Lecture 10
Graphs

Dr. Sara S. Elhishi
Information Systems Dept.
Mansoura University, Egypt
Sara_shaker2008@mans.edu.eg

Graphs

Mathematically speaking, a tree is a specific type of graph.

Within computer programming, graphs serve distinct purposes compared to trees.

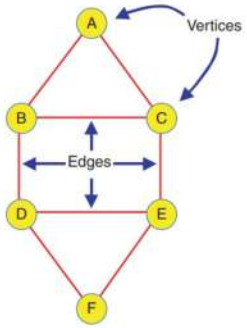
Trees serve data searching.

Graphs depict real-world phenomena.

Ex. Travel Sales Man problem, activities to finish a project, Social Network interactions, etc.

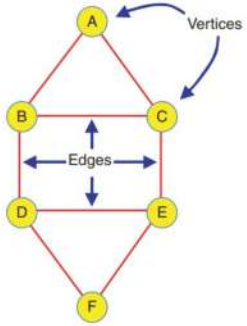
Graphs

- Graphs consist of:
  - Nodes / Vertices
  - Edges / Connections
- Degree of Connectivity rather than special routs.



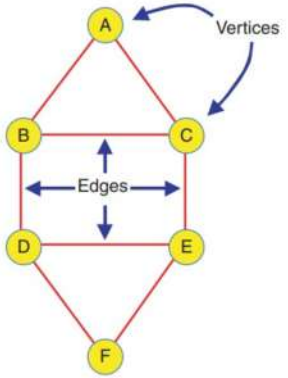
Adjacency / neighbors

- Presence of an edge between 2 vertices.
- C and E are adjacent, while vertices C and F are not.



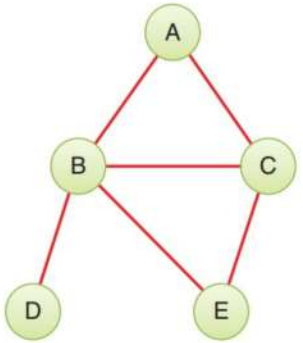
Path

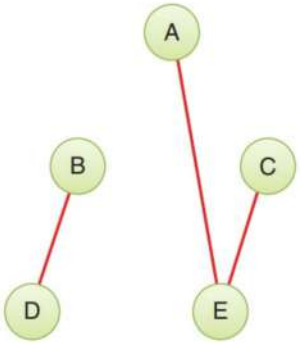
- A sequence of edges.
- Paths from A to F:
  - ABDF
  - ACEF
- Which one is better?



Connected Graphs

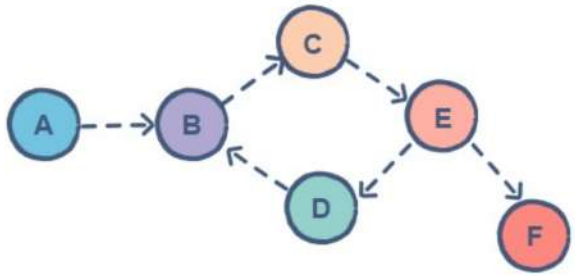
- A **connected** graph is characterized by the presence of at least one path connecting every vertex to every other vertex
- " You Cannot get there from Here?" Not Connected
- Not connected graphs consist of Connected graphs.



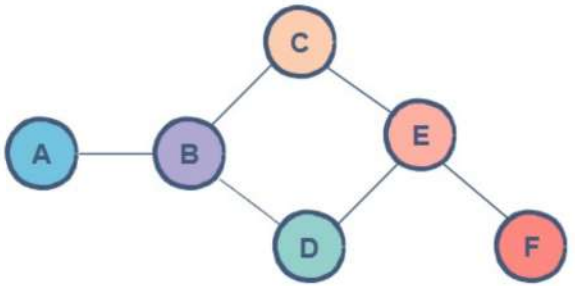


Directed / Undirected Graphs

- Graphs with a specific orientation, an arrowhead at the end of the edge conventionally indicates the permissible direction
- Undirected or Bidirectional graphs.



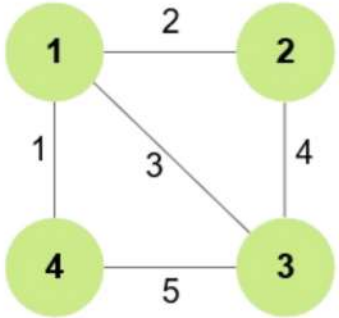
Directed Graph



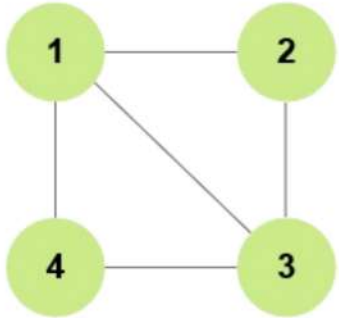
Undirected Graph

Weighted / Unweighted Graphs

- A number on edges represents a property of the edge.
- Ex. Distance, time, cost…etc.



Weighted Graph



Unweighted Graph

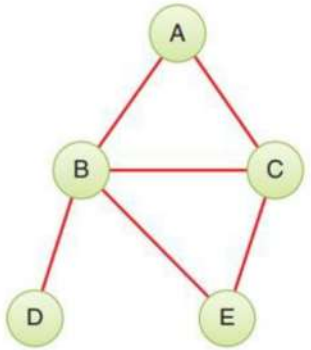
Representing a Graph in Python

- A vertex could be represented using numbers ranging from 0 to N-1.
- A vertex corresponds to an entity in real-world (e.g, a city)
- Hence, it is often advantageous to specify a vertex as an object belonging to a vertex class.
- For example, we will solely store the name

```
1   class Vertex(object):
2       def __init__(self, name):
3           self.name = name # Store the name
4
5       def __str__(self):
6           return '<Vertex {}>'.format(self.name)
7
```

Representing a Graph in Python

- Now, What about edges?
- Unlike Trees, Graphs are unstructured organized.
- Each vertex in a graph can be linked to an unlimited number of other vertices!
- B is connected to 4 vertices, while D is only connected to 1.



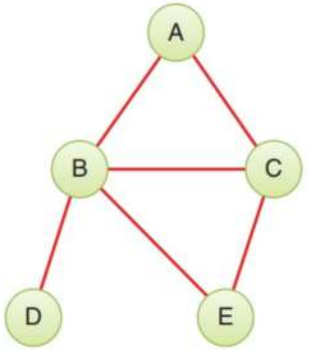
Representing Edges

- To handle this unstructured organization:
  - Adjacency Matrix
  - Adjacency List
- An adjacent vertex is a vertex related to another vertex by a single edge rather than a path across many edges.

Adjacency Matrix

- A two-dimensional array represented by elements that indicate the presence of an edge between two vertices.
- 0s , 1s, and identity diagonal.
- Reflection above and below identity diagonal cause Redundant Storage.

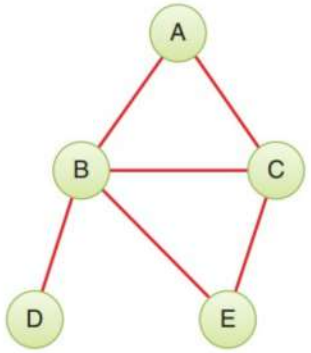
	A	B	C	D	E
A	0	1	1	0	0
B	1	0	1	1	1
C	1	1	0	0	1
D	0	1	0	0	0
E	0	1	1	0	0



Adjacency List

- A type of Linked List.
- A list of lists
- Every individual list incorporates references to the neighboring vertices of that vertex.

Vertex	Adjacent Vertices List
A	B→C
B	A→C→D→E
C	A→B→E
D	B
E	B→C



The Graph Class

```
class Graph(object):
    def __init__(self):
        self.__vertices = []    # list array of vertices
        self.__adjMat = {}      # a hash table mapping vertex pairs

    def nVertices(self):
        return len(self.__vertices)
    def nEdges(self):
        return len(self.__adjMat) // 2 # dividing number of keys by 2

    def addVertex(self, vertex):
        self.__vertices.append(vertex)  # place at end of vertex list

    def validIndex(self, n):
        # check that n is a valid vertex index
        if n < 0 or self.nVertices() <= n:  #outside the range
            raise IndexError
        return True         #otherwise, it is valid

    #get the nth vertex in the graph
    def getVertex(self, n):
        if self.validIndex(n):
            return self.__vertices[n]

    # add an edge between two vertices A and B
    def addEdge(self, A, B):
        self.validIndex(A)
        self.validIndex(B)

        if A == B:
            raise ValueError
        self.__adjMat[A, B] = 1      # add edge in one direction
        self.__adjMat[B, A] = 1      # add edge in the reverse direction

    def hasEdge(self, A, B):
        self.validIndex(A)
        self.validIndex(B)

        return self.__adjMat.get((A, B), False)      # get the edge count or false if there is no
```

Mid Term Q & A

- When we give an abstract data type a physical implementation, we call it a ___
- Back to the parentheses checker problem, having an expression and if the current symbol is ")", we use a/an ____ stack operation
- To evaluate a postfix expression having a stack of operands, and you in case of facing an operator, so it is convienet to ___ .
- What is the postfix notation to the following infix expression: A+BxC ?
- ___ is a mathematical notation in which the operator is placed before the operands such as +AB.

- When we give an abstract data type a physical implementation, we call it a Data Structure
- Back to the parentheses checker problem, having an expression and if the current symbol is ")", we use a/an Pop() stack operation
- To evaluate a postfix expression having a stack of operands, and you in case of facing an operator, so it is convenient to
  Pop 2 operands and evaluate them then push the result to the stack
- What is the postfix notation to the following infix expression: A+BxC ? ABCx+
- Prefix is a mathematical notation in which the operator is placed before the operands such as +AB.

- Search and traversal work on stacks, queues, and priority queues by ___
- The time complexity for inserting an element in a priority queue is ___
- When faced with a mail large inbox, a reader may wish to organize messages into folders based on their level of importance. This task should be implemented using ___.
- differentiate that state from a queue with no elements, you may determine the number of items left by ___
- Let's examine the scenario when the queue wraps around the circular array. Once the value of [maxSize - 1] is reached, the subsequent insertion causes rear to be set to index ___.
- The time it takes to push or pop an item to/frm a stack is ___ by the number of items in the stack.
- Suppose s is a stack with the following ordered operations: s.stack(), s.push("a"), s.push("b"), s.push("c"), s.peek(), s.pop(), s.push("d"). The mentioned s.peek() operation result in _____

- Search and traversal work on stacks, queues, and priority queues by Not applicable
- The time complexity for inserting an element in a priority queue is O(N)
- When faced with a mail large inbox, a reader may wish to organize messages into folders based on their level of importance. This task should be implemented using Priority Queue
- differentiate that state from a queue with no elements, you may determine the number of items left by (front-rear)-1
- Let's examine the scenario when the queue wraps around the circular array. Once the value of [maxSize - 1] is reached, the subsequent insertion causes rear to be set to index zero
- The time it takes to push or pop an item to/frm a stack is Not Influenced by the number of items in the stack.
- Suppose s is a stack with the following ordered operations: s.stack(), s.push("a"), s.push("b"), s.push("c"), s.peek(), s.pop(), s.push("d"). The mentioned s.peek() operation result in c

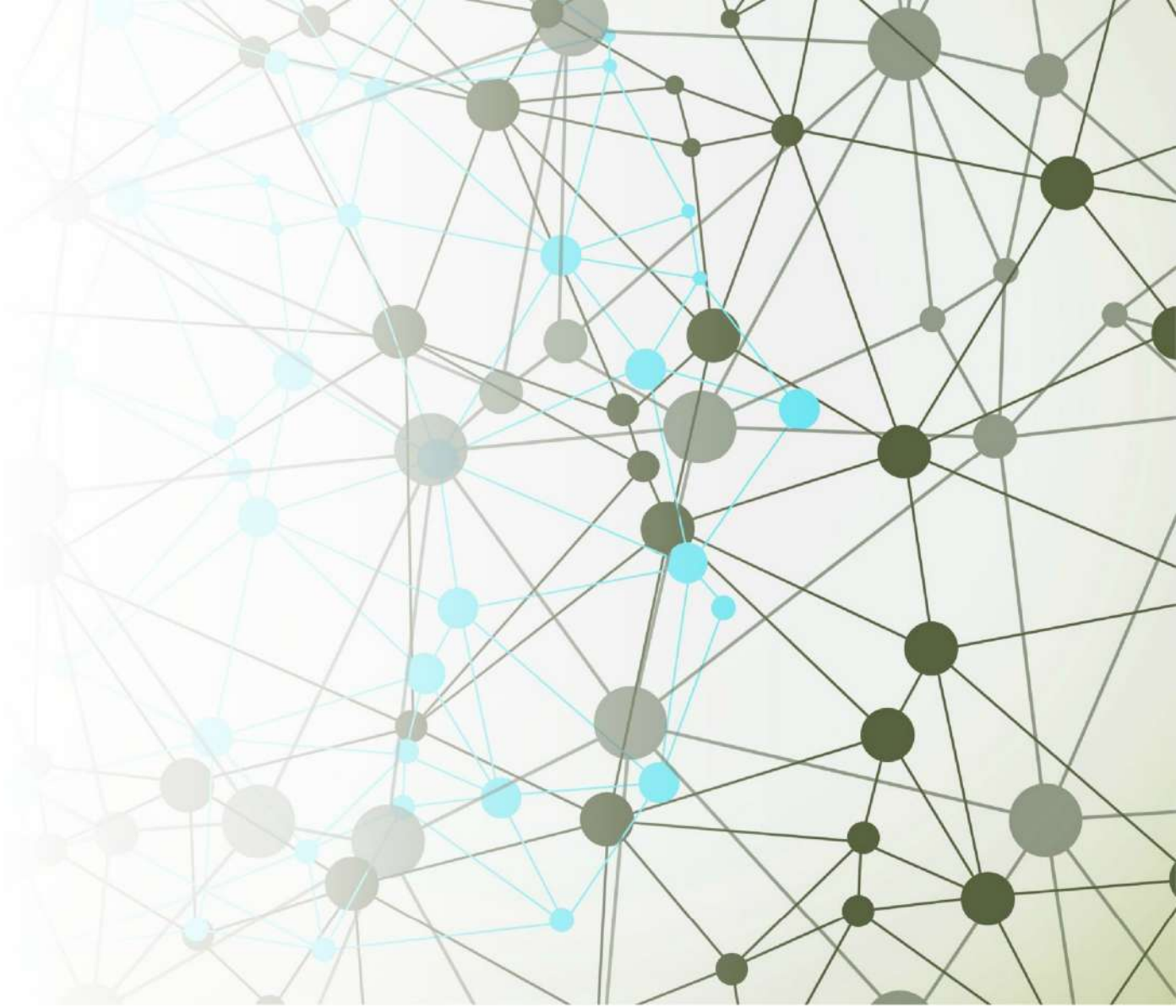
Lecture 11
Graphs

Dr. Sara Elhishi

Information Systems Dept.

Mansoura University, Egypt.

Sara_shaker2008@mans.edu.eg



Graph Revisited

- Graphs depict real-world phenomena.
- Graphs consist of: Nodes / Vertices and Edges / Connections
- Adjacency
- Path
- Connected/Not Connected
- Directed/ Undirected
- Weighted/ Unweighted

The Graph Class

```
class Graph(object):
    def __init__(self):
        self.__vertices = []    # list array of vertices
        self.__adjMat = {}      # a hash table mapping vertex pairs

    def nVertices(self):
        return len(self.__vertices)
    def nEdges(self):
        return len(self.__adjMat) // 2 # dividing number of keys by 2

    def addVertex(self, vertex):
        self.__vertices.append(vertex)  # place at end of vertex list

    def validIndex(self, n):
        # check that n is a valid vertex index
        if n < 0 or self.nVertices() <= n:  #outside the range
            raise IndexError
        return True         #otherwise, it is valid

    #get the nth vertex in the graph
    def getVertex(self, n):
        if self.validIndex(n):
            return self.__vertices[n]

    # add an edge between two vertices A and B
    def addEdge(self, A, B):
        self.validIndex(A)
        self.validIndex(B)

        if A == B:
            raise ValueError
        self.__adjMat[A, B] = 1      # add edge in one direction
        self.__adjMat[B, A] = 1      # add edge in the reverse direction

    def hasEdge(self, A, B):
        self.validIndex(A)
        self.validIndex(B)

        return self.__adjMat.get((A, B), False)      # get the edge count or false if there is no
```

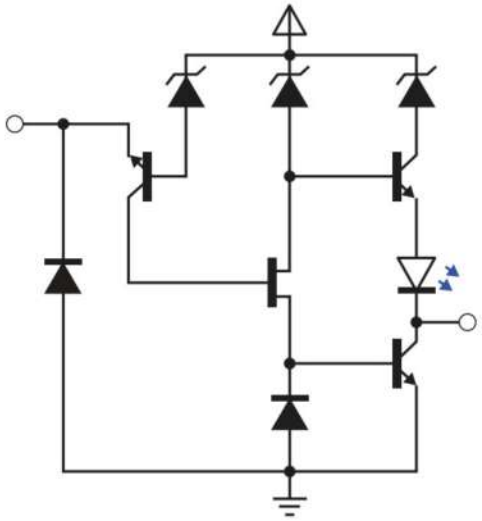
Traversal and Search

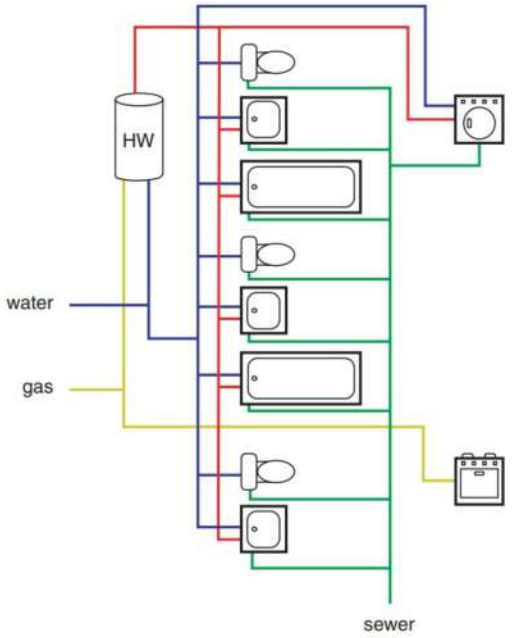
Scenario 1

- Imagine taking a relief trip to a new place by air or sea.
- You'll explore rural areas by bicycle, needing thorough info on destinations.
- While paved and some dirt roads are fine, avoid routes under maintenance or damaged by mud or floods.
- Some towns are accessible, others aren't due to road conditions.
- Adjust your plan as you learn more about road conditions.

Scenario 2

- Circuit design and plumbing network construction
- Components of electronic circuits include transistors that are interconnected by conductors such as cables or metal pathways.
- Within plumbing networks, several components, including water heaters, faucets, drains, and gas burners, are interconnected by pipes.





Graph Traversal

The determination of the vertices that can be accessed from a given vertex is a key operation in graph analysis

There are two fundamental methods: **depth-first (DF)** and **breadth-first (BF).**

When the aim is to stop at a specific vertex, the process is called **depth-first search (DFS**) or **breadth-first search (BFS).**

Depth First

- The depth-first traversal algorithm preserves the destination of a dead end by using a stack
- operates under the assumption of maximizing the distance from the initial point in the shortest possible time.

- ***Rule 1***

- If possible, visit an adjacent unvisited vertex, mark it, and push it on the stack.

- ***Rule 2***

- If you can't follow Rule 1, then, if possible, pop a vertex off the stack.

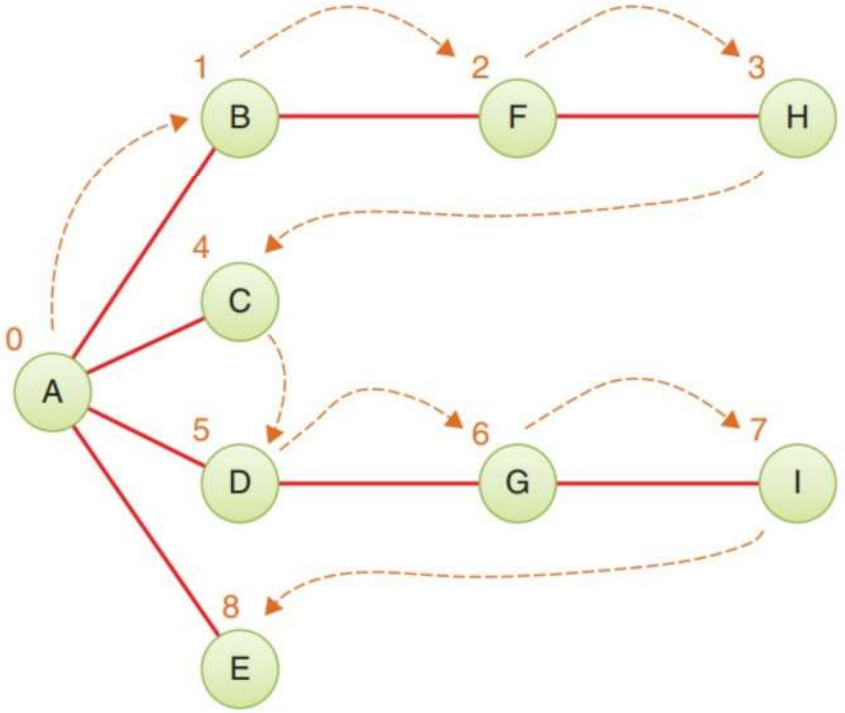
- ***Rule 3***

- If you can't follow Rule 1 or Rule 2, you're done.

Depth First

ABFHCDGIE

Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH
Pop H	ABF
Pop F	AB
Pop B	A
Visit C	AC
Pop C	A
Visit D	AD
Visit G	ADG
Visit I	ADGI
Pop I	ADG
Pop G	AD
Pop D	A
Visit E	AE
Pop E	A
Pop A	
Done	



Python Implementation

Get Unvisited Vertices adjacent to a specific Vertex

```
class Graph(object):

    # Traversing Adjacent Vertices
    # generate sequence of all vertex indices
    def vertices(self):
        return range(self.nVertices)
    # generate sequence of all vertex indices that are adjacent of vertex n
    def adjacentVertices(self, n):
        self.validIndex(n)
        for j in self.vertices():
            if j != n and self.hasEdge(n, j):
                yield j     # if other vertex connects via edge, yeild other vertex index

    # generate sequence of all vertex indices that are adjacent of vertex n
    # that are not show in the visited list
    def adjacentUnvisitedVertices(self, n, visited, markVisited=True):
        for j in self.adjacentVertices(n):  # loop through adjacents
            if not visited[j]:
                if markVisited:             # check visited flag
                    visited[j] = True       # mark the visit
                yield j                     # yield the vertex index
```

The depthFirst() Method

```
class Stack(list):
    def push(self, item): self.append(item)
    def peek(self): return  self[-1] # last element is top of stack
    def isEmpty(self): return len(self) == 0
```

```
class Graph(object):

    # Traverse in Depth-First Order
    def depthFirst(self, n):
        self.validIndex(n)
        visited = [False] * self.nVertices()    # nothing is visited initially
        stack= Stack()
        stack.push(n)
        visited[n] = True
        yield(n, stack)     # yeild initial vertex and initial path
        while not stack.isEmpty():
            visit = stack.peek()    # top is vertex being visited
            adj = None
            # Loop over adjacent vertices marking them as we visit them
            for j in self.adjacentUnvisitedVertices(visit, visited):
                adj = j     # Next vertex is first adjacent unvisited one, and the rest will be visited later
                break
            if adj is not None: # If there's an adjacent unvisited vertex
                stack.push(adj)
                yield(adj, stack)
            else:
                stack.pop()
```

Breadth First

- The algorithm first traverses all the vertices in close proximity to the initial vertex and, after that, proceeds to distant vertices.
- Utilizes a queue

- ***Rule 1***

  - Take the first vertex in the queue (if there is one) and insert all its adjacent unvisited vertices into the queue, marking them as visited.

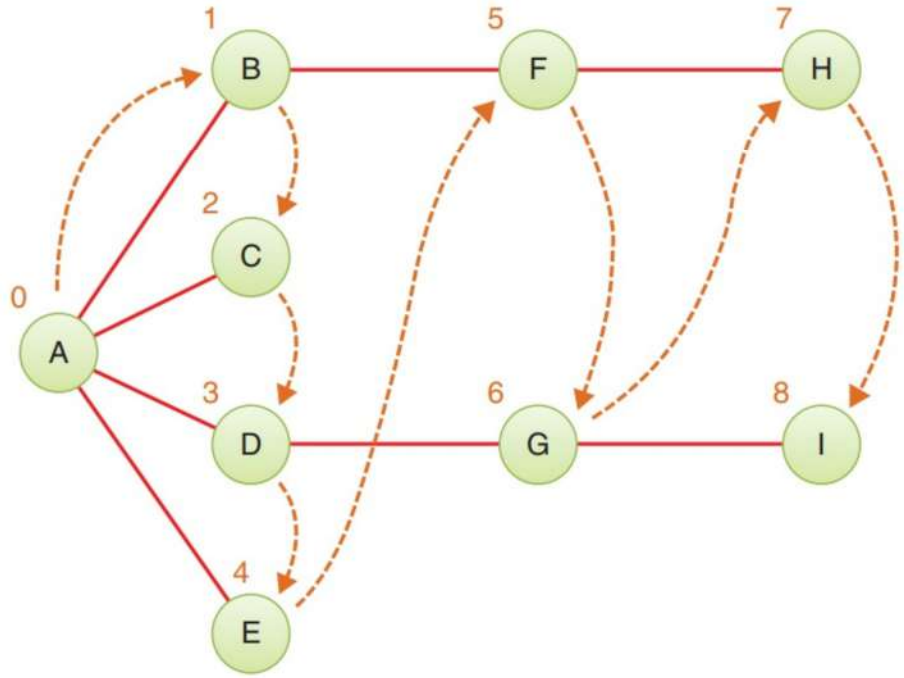
- ***Rule 2***

  - If you can't carry out Rule 1 because the queue is empty, you're done.

Breadth First

ABCDEFGHI

Event	Queue (Front to Rear)
Visit A	A
Remove A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Remove B	CDE
Visit F	CDEF
Remove C	DEF
Remove D	EF
Visit G	EFG
Remove E	FG
Remove F	G
Visit H	GH
Remove G	H
Visit I	HI
Remove H	I
Remove I	
Done	



Python Implementation

```
7   class Queue(object):
8       def insert(self, j): self.append(j) # insert == append
9       def peek(self): return self[0] # First element is front of queue
10      def remove(self): return self.pop(0) # Remove first element
11      def isEmpty(self): return len(self) == 0
12
```

```
94      # Traverse in Breadth-First Order
95      def breadthFirst( self, n):
96          self.validIndex(n) # Check that vertex n is valid
97          visited = [False] * self.nVertices() # Nothing visited initially
98          queue = Queue() # Start with an empty queue and
99          queue.insert(n) # insert the starting vertex index on it
100         visited[n] = True # and mark starting vertex as visited
101         while not queue.isEmpty(): # Loop until nothing left on queue
102             visit = queue.remove() # Visit vertex at front of queue
103             yield visit # Yield vertex to visit it
104             # Loop over adjacent unvisited vertices
105             for j in self.adjacentUnvisitedVertices(visit, visited):
106                 queue.insert(j) # and insert them in the queue
107
```

DF vs. BF

Feature	DFS	BFS
Data Structure	Stack (explicit or implicit)	Queue
Traversal Pattern	Deep first, then backtrack	Level by level
Memory Usage	Proportional to depth	Proportional to breadth
Shortest Path	No (in unweighted graphs)	Yes (in unweighted graphs)
Applications	Topological sorting, puzzles	Shortest path, level-order traversal
Advantages	Lower memory in sparse graphs, efficient	Guaranteed shortest path, finds all levels
Disadvantages	Can get trapped in deep loops, no shortest path	Higher memory in large, wide graphs

Thanks