we get the minimum cost.

A	0	-
B	3	C
C	1	A
D	5	C
E	7	B



Dystra follows

The final minimum cost tree which Dijkstra's algorithm generates is:



Performance

- In Dijkstra's algorithm, the efficiency depends on the number of deleteMins (V deleteMins) and updates for priority queues (E updates) that are used. If a $standard\ binary\ heap$ is used then the complexity is O($ElogV$). The term $ElogV$ comes from E updates (each update takes $logV$) for the standard heap. If the set used is an array then the complexity is O($E + V2$).

Unweighted Shortest Path vs Dijkstra

- Difference between Unweighted Shortest Path and Dijkstra's Algorithm
  1. To represent weights in the adjacency list, each vertex contains the weights of the edges (in addition to their identifier).
  2. Instead of ordinary queue we use priority queue [distances are the priorities] and the vertex with the smallest distance is selected for processing.
  3. The distance to a vertex is calculated by the sum of the weights of the edges on the path from the source to that vertex.
  4. We update the distances in case the newly computed distance is smaller than the old distance which we have already computed.

Disadvantages of Dijkstra's Algorithm

- As discussed above, the major disadvantage of the algorithm is that it does a blind search, thereby wasting time and necessary resources.
- Another disadvantage is that it cannot handle negative edges. This leads to acyclic graphs and most often cannot obtain the right shortest path.

- We will see other graph algorithms in LAB:
  - Kruskals
  - Bellman–Ford
  - Hamiltonian path

In Lab session

- You will play with the concepts and starts getting more and more familiar with graph
- This can be useful for your **FINAL** project

- Lab is done by Remy Belmonte