Data Structures and Algorithms

Lesson 8. Graphs

Alexandra Ciobotaru

7 apr 2022

Syllabus

- **Lesson 1. Introduction: Algorithms, Data Structures and Cognitive Science.**
- **Lesson 2. Python Data Types and Structures.**
- **Lesson 3. Principles of Algorithm Design.**
- **Lesson 4. Lists and Pointer Structures.**
- **Lesson 5. Stacks and Queues.**
- **Lesson 6. Trees.**
- **Lesson 7. Hashing and Symbol Tables.**
- **Lesson 8. Graphs.**
- Lesson 9. Searching.
- Lesson 10. Sorting. Mid-term evaluation quiz (30%)
- Lesson 11. Selection Algorithms.
- Lesson 12. String Algorithms and Techniques.
- Lesson 13. Design Techniques and Strategies.
- Lesson 14. Implementations, Applications and Tools.
- Lab (20%) + Project (50%).

Graphs

- The concept of graphs comes from a branch of mathematics called graph theory
- A graph is a set of vertices (nodes) and edges that form connections between the vertices.
- A graph **G** is an ordered pair of set **V** of vertices and a set **E** of edges

$$G = (V, E)$$

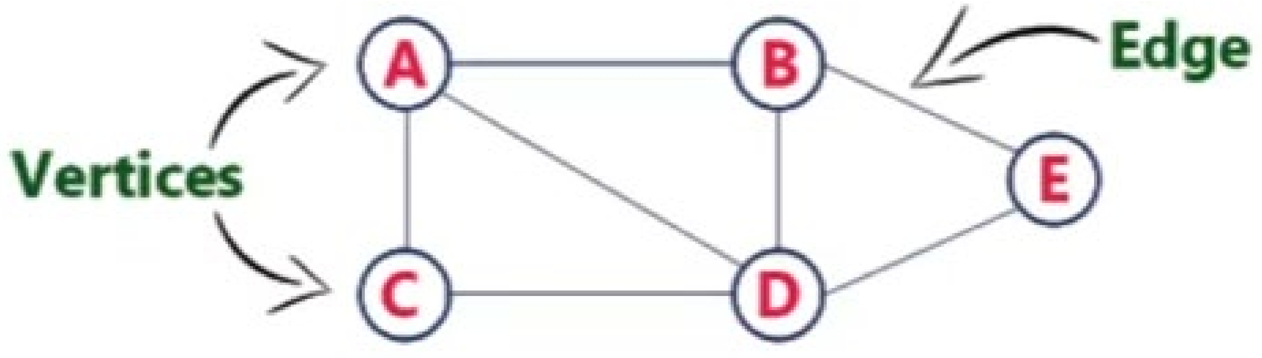
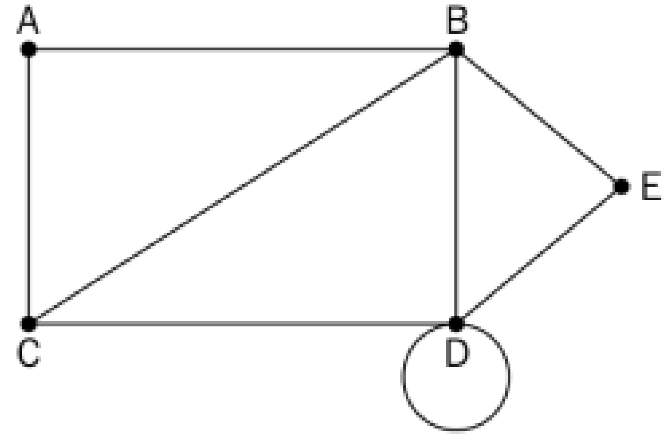


Image source: https://www.youtube.com/watch?v=iv5DcAi411I

Graphs

- Graphs are non-linear data structures that represent data by connecting a set of nodes (vertices) along their edges
- *Trees* are a form of graphs
- Graphs are a common method to visually illustrate relationships in the data and solve a number of computer problems
  - http://wordvis.com/
  - Shortest paths in google maps
  - Social networking (connections, mutual friends)
  - Routing algorithms (internet data comes on your pc by passing through a network of routers)
  - Delivery routes (salesman problem) – warehouse -> cities -> warehouse
  - Neural networks

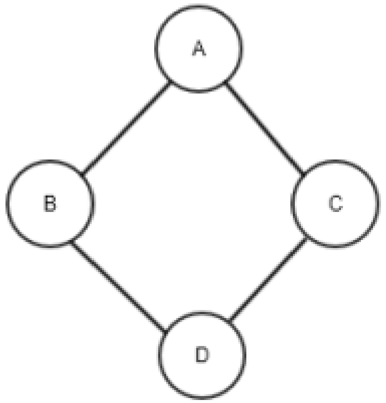
Definitions

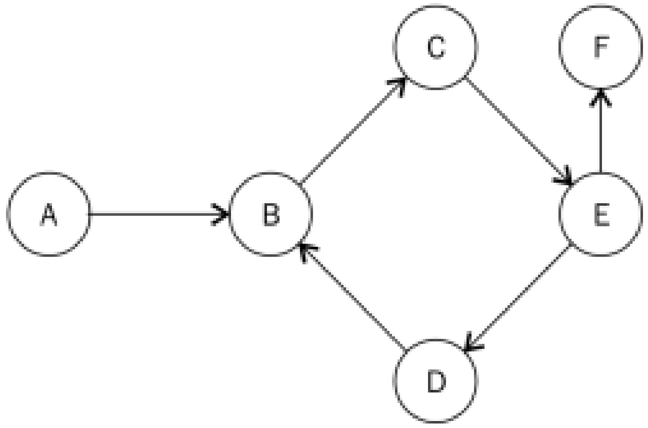


- **Node or Vertex** – a point in the graph (A, B, C, D, E)
- **Edge** – connection between two vertices (ex. the AB line)
- **Loop** – when an edge from a node is incident on itself (D)
- **Degree** of a vertex – total no. of edges that are incident on a given vertex (ex. B has degree 4)
- **Adjacency** - Two nodes are said to be adjacent if there is a direct connection between them (ex. A and B, A and C)
- **Path** – sequence of adjacent vertices between two nodes (ex. CABE is a path from C to E, but so is the path CDE)
- **Leaf** vertex (pendant vertex) – a vertex is a leaf vertex if it has exactly one degree.

Directed or undirected graphs

- The connecting edges can be considered directed or undirected
  - Undirected edges -> undirected graph (or bidirectional)
  - Directed edges -> directed graph
- An undirected graph represents edges as lines between nodes
- In a directed graph, the edges provide information of connection between any two nodes in the graph
- If the edge between A and B is directed, then:
  - (A, B) ≠ (B, A)
  - One can move from A to B, not from B to A

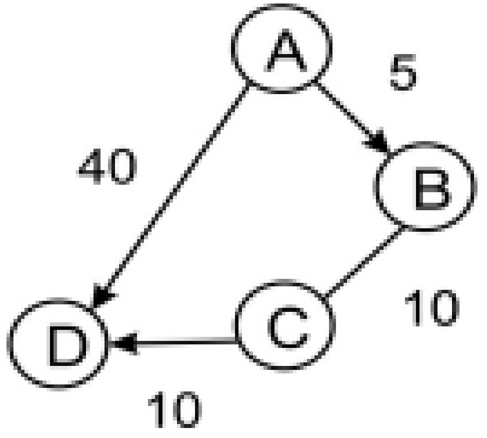




Indegree and Outdegree

- In a directed graph, each node (or vertex) has an *indegree* and an *outdegree*:
  - Indegree = The total number of edges that come into the vertex
  - Outdegree = The total number of edges that go out from the vertex
- What is the indegree of node E? What about its outdegree?
- Other definitions:
  - Isolated node = node that has a degree of 0
  - Source node = node that has an indegree of 0 (what is the source node in the previous diagram?)
  - Sink node = node that has an outdegree of 0 (what is the sink node in the previous diagram?)

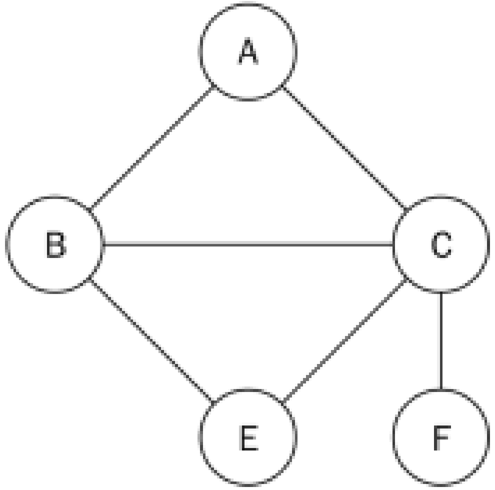
Weighted Graphs



- A weighted graph is a graph that has a numeric weight associated with the edges in the graph.
- It can be either a directed or an undirected graph.
- Depending on the purpose of the graph, the weight can be used, for example, to indicate distance or cost
- Example: let’s consider the weights the amount of time, in minutes, for the journey to the next node:
  - Path A-D (40min)
  - Path A-B-C-D (25min)
- If the only concern is time, then it would be better to travel on A-B-C-D

Graph representations

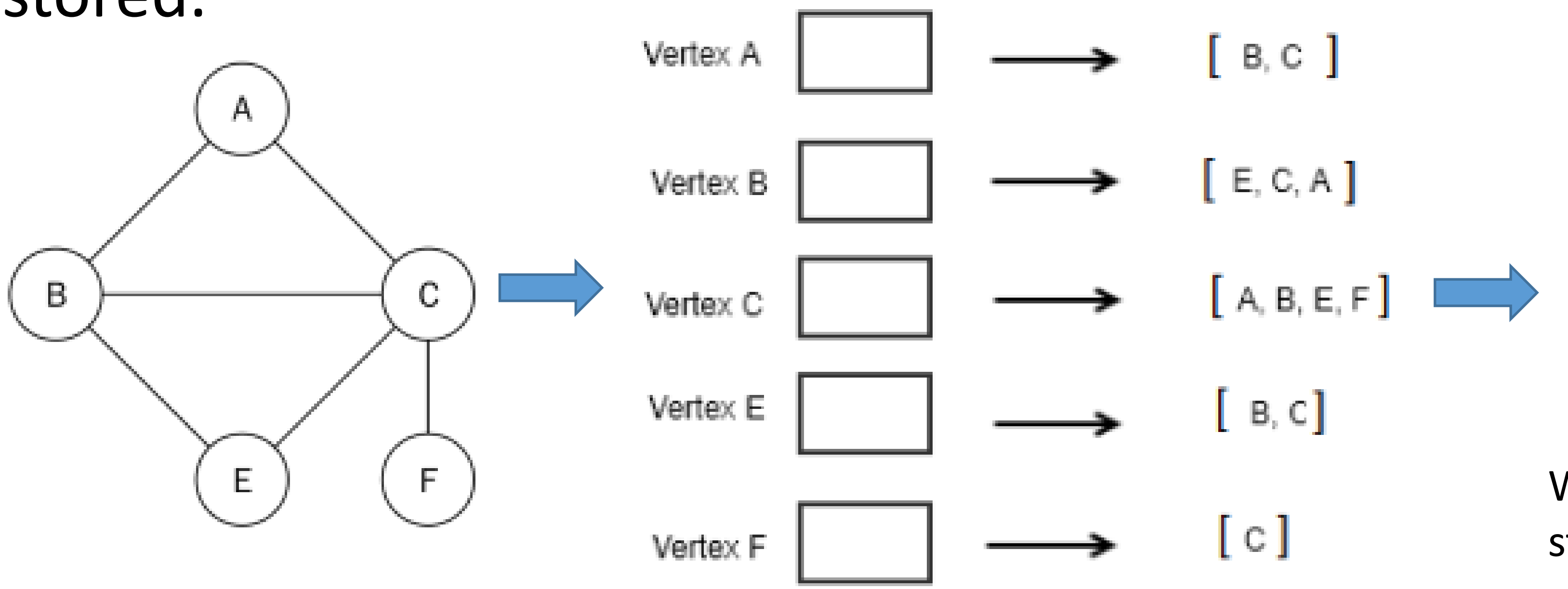
- There are two main ways of representing graphs:
  - Using adjacency lists
  - Using adjacency matrices
- Let’s consider the following graph, and we’ll represent it in both forms:



- What kind of graph is it? Directed or undirected?

Adjacency Lists (what we will use)

- An adjacency list stores all the nodes, along with other nodes that are directly connected to them in the graph.
- The indices of the list can be used to represent the nodes or vertices in the graph. At each index, the adjacent nodes to that vertex are stored.

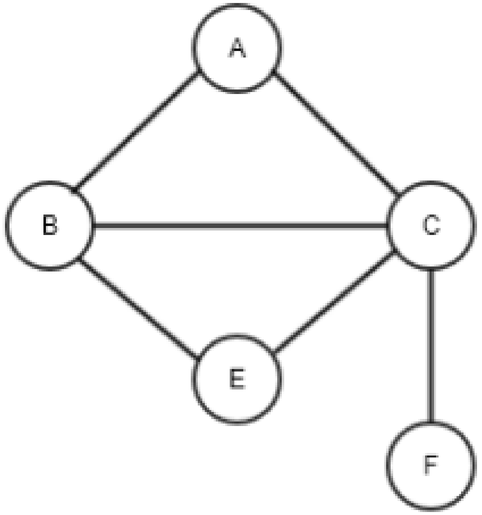



```
graph = dict()
graph['A'] = ['B', 'C']
graph['B'] = ['E','C', 'A']
graph['C'] = ['A', 'B', 'E','F']
graph['E'] = ['B', 'C']
graph['F'] = ['C']
```

We can also use a dictionary data structure to represent the graph

Adjacency Matrix

- We represent the cells in the matrix with a 1 or 0, depending on whether two vertices are connected by an edge or not.



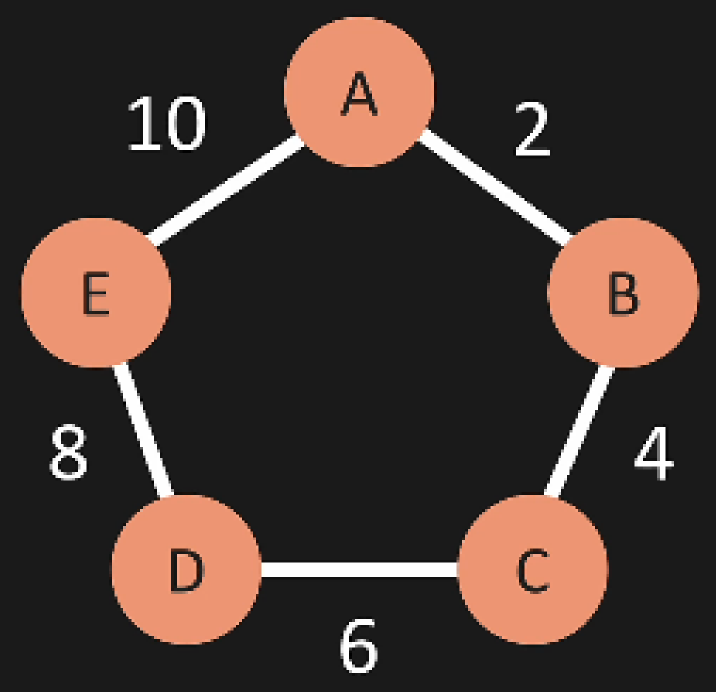
	A	B	C	E	F
A	0	1	1	0	0
B	1	0	1	1	0
C	1	1	0	1	1
E	0	1	1	0	0
F	0	0	1	0	0

The items the node has an edge with

The nodes

- A has edges with B and C, B has edges with A, C and E
- The adjacency matrix has zeros on the diagonal
- If the graph is unidirectional (bidirectional) this matrix is always symmetrical on the diagonal

Weighted Adjacency Matrices



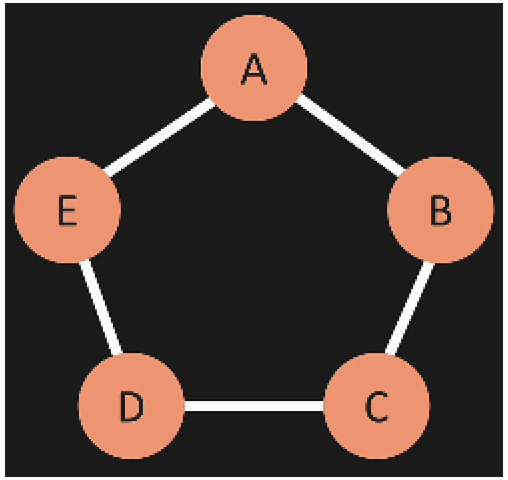
	A	B	C	D	E
A	0	2	0	0	10
B	2	0	4	0	0
C	0	4	0	6	0
D	0	0	6	0	8
E	10	0	0	8	0

Big O

- Suppose we have this graph – we'll represent it with both adjacency list and adjacency matrix

```
{
  "A": ['B','E'],
  'B': ['A','C'],
  'C': ['B','D'],
  'D': ['C','E'],
  'E': ['A','D']
}
```

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



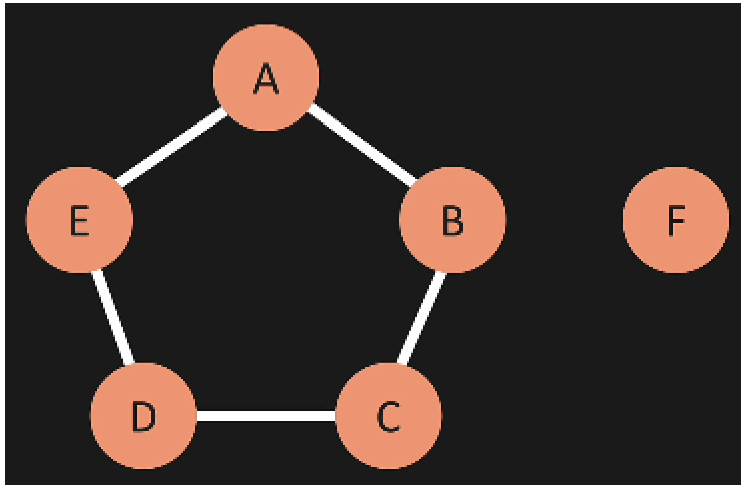
- In a matrix, each vertex has to store all of the vertices it is not connected to – so from a space complexity stand point the adjacency matrix is $O(|V|^2)$ while the adjacency list is $O(|V|+|E|)$

Big O – adding a vertex

- Suppose we want to add the vertex F

```
{
    "A": ['B','E'],
    'B': ['A','C'],
    'C': ['B','D'],
    'D': ['C','E'],
    'E': ['A','D'],
    ‘F’: []
}
```

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



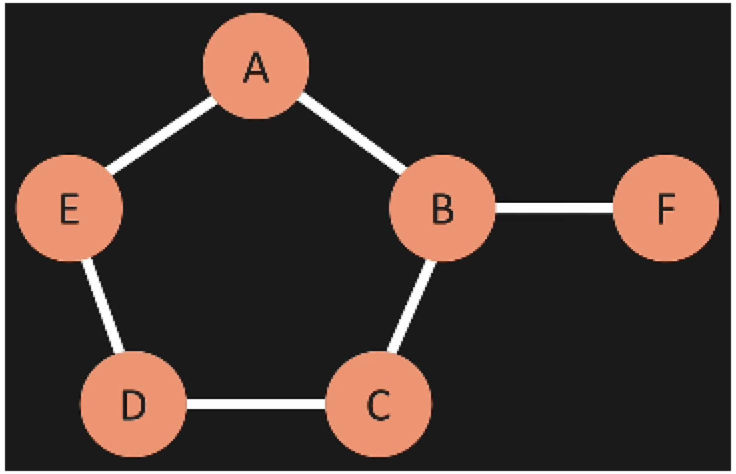
- In a matrix, adding a new vertex means adding a new row and a new column – it’s $O(|V^2|)$, while for an adjacency list this operation is O(1)

Big O – adding an edge

- Suppose we want to add the edge between vertices B and F

{
"A": ['B','E'],
'B': ['A','C‘, ‘F’],
'C': ['B','D'],
'D': ['C','E'],
'E': ['A','D'],
‘F’: [‘B’]
}

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



- In this case, both in adjacency list and adjacency matrix the operation is O(1)

Big O – removing an vertex

- Suppose we want to remove the vertex F

{

"A": ['B','E'],

'B': ['A','C'~~,'F'~~],

'C': ['B','D'],

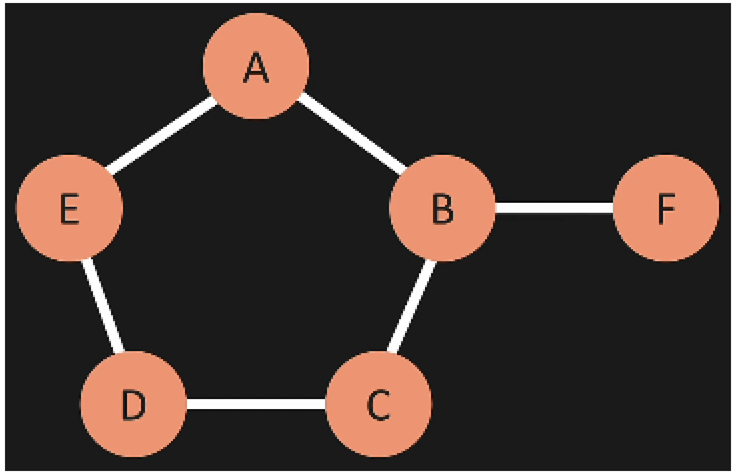
'D': ['C','E'],

'E': ['A','D'],

'F': ~~['B']~~

}

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



- In the adjacency list, first we go to key F and remove its elements O(|E|), then loop through each key to search for F edges (O(|V|) => O(|E|) + O(|V|)
- In the matrix, we change the 1s on F into 0s – O(1), then we need to remove the line and column associated with F – so its O(|V²|)

Add Vertex (Node)

That is all we want to create for now:

{
  'A': []
}

```
class Graph:
    def __init__(self):
        self.adj_list = {} #create empty dict
    def print_graph(self):
        for vertex in self.adj_list:
            print(vertex, ':', self.adj_list[vertex])
    def add_vertex(self, vertex):
        # we add the new vertex only if it’s not already 
in the adjacency list
        if vertex not in self.adj_list.keys():
            self.adj_list[vertex] = []
            return True
        return False
```

Add Edge

We want to connect two nodes, 1 and 2:

```
{
   ‘1': [2],
   ‘2’: [1]
}
```

```
def add_edge(self, v1, v2):
      # if v1 and v2 both exist, we create an edge between them
    if v1 in self.adj_list.keys() and v2 in self.adj_list.keys():
        # we append v2 at v1
        self.adj_list[v1].append(v2)
         # we append v2 at v1
        self.adj_list[v2].append(v1)
        return True
    # otherwise, the method returns False
    return False
```

Remove Edge

We want to eliminate the connection:
{
 '1': [],
 '2': []
}

- The methods gets passed in the name of the vertexes to be removed

```
def remove_edge(self, v1, v2):
    # if v1 and v2 both exist as keys in the adjacency list
    if v1 in self.adj_list.keys() and v2 in self.adj_list.keys():
        # at the key of v1 we remove v2
        self.adj_list[v1].remove(v2)
        # at the key of v2 we remove v1
        self.adj_list[v2].remove(v1)
        return True
    # otherwise, return False
    return False
```

Test the method

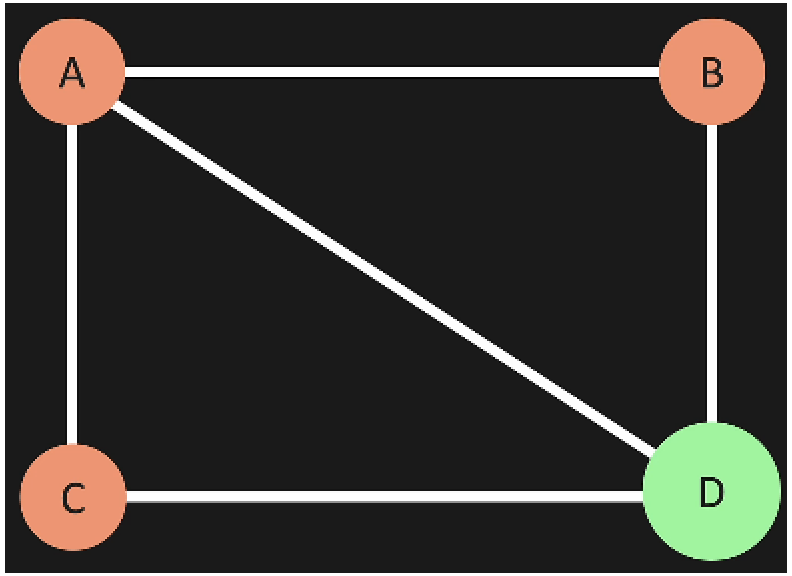
```
my_graph = Graph()
my_graph.add_vertex('A')
my_graph.add_vertex('B')
my_graph.add_vertex('C')

my_graph.add_edge('A','B')
my_graph.add_edge('B','C')
my_graph.add_edge('C','A')

my_graph.remove_edge('A', 'B')

my_graph.print_graph()
```

Test the method

- But if I add a new vertex and try to remove an edge that does not exist, the script will through a ValueErorr. We need to catch this error:

```
def remove_edge(self, v1, v2):
        if v1 in self.adj_list.keys() and v2 in
self.adj_list.keys():
            try:
                self.adj_list[v1].remove(v2)
                self.adj_list[v2].remove(v1)
            except ValueError:
                pass #pass says "ignore this and move on"
            return True
        return False
```

Remove Vertex (Node)

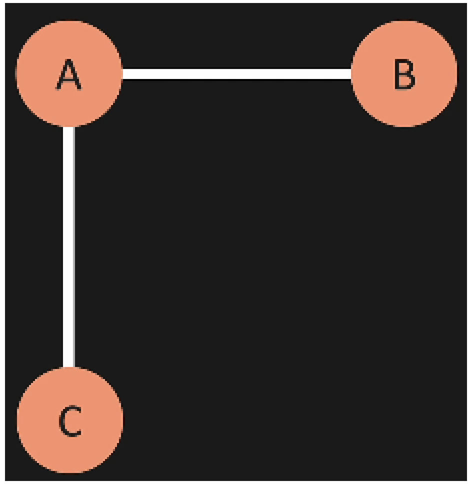


- We want to remove the node D
- For that, first we need to remove its edges

```
{
    "A": ['B','C','D'],
    'B': ['A','D'],
    'C': ['A','D'],
    'D': ['A','B','C']
}
```

- If D has an edge with another vertex, that vertex also has an edge with D
- We loop through the list in key D and for each element we remove D from the list at the element’s key
- Then we remove D vertex

Remove Vertex (Node)



- We want to remove the node D
- For that, first we need to remove its edges

```
{
      "A": ['B','C','D'],
      'B': ['A','D'],
      'C': ['A','D'],
      'D': ['A','B','C']
}
```

- If D has an edge with another vertex, that vertex also has an edge with D
- We loop through the list in key D and for each element we remove D from the list at the element's key
- Then we remove D vertex

Remove Vertex (Node)

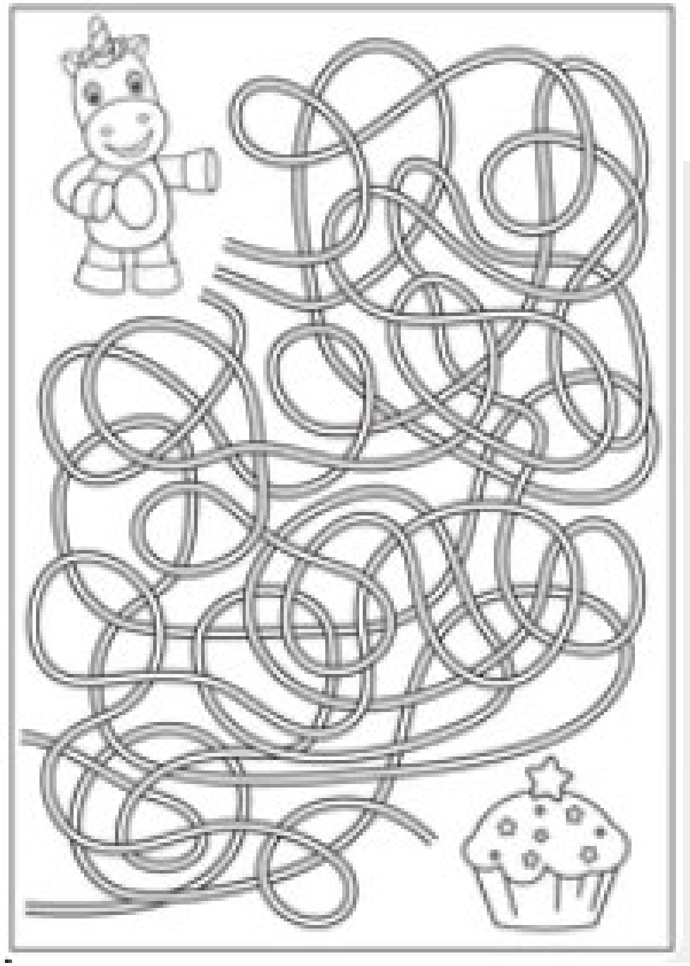
```
def remove_vertex(self, vertex):
    # first we make sure that vertex actually exists
    if vertex in self.adj_list.keys():
        # we go through the list of edges associated with the vertex
        for other_vertex in self.adj_list[vertex]:
            # we go to the list of the other vertices and remove the edge associated to the vertex that we want to remove from the graph
            self.adj_list[other_vertex].remove(vertex)
        # now we delete the vertex itself
        del self.adj_list[vertex]
        return True
    return False
```

Testing the method

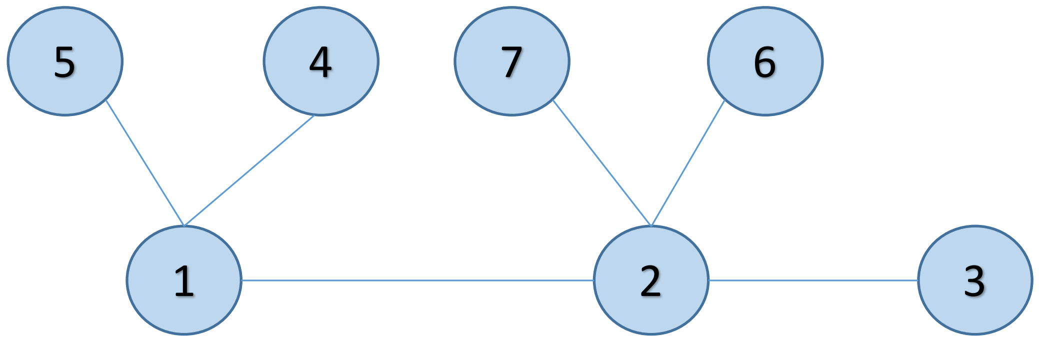
```
my_graph = Graph()
my_graph.add_vertex('A')
my_graph.add_vertex('B')
my_graph.add_vertex('C')
my_graph.add_vertex('D')

my_graph.add_edge('A','B')
my_graph.add_edge('A','C')
my_graph.add_edge('A','D')
my_graph.add_edge('B','D')
my_graph.add_edge('C','D')

my_graph.remove_vertex('D')
```

Graph Traversals

- A graph traversal means to visit all the vertices of the graph, while keeping track of which nodes (vertices) have already been visited and which ones have not.
- A graph traversal algorithm is ***efficient*** <u>if it traverses all the nodes of the graph in the minimum possible time</u>.
- A common strategy of graph traversal is to follow a path until a dead end is reached, then traverse back up until there is a point where we meet an alternative path (just like in a kids puzzle) – this is called “depth first search”.
- Graph traversal algorithms can be useful to determine how to reach from one vertex to another in a graph, and which path from A to B vertices in the graph is better than other paths.



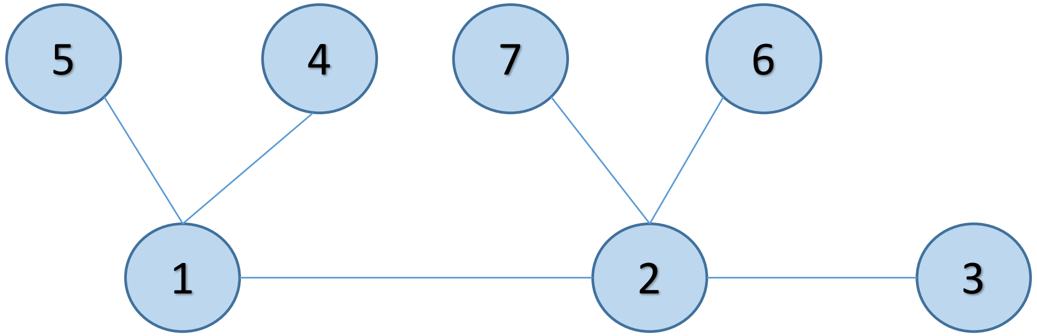
Breadth-First Traversal (BFS)



- Visiting a vertex
- Exploration of a vertex = visiting all its adjacent vertices

- We select 1 as the starting vertex (we can start with any vertex). BFS: **1**
- Now I start exploring vertex 1 – its adjacent vertices are 5, 4, 2 (I can visit them in any order). BFS: 1, **2, 4, 5**
- Next I should select another vertex for exploration. 4 and 5 are already visited (they have no adjacent nodes) so I move to vertex 2 for exploration – its adjacent vertices are 7, 6, 3 (I can visit them in any order). BFS: 1, 2, 4, 5, **7, 6, 3**
- That's all – all vertices are visited and there are no vertices remaining for exploration

Depth-First Search (DFS)

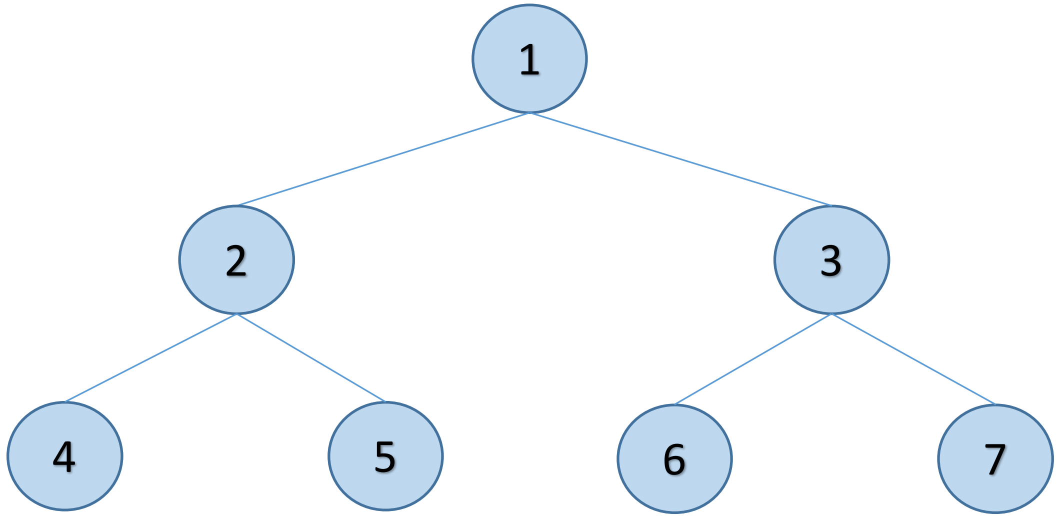


- DFS traverses the depth of a path before traversing its breadth
- Thus, child nodes are visited first, before sibling nodes

- We select 1 as the starting vertex (we can start with any vertex). DFS: **1**
- From 1 I need to start exploration, so I go to vertex 2. DFS: 1, **2**
  - The other adjacent vertices were 5 and 4, but we don't visit them. We have reached a new vertex, so we start exploring that vertex
- Now I start exploring vertex 2 – its adjacent vertices are 7, 6 and 3. I want to go to 3. DFS: 1, 2, **3**
- Now I start exploring vertex 3 – it has no adjacent vertices, so 3 is completely explored. In this point I come back and continue the exploration of 2 => DFS: 1, 2, 3, **6**
- I explore 6, nothing is there – I come back to 2. I explore 7, nothing is there – I come back to 2 => DFS: 1, 2, 3, 6, **7**
- After I finished exploring 2 I go back to 1 and continue the exploration of 1 – I visit 4. DFS: 1, 2, 3, 6, 7, **4**
- I explore 4, nothing is there – I come back to 1. I visit 5. DFS: 1, 2, 3, 6, 7, 4, **5.** Now all vertices are explored.

Exercise

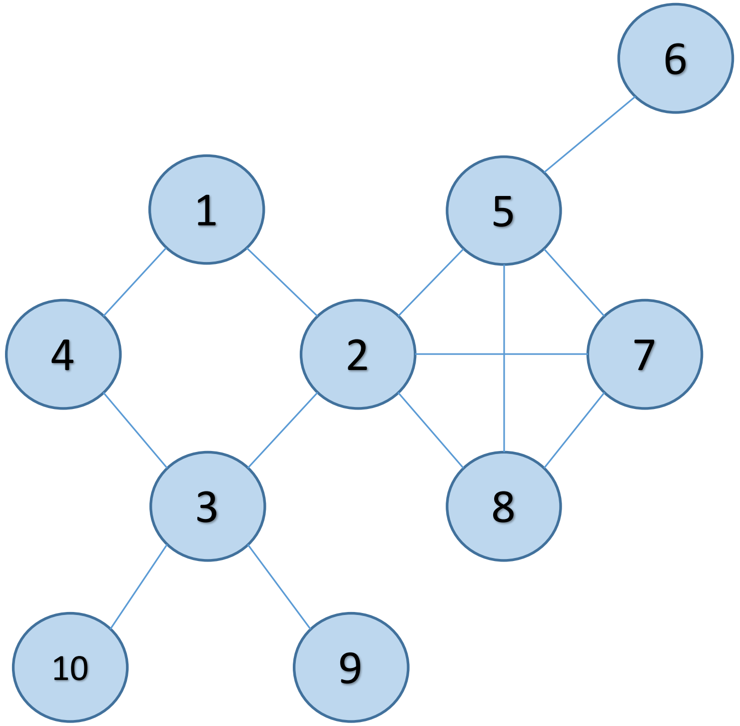
- Explore the following binary tree, BFS and DFS, starting from vertex 1.

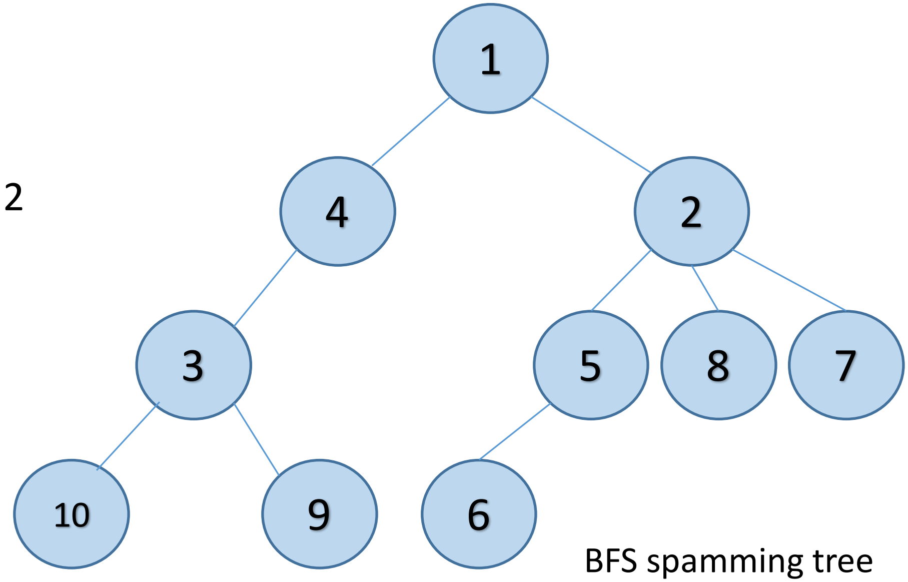


BFS in-depth

- For traversing this complex graph we'll use a queue Q: | | | | | | | | | |.
- We start exploration from any vertex. I'll select vertex 1: Q: |1| | | | | | | | |. BFS: 1
- I take out from Q the vertex 1 and start exploring it – first I visit 4, so I add it to result and I add it to Q: |~~1~~|4| | | | | | | |. BFS: 1, 4
- Next I visit 2, so I add it to result and I add it to Q. Now 1 is completely explored. Q: |~~1~~|4|2| | | | | | |. BFS: 1, 4, 2
- Now I select the next node for exploration from Q, that is 4, and I start exploring 4. I visit 3, and I add it to the result and I add it to Q: |~~1~~|4|2|3| | | | | |. BFS: 1, 4, 2, 3
- Now I select in Q the next node for exploration, which is 2. It's adjacent vertices are 3, which is already explored, 5, 7, 8. I select 5, add it to result and add it to Q. Q: |~~1~~|4|~~2~~|3|5| | | | |. BFS: 1, 4, 2, 3, 5
- Next I go to 8: Q: |~~1~~|~~4~~| ~~2~~|3 | 5|8 | | | |. BFS: 1, 4, 2, 3, 5, 8
- Next I go to 7: Q: |~~1~~|~~4~~| ~~2~~|3 | 5|8 |7 | | |. BFS: 1, 4, 2, 3, 5, 8, 7
- Now I select from Q the next vertex for exploration, 3, which has the adjacent nodes: 2 (was visited), 9, 10. I select 10: Q: |~~1~~|~~4~~| ~~2~~|~~3~~| 5|8|7|10| |. BFS: 1, 4, 2, 3, 5, 8, 7, 10
- Next I go to 9: Q: |~~1~~|~~4~~| ~~2~~|~~3~~| 5|8 |7 | 10| 9|. BFS: 1, 4, 2, 3, 5, 8, 7, 10, 9
- Now I select from Q the next vertex for exploration, 5, which has the neighbors: 2 , 8, 7, 6. I visit 6: Q: |~~1~~|4|~~2~~|~~3~~|~~5~~|8|7|10|9|6|. BFS: 1, 4, 2, 3, 5, 8, 7, 10, 9, 6
- All the other vertices were explored.

Source and DFS in-depth: https://youtu.be/pcKY4hjDrxk





BFS spamming tree

Thank you!