Programação e Algoritmia
--x--
Programming and Algorithms

1 – Object-Oriented Programming

Non-linear data structures

With less implementation details ☺

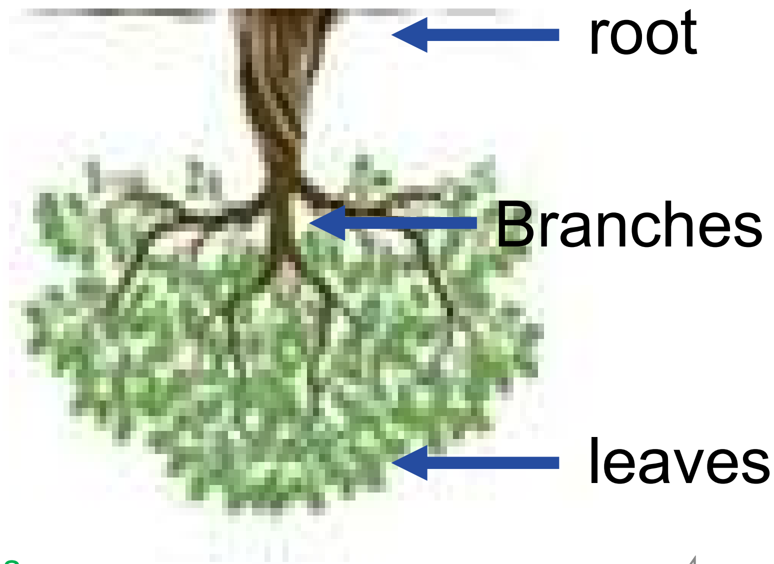
Trees

A brief introduction

What are trees?

- A tree is a hierarchical form of data structure.
- In a tree* there is a parent-child relationship between items.
  - Instead of items followed each other as in lists, queues and stacks

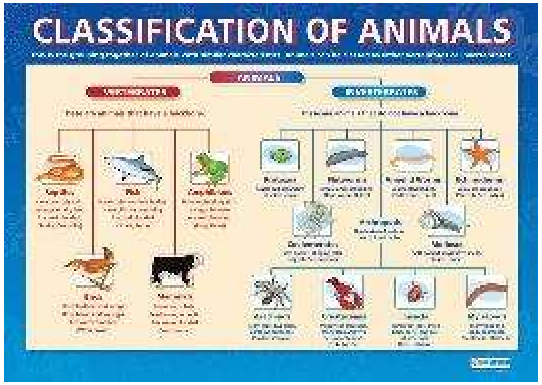
- To visualize what trees look like, imagine a tree growing up from the ground.
- Trees in programming are normally drawn downward, so you would be better off imagining the root structure of the tree growing downward.



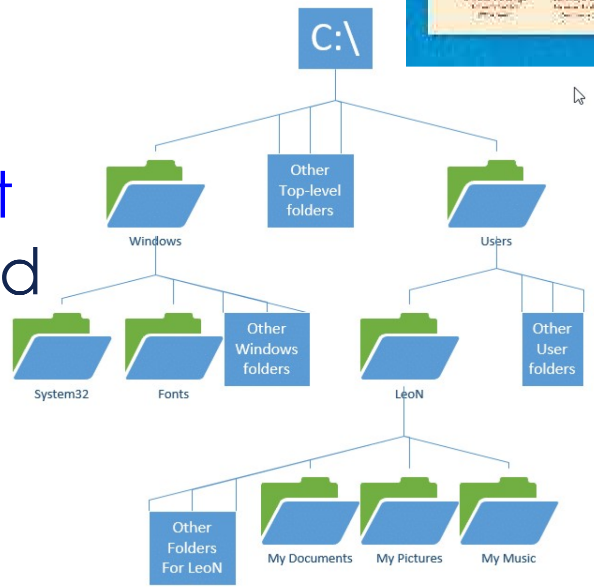
* More precisely in trees of interest to us, with oriented branches.

Applications

- Organograms
- Taxonomies



- Representation of hierarchical information
  - Example: filesystems

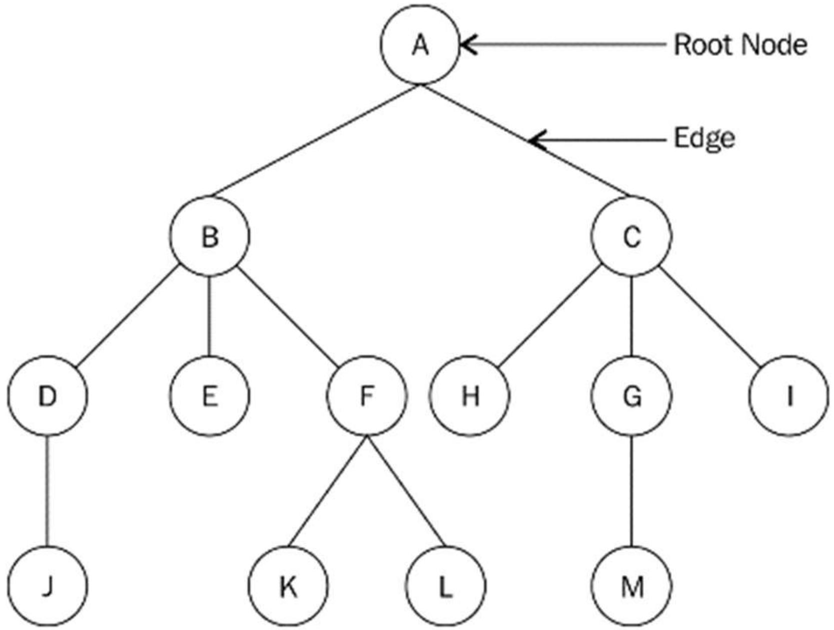


- The HTML/XML Document Object Model is organized in the form of a tree.
- Fast search
- …

Terminology

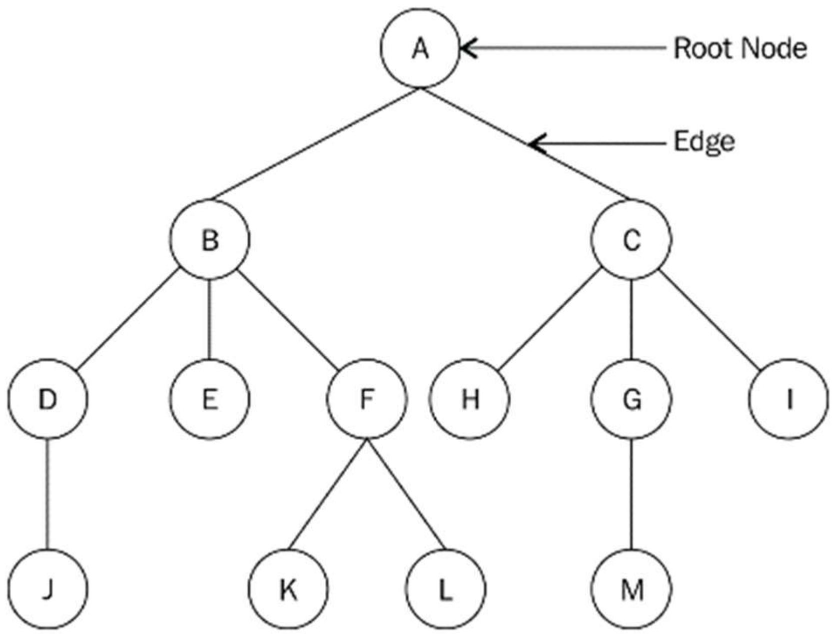
- To understand trees, we need to first understand the basic ideas on which they rest.

- Terms associated with a Tree:
- **Node:**
  - Each circled alphabet represents a node.
  - A node is any structure that holds data.
- **Root node:**
  - It is the node where the tree begins
  - It is the only node without a parent node.
  - Is the only node from which all other nodes come.
  - The root node in our tree is the node A.
- **Edge**:
  - The connection between two nodes.



Terminology (cont.)

- **Parent:**
  - A node in the tree with other connecting nodes is the parent of those nodes
  - Node B is the parent of nodes D, E, and F.



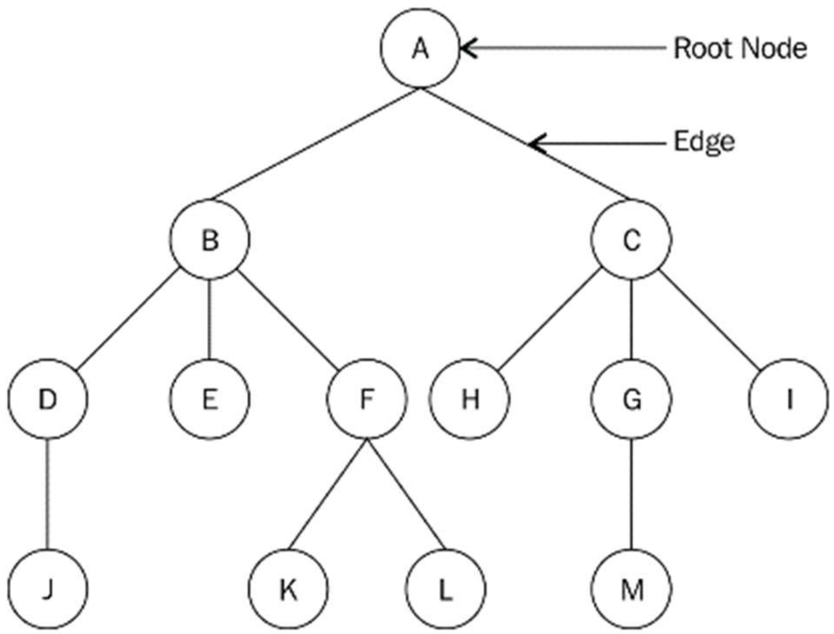
- **Child:**
  - This is a node connected to its parent.
  - Nodes B and C are children of node A, the parent and root node.

- **Sibling:**
  - All nodes with the same parent are siblings.
  - This makes the nodes B and C siblings.

Terminology (cont.)

- Degree:
  - Number of children of the node.
  - The degree of node A is 2.

- **Leaf node:**
  - This is a node with a degree of 0.
  - Nodes J, E, K, L, H, M, and I are all leaf nodes.



Binary trees

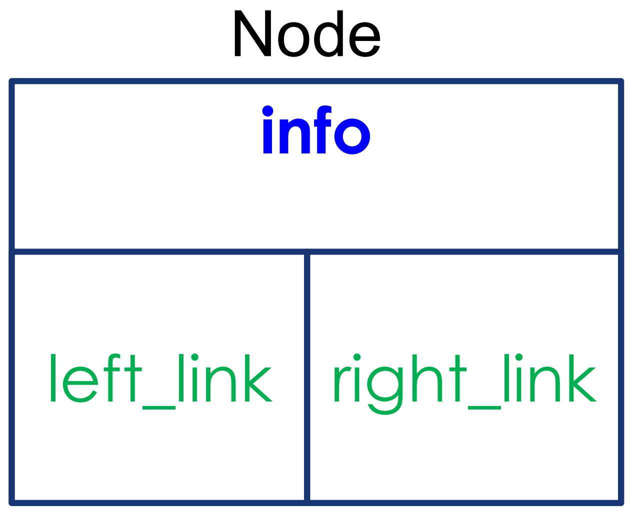
- Binary trees are a specific type of trees
- In a binary tree each node has a maximum of 2 children
  - They are designated as left child and right child

Tree nodes

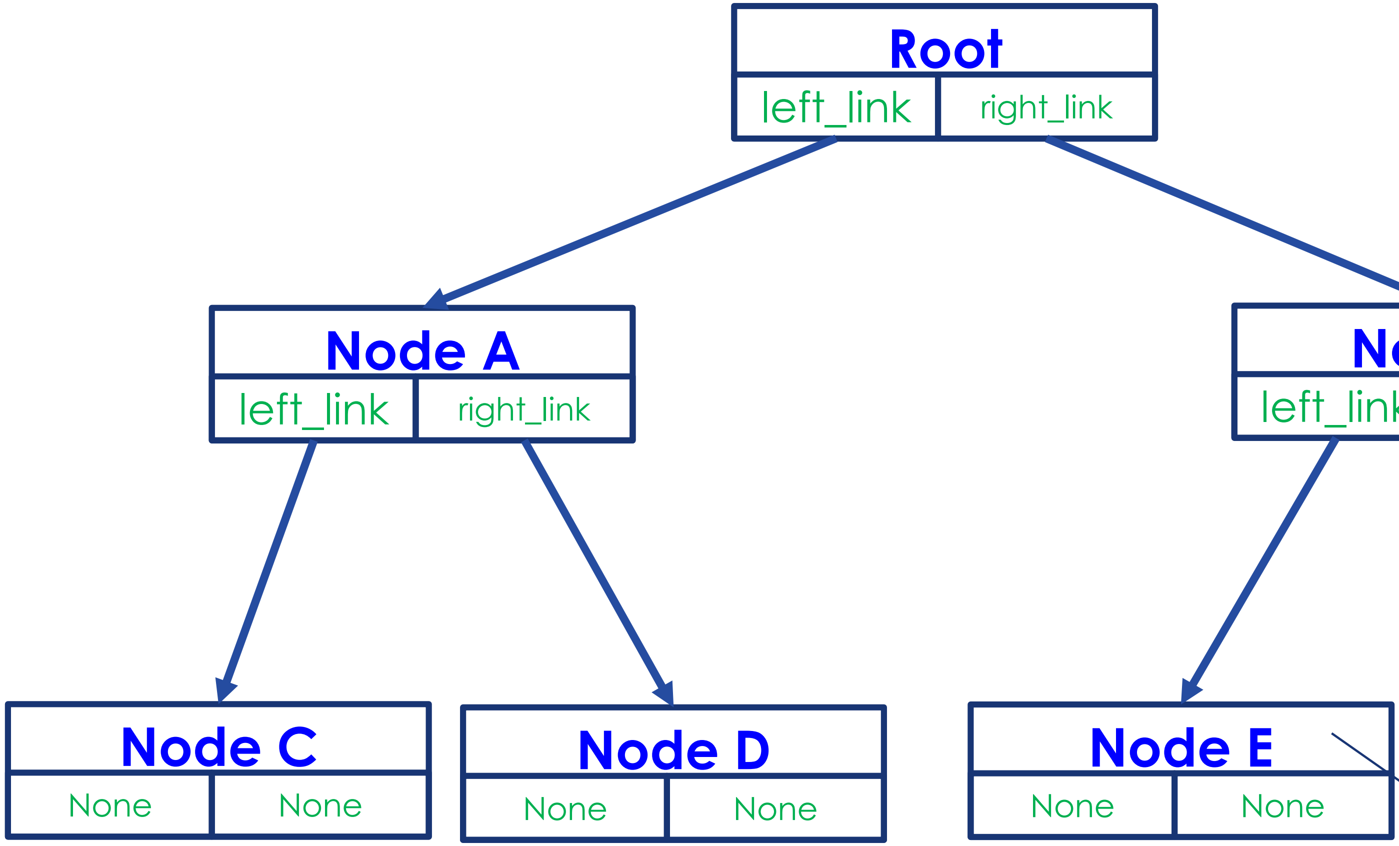
- Like Linked Lists, **trees are built up of nodes**.
  - But now the nodes that make up a tree need to contain data about the parent-child relationship

- A simple binary tree node class in Python:

```
class Node:
    def __init__(self, value):
        self.info = value
        self.right_link = None
        self.left_link = None
```



Example (binary tree)



A leaf node is a node whose left and right links have value None

binarytree library

- In Python, a binary tree can be represented in different ways with different data structures(dictionary, list) and class representation for a node.
- For example, binarytree library helps to directly implement a binary tree.
  - Binarytree Module in Python - GeeksforGeeks

- This module does not come pre-installed with Python's standard utility module.
  - To install it type the below command in the terminal.
  - pip install binarytree

Creating a simple tree

```python
from binarytree import Node
root = Node(3)
root.left = Node(6)
root.right = Node(8)
root.left.left = Node(10)

# Getting binary tree
print('Binary tree :', root)

# Getting list of nodes
print('List of nodes :', list(root))

# Getting inorder of nodes
print('Inorder of nodes :', root.inorder)

# Checking tree properties
print('Size of tree :', root.size)
print('Height of tree :', root.height)
```

```
Binary tree :
    3
   / \
  _6  8
 /
10

List of nodes : [Node(3), Node(6), Node(8), Node(10)]
Inorder of nodes : [Node(10), Node(6), Node(3), Node(8)]
Size of tree : 4
Height of tree : 2
```

(some) Tree operations

- Insertion

- Traversals
  - Since a binary tree is a non-linear data structure, there is more than one way to traverse through the tree data, including:

    inorder traversal

    preorder traversal

    postorder traversal.

Inorder Traversal

❖ In an inorder traversal, the **left** child is visited first, followed by the **parent** node, then followed by the **right** child.

```python
def inorder(node):
    if node:
      # call inorder on the left subtree until it reaches a leaf
      inorder(node.left)

      # Once we reach a leaf, we print the data
      print(node.val)  # binarytree node uses val

      # Now, since the left subtree and the root has been printed,
      #call inorder on right subtree until we reach a leaf node.
      inorder(node.right)
```

Example (inorder)

```
from binarytree import Node

root = Node(3)
root.left = Node(6)
root.right = Node(8)
root.left.left = Node(10)
root.left.right = Node(11)
root.right.left = Node(12)

print(root)
inorder(root)
```

```
    ___3___
   /       \
  _6       _8
 /  \      /
10   11   12

10
6
11
3
12
8
```

Code

Preorder

- In a preorder traversal, the **root node is visited first**, followed by the left child, then the right child.

```
def preorder(node):
    if node:
        # Print the value of the root node first
        print(node.val)

        # Recursively call preorder on the left subtree
        preorder(node.left)

        # Recursively call preorder on the right subtree
        preorder(node.right)
```

Example (preorder)

```
# For the tree,
#          10
#        /    \
#      34      89
#    /    \  /    \
#  20     45  56    54

# Preorder traversal: 10 34 20 45 89 56 54
```

Postorder

- In a postorder traversal, the left child is visited first, followed by the right child, then the root node.

```
def postorder(node):
    if node:
        # call postorder on the left subtree
        postorder(node.left)

        # call postorder on the right subtree
        postorder(node.right)

       # Print the value of the root node
        print(node.val)
```

Example (postorder)

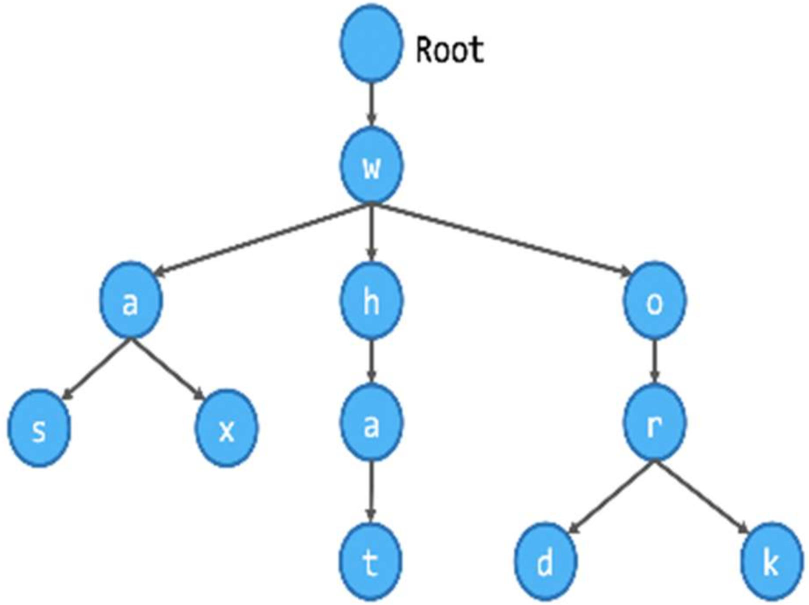
```
# For the tree,
#          10
#        /    \
#      34      89
#     /   \  /    \
#   20    45  56    54

# Postorder traversal: 20 45 34 56 54 89 10
```

Tries

It is not a typo ☺

Trie

- It is a tree-like data structure made up of nodes.
- Each node may have none, one or more children.
- When used to store a vocabulary, each node is used to store a character
  - consequently each "branch" of the trie represents a unique word.
- Example at right:
  - a trie with five words (was, wax, what, word, work) stored in it.



Applications

- Trie is a very useful data structure.
- It is commonly used to represent a dictionary for looking up words in a vocabulary.
- For example, consider the task of implementing a search bar with **auto-completion** or **query suggestion**.
  - When the user enters a query, the search bar will automatically suggest common queries **starting with** the characters input by the user.

How does a Trie Work?

- There are two major operations that can be performed on a trie, namely:

  - **Inserting** a word into the trie

  - **Searching** for words using a prefix

- Both operations involves traversing the trie by starting from the root node.

Advantages

- Fast access to the start of all words having specified initial characters
- Lower memory requirements
  - No need to repeat the characters

Implementation example

- You can find an implementation in Python at
  - Implementing Trie in Python | Albert Au Yeung

- It will be used in Practical Classes

Example of use

```
def main():

    trie = trie_from_file("words.txt")

    word = input('Beginning of the Word ?')
    while len (word) > 0:

        res = trie.query(word)

        for w in res:     # res is a list
            print(w[0])

        word = input('Beginning of the Word ?')
```

Graphs

Graphs

- Graphs have become a powerful means of modelling and capturing data in real-world scenarios such as social media networks, web pages and links, and locations and routes in GPS.

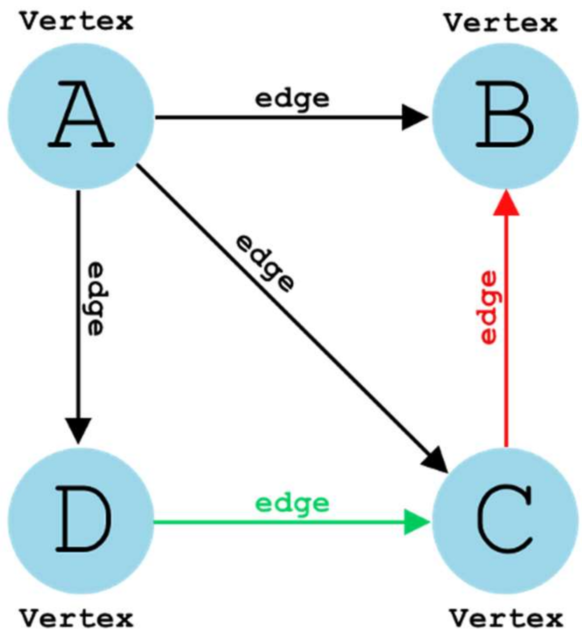
- If you have **a set of objects that are related to each other**, then you can represent them using a graph.

Graphs

❖ Are complex, non-linear data structures

❖ Are characterized by a group of vertices…

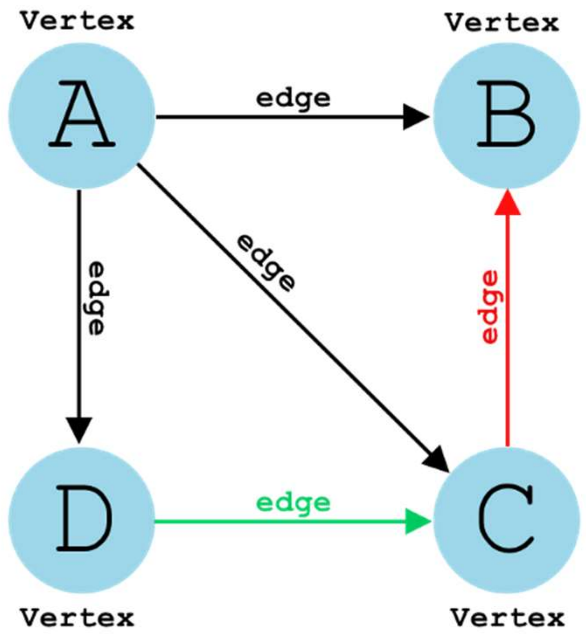
❖ … connected by edges.



❖ PT:
  - Vertices: Vértices (ou nós)
  - Edges: Ligações (ou arestas ou arcos)

Graphs

- **Vertices**
  - Represent entities in a graph.
  - Every vertex has a value associated with it.
  - Example: if we represent a list of cities using a graph, the vertices will represent the cities.



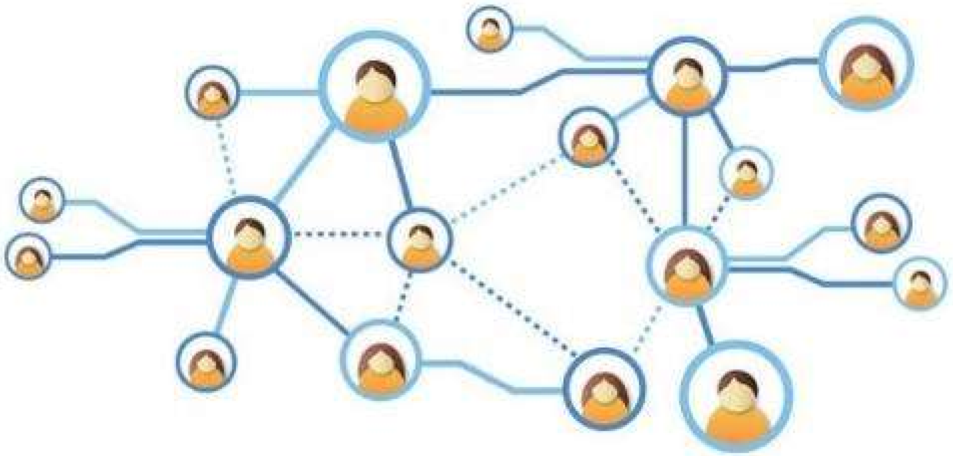
- **Edges**
  - Represent the relationship between the vertices in the graph.
  - Edges may or may not have a value associated with them.
  - Example: if we represent a list of cities using a graph, the edges will represent the path between the cities.

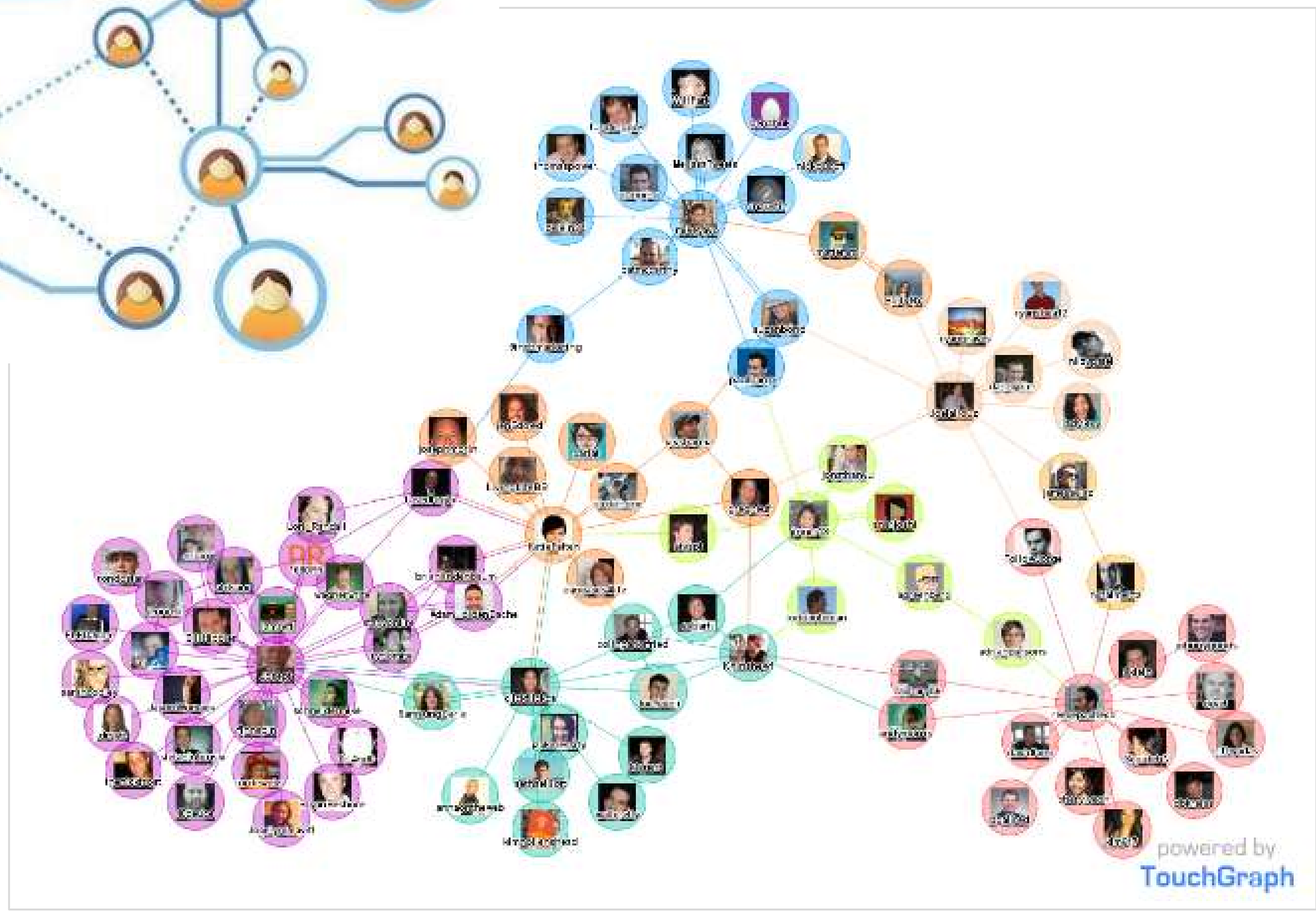
Applications of Graphs

- Graphs are used everywhere, from schooling to business.
  - Especially in the fields of computer science, physics, and chemistry.

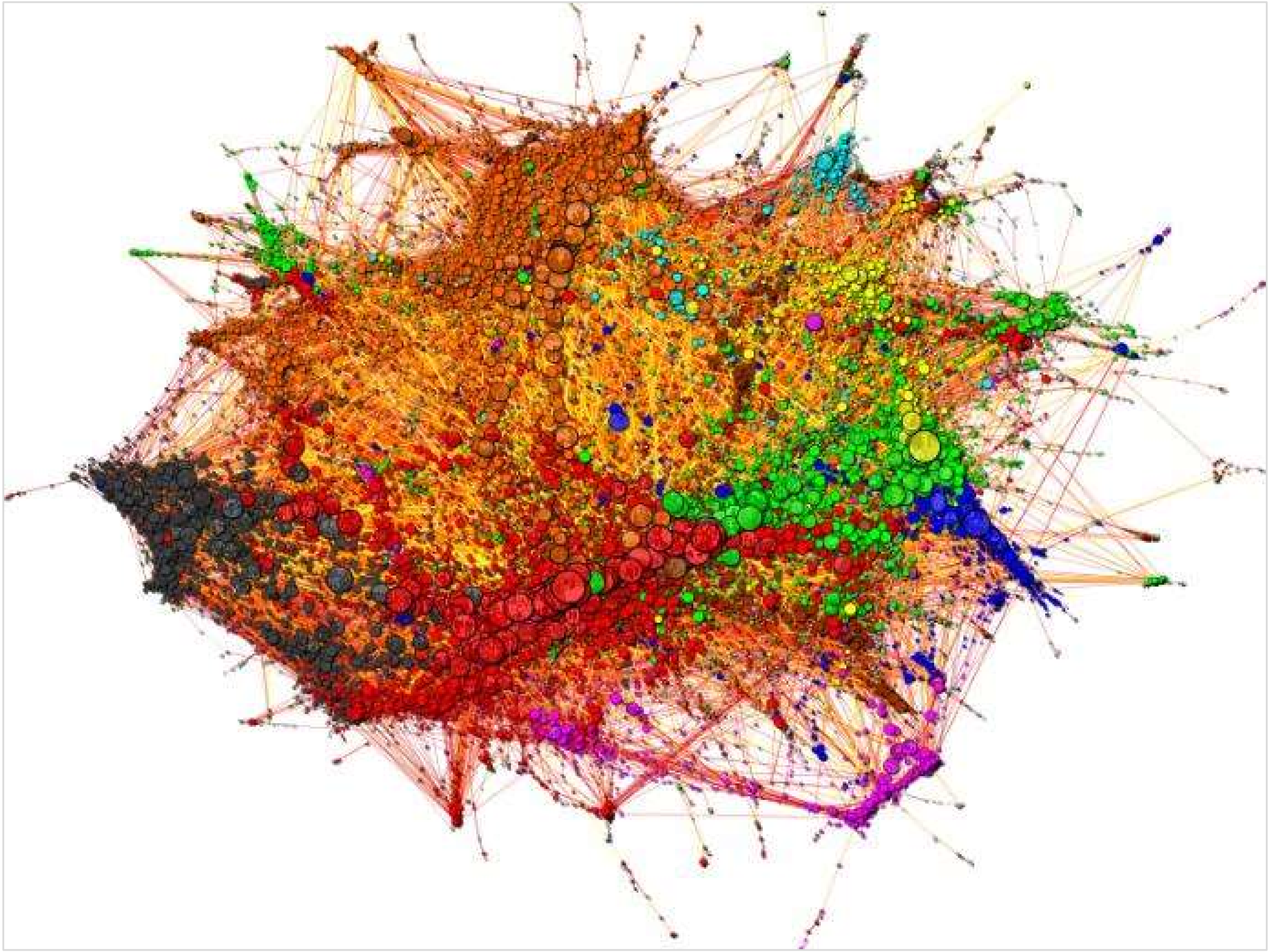
- A few other applications of graphs are:
  - To visualize organized data.
  - Directed Graphs are used in Google's Page Ranking Algorithm.
  - Social Networks use graphs to represent different users as vertices and edges to represent the connections between them.
  - In a mapping application, graphs are used to represent places and the path (distance) between them.

Applications of Graphs – Social Network

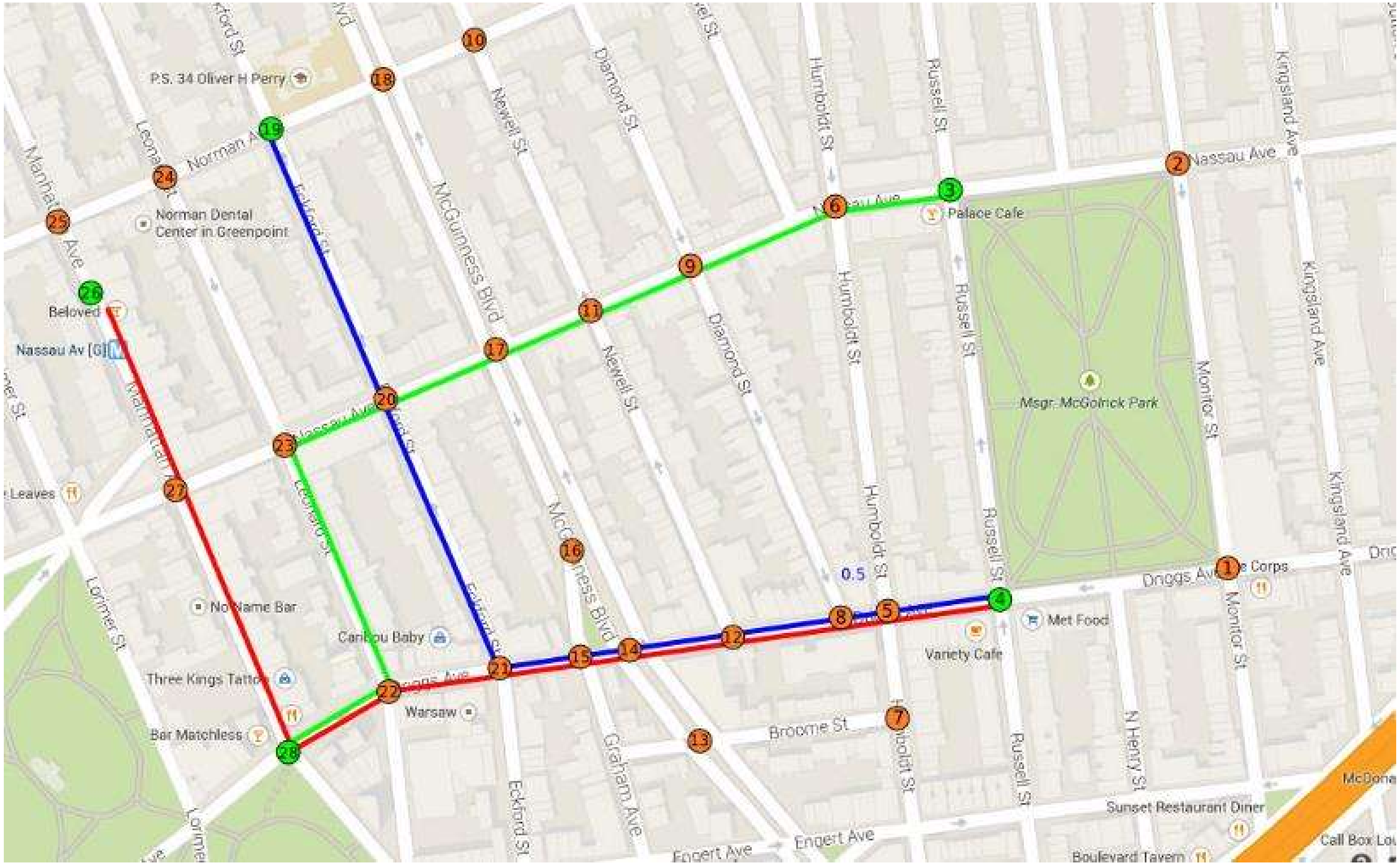




Applications of Graphs – Web



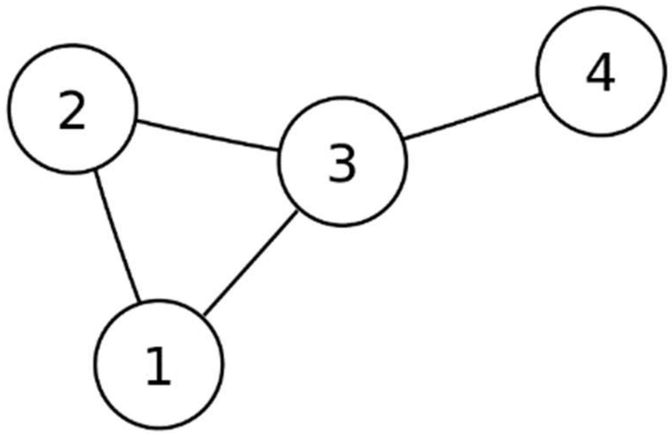
Applications of Graphs



Types of graphs

- There are many types of graphs, based on weights, direction, interconnectivity, and special properties:
- **Undirected Graphs**
  - In an undirected graph, the edges have no direction.
  - If there is a path from vertex X to vertex Y, then there is a path from vertex Y to vertex X. Edge (X, Y) represents the edge connecting vertex X to vertex Y.

    That is, edge (X, Y) == edge (Y, X)

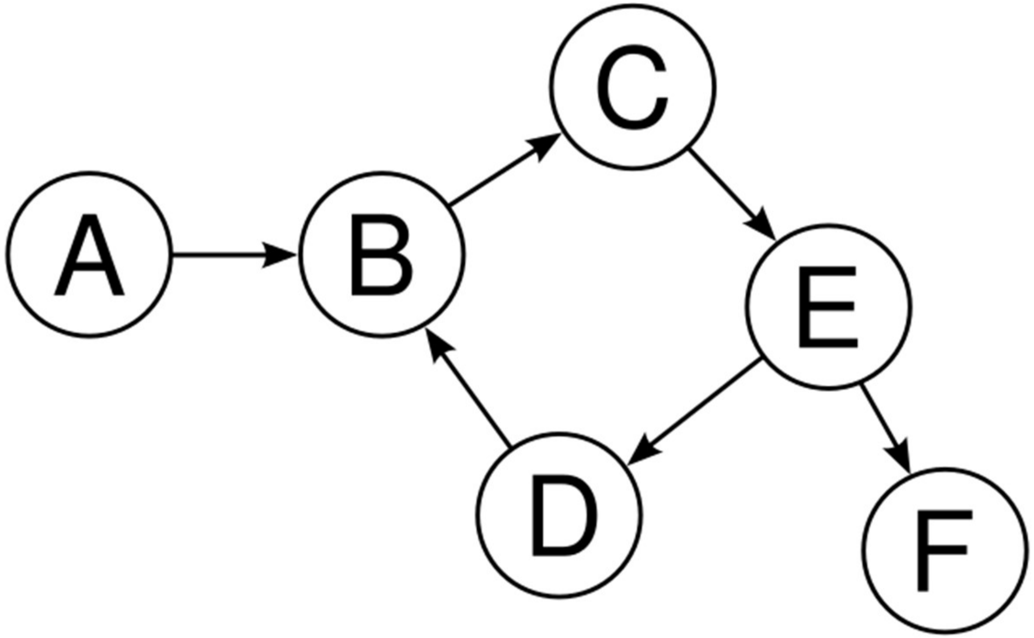


Types of graphs

❖ Directed Graphs

- In a directed graph or digraph, the edges have an orientation.
- If there is a path from vertex X to vertex Y, then there isn't necessarily a path from vertex Y to vertex X.

    That is, edge (X, Y) != edge (Y, X)



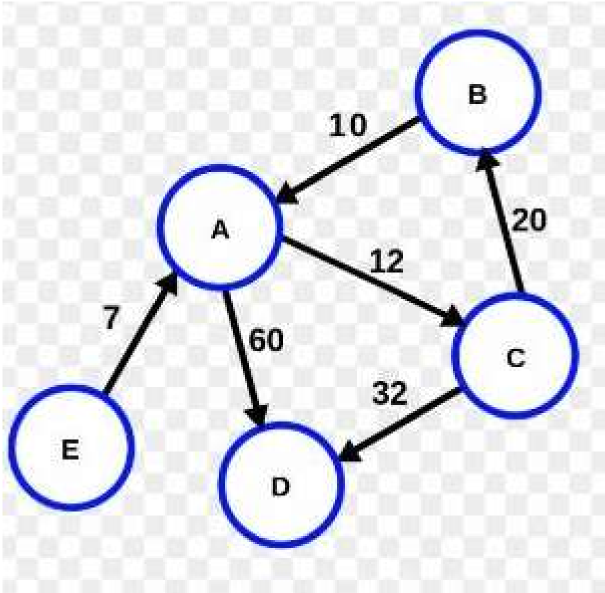
Types of graphs

- Weighted Graphs
  - A weighted graph has a value associated with every edge.
  - The value may represent quantities like cost, distance, time, etc., depending on the graph.
  - An edge of a weighted graph is represented as, (u, v, w).

    u -> Source vertex

    v -> Destination vertex

    w -> Weight associated to go from u to v.

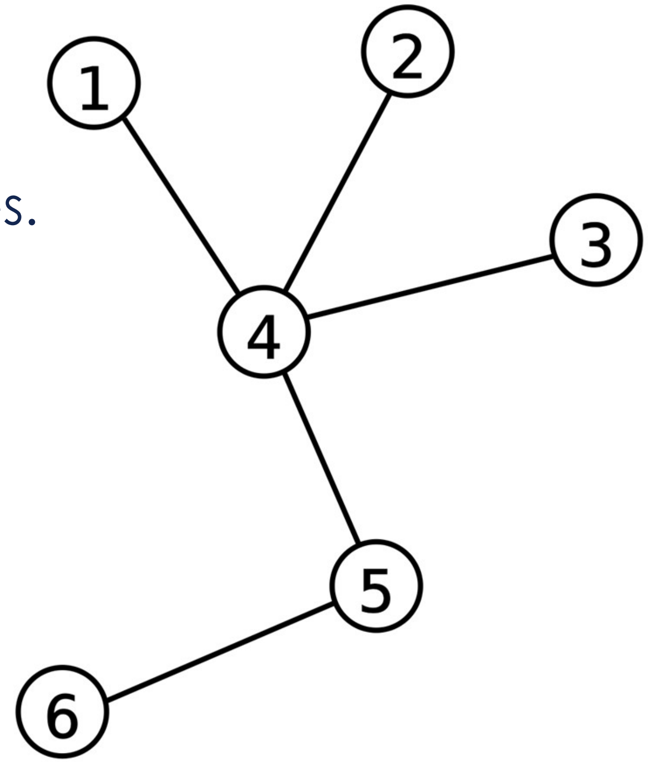


Special Graphs

- **Trees**
  - A Tree is a connected graph without cycles.

    A cycle in a graph is a sequence with the first and last vertices in the repeating sequence.

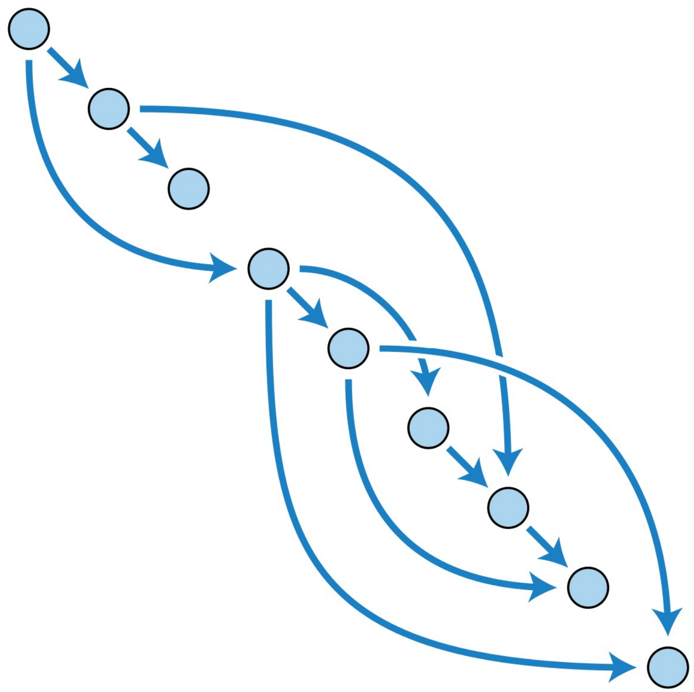
  - They have X vertices and X-1 edges.



Special Graphs

❖ Directed Acyclic Graphs

- Directed Acyclic Graphs or DAGs are graphs with no directed cycles.
- They represent structures with dependencies.

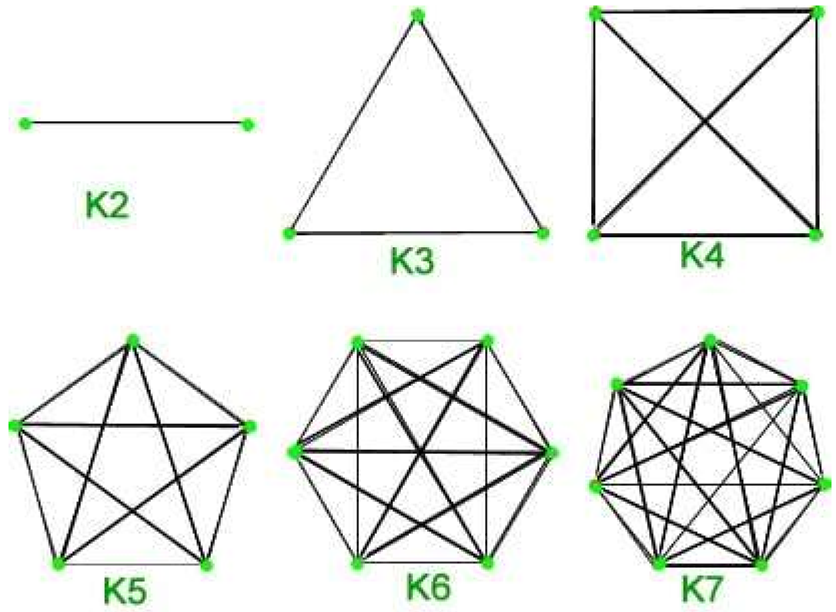


Special Graphs

❖ Complete Graphs

– Complete graphs have a unique edge between every pair of vertices.

– A complete graph n vertices have (n*(n-1)) / 2 edges and are represented by Kn.



Representing Graphs

- There are multiple ways of using data structures to represent a graph.

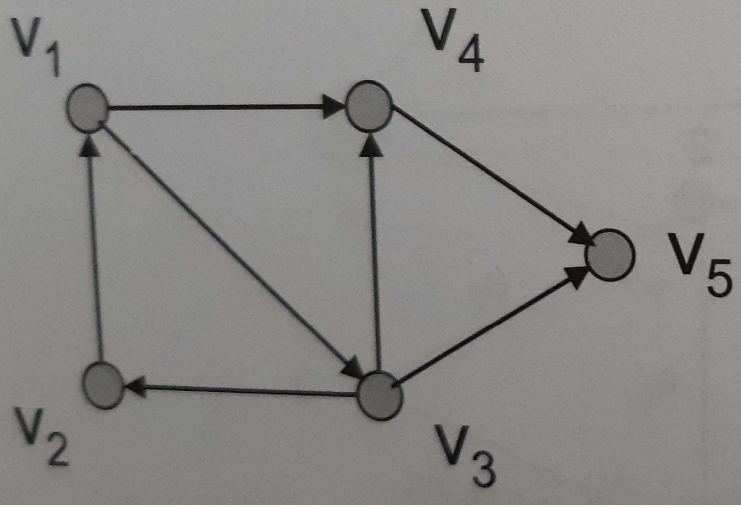
- The three most common ways are:
  - Adjacency Matrix
  - Adjacency List
  - Edge List

Adjacency Matrix

- It is a very simple way to represent a graph using a 2D array
- In a weighted graph, the element A[i][j] represents the cost of moving from vertex i to vertex j.
- In an unweighted graph, the element A[i][j] represents a Boolean value that determines if a path exists from vertex i to vertex j.
  - If A[i][j] == 0, then no path from vertex i to vertex j exists.
  - If A[i][j] == 1, there is a path from vertex i to vertex j.

- For example, a snake game can be represented by using an adjacency matrix.
  - This enables us to use various algorithms to find the shortest path to finish the game.
- Similarly, many shortest path algorithms use an adjacency matrix.

Example

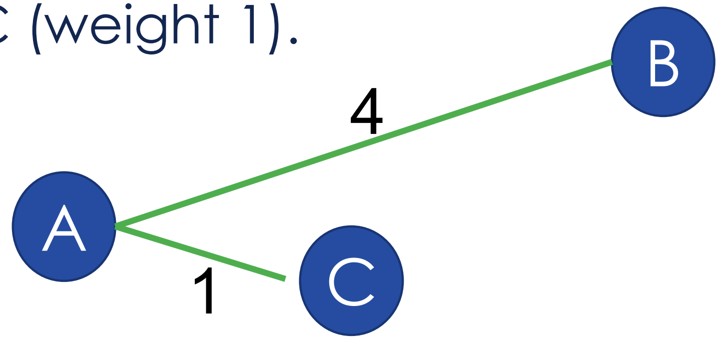


	v_1	v_2	v_3	v_4	v_5
v_1	0	0	1	1	0
v_2	1	0	0	0	0
v_3	0	1	0	1	1
v_4	0	0	0	0	1
v_5	0	0	0	0	0

Adjacency List

- An adjacency list represents a graph as a list that has vertex-edge mappings.

- Example:
  - A → [(B, 4), (C, 1)]
  - represents an adjacency list where the vertex A is connected to B (weight 4) and C (weight 1).



- This works really well for sparse graphs.

Edge list

- An edge list represents the graph as an unstructured list of edges.

- Example:

  graph = [(C, A, 4), (A, C, 1), (B, C, 6),
  (A, B, 4), (C, B, 1), (C, D, 2)]

Algorithms for Graphs

- Traversing is one of the fundamental operations that can be performed on graphs.
  - Needed, for example, to display the graph and search

- In next slides the most relevant are Visually explained

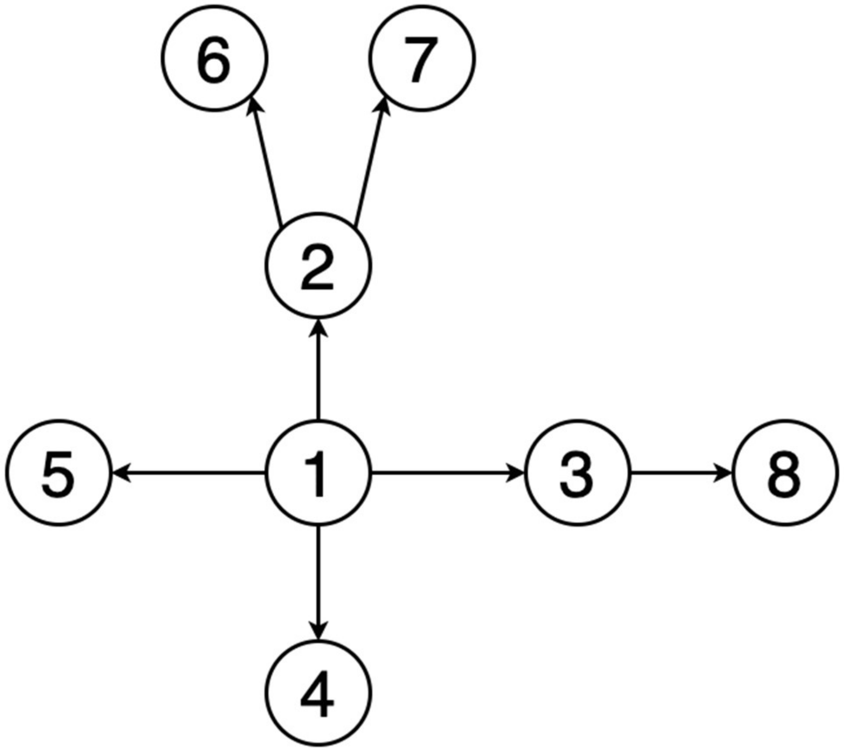
Breadth-first search (BFS)

- In breadth-first search (BFS), we start at a particular vertex and explore all its neighbors at the present depth before moving on to the vertices in the next level.

- Unlike trees, graphs can contain cycles

  (a path where the first and last vertices are the same)

- Hence, we must keep track of the visited vertices.
  - a queue is used.



- Note how vertices are discovered (yellow) and get visited (red).

Applications of BFS

- Used to determine the shortest paths and minimum spanning trees.
- Used by search engine crawlers to build indexes of web pages.
- Used to search on social networks.
- Used to find available neighbor nodes in peer-to-peer networks such as BitTorrent.

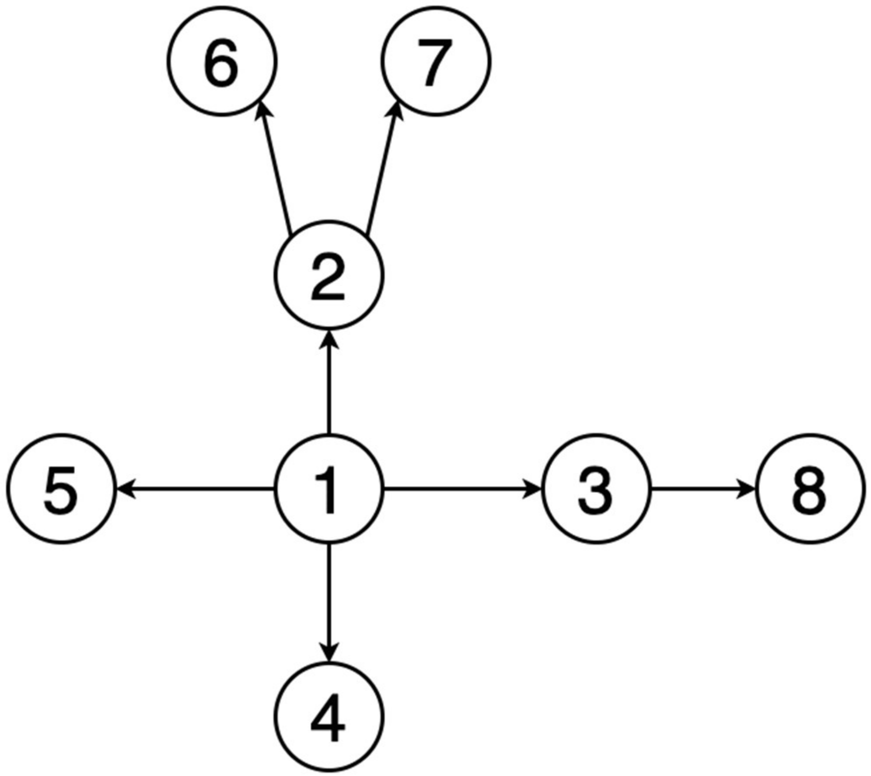
Depth-first search (DFS)

- In depth-first search (DFS) we start from a particular vertex and explore as far as possible along each branch before retracing back

  (backtracking)

- In DFS also we must keep track of the visited vertices.

- When implementing DFS, we use a stack to support backtracking.

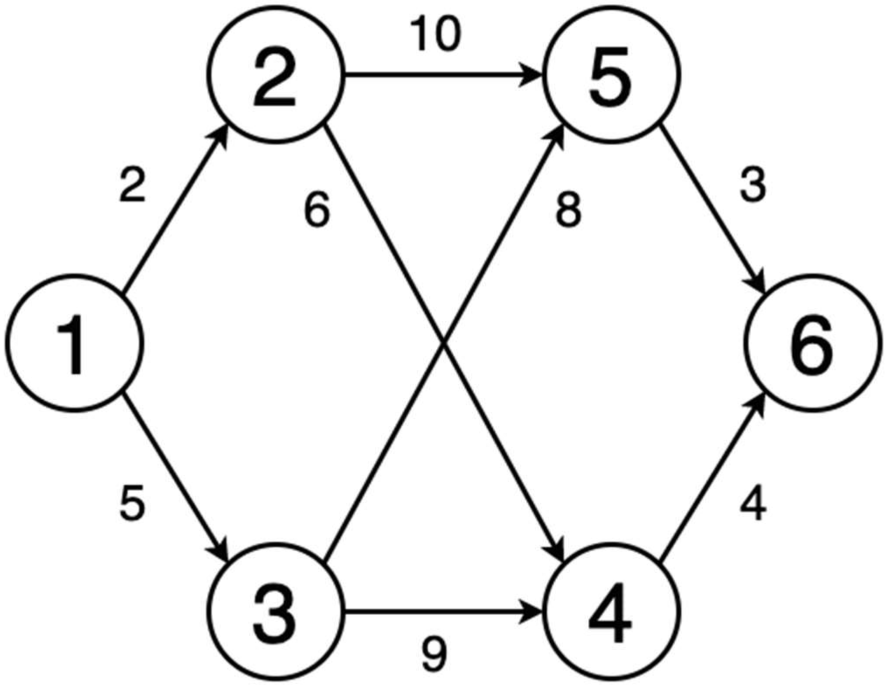


- Note how it traverses to the depths and backtracks.

Applications of DFS

- Used to find a path between two vertices.
- Used to detect cycles in a graph.
- Used in topological sorting.
- Used to solve puzzles having only one solution
  - e.g., mazes

Shortest path



- The **shortest path** is a path in the graph such that the sum of the weights of the edges that should be travelled is minimum.
- In the animation the shortest path from vertex 1 to vertex 6 is determined

Shortest path (cont.)

- Algorithms
  - Dijkstra's shortest path algorithm
  - Bellman–Ford algorithm

- Applications
  - Used to find directions to travel from one location to another in mapping software like Google maps or Apple maps.
  - Used in networking to solve the min-delay path problem.
  - Used in abstract machines to determine the choices to reach a certain goal state via transitioning among different states

    e.g., can be used to determine the minimum possible number of moves to win a game

(Example of) Modules for graphs

- **NetworkX**
  - Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.
  - https://networkx.org/
  - Data structures for graphs, digraphs, and multigraphs
  - Many standard graph algorithms

- Python-**igraph**
  - It is a library for creating, manipulating and analyzing graphs.
  - It is intended to be as powerful (i.e. fast) as possible to enable working with large graphs
  - https://igraph.org/python/

Examples using NetworkX

- use of **Dijkstra's algorithm** to find the shortest weighted path:

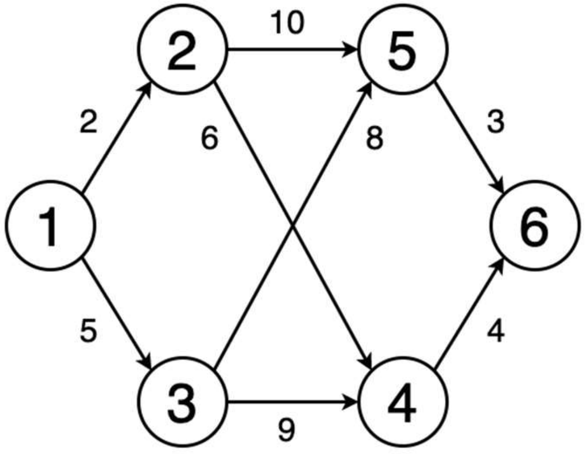
```
>>> G = nx.Graph()
>>> e = [('a', 'b', 0.3), ('b', 'c', 0.9), ('a', 'c', 0.5), ('c', 'd', 1.2)]
>>> G.add_weighted_edges_from(e)
>>> print(nx.dijkstra_path(G, 'a', 'd'))
['a', 'c', 'd']
```

  – Note the use of class Graph

- Drawing graphs

```
>>> import matplotlib.pyplot as plt
>>> G = nx.cubical_graph()
>>> subax1 = plt.subplot(121)
>>> nx.draw(G)    # default spring_layout
>>> subax2 = plt.subplot(122)
>>> nx.draw(G, pos=nx.circular_layout(G), node_color='r', edge_color='b')
```

Example 2 - Shortest path




```python
import networkx as nx

g = nx.Graph()
e = [ ('1', '2', 2),
      ('1', '3', 5),
      ('2', '4', 6),
      ('3', '5', 8),
      ('2', '5', 10),
      ('3', '4', 9),
      ('5', '6', 3),
      ('4', '6', 4)]

g.add_weighted_edges_from(e)

print(nx.dijkstra_path(g, '1', '6'))
```

Result:

['1', '2', '4', '6']

More information

- **10 Graph Algorithms Visually Explained**
  - A quick introduction to 10 basic graph algorithms with examples and visualisations
  - https://towardsdatascience.com/10-graph-algorithms-visually-explained-e57faa1336f3
- You can check out the implementations of graph algorithms found in the **networkx** and **igraph** python modules.
  - Python- igraph manual :

    https://igraph.org/python/tutorial/latest/tutorial.html
- You can read about python-igraph in article Newbies Guide to Python-igraph.

Object Orient Analysis, Design and Programming

Basics