**Graphs**

martes, 10 de octubre de 2023

8:29 p. m.

Graphs

The Roadmap Intro: [Graph Data Structure | Illustrated Data Structures](https://www.youtube.com/watch?v=0sQE8zKhad0)

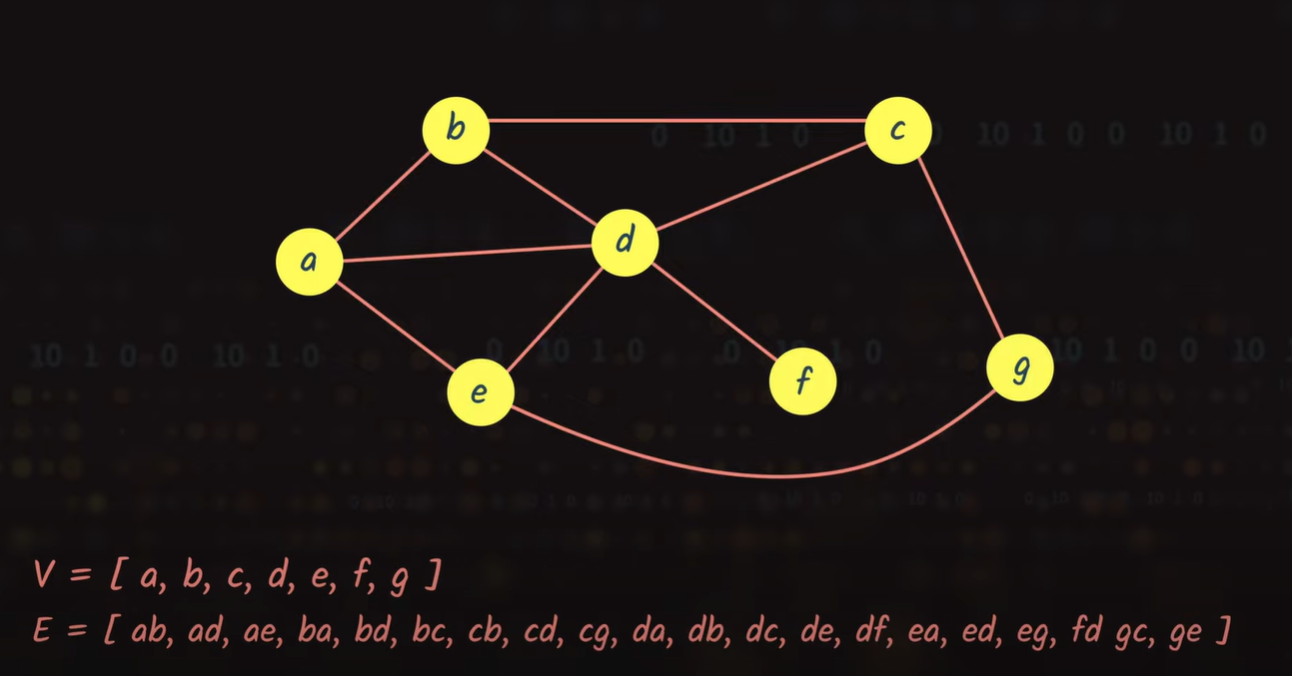
Geeks for geeks YT course ref: [Graph | Data Structures & Algorithms | Programming Tutorials | GeeksforGeeks](https://www.youtube.com/playlist?list=PLqM7alHXFySEaZgcg7uRYJFBnYMLti-nh)

ορτρ STRlJCTlJRES 
GRAPH 
EVERYTHlNG γου NEEO ΤΟ KNOW 

Graph's generic structure

Node aka Vertex / Edges 
GRAPH DATA STRUCTURE 

Graphs have, like trees, nodes and edges, but in a graph like structure nodes are called vertices.



Types of Graphs

Graphs can be either directed or undirected. The case below is from an undirected graph, in which is possible to go from a to b and back, and the same goes for the rest of the vertices. Kind of like a doubly linked list.

UNDIRECTED 
E = [ ab, ad, ae, ba, bd, bc, cb, cd, c9, da, db, dc, de, df, ea, ed, eg, fd 9c, ge J 

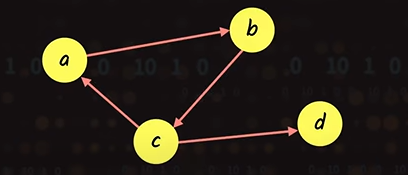
And edges can algo be undirected, meaning that from each vertex is possible to go only in the direction specified

b 
DIRECTED 
Directions assigned to edges. 

Graph's terminology

Path: A set of edges that connect a set of nodes. Paths can be either Closed or Simple.

Path from a to d: [a, b, c, d]



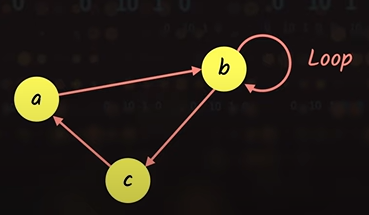
A Closed Path is a path in which the starting and ending node are the same.

A close path would be: [a, b, c, a ]

a Closed Path is a path in which no node is repeated.

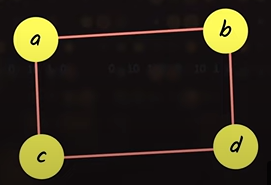
A simple path would be: [b, c, d ]

Loop: An edge that connects a vertex to itself.

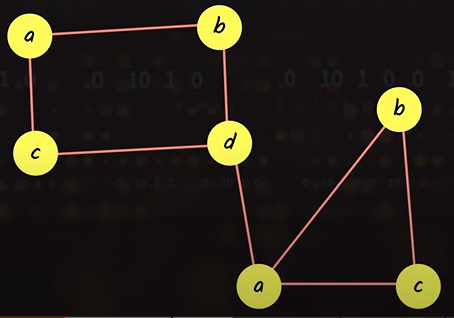


Degree of Node: Number of edges connecting a node to graph. And specifically to Undirected and Directed graphs, varies the definition. For Undirected, the degree of a node, is simply the number of edges connecting the node to the graphs, but if is Directed, there are two types of degrees: In Degree (The number of edges coming into the node) and Out Degree (The number of edge coming out of the node).

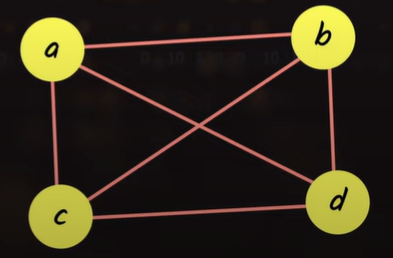
Cycle Graph: A Graph with a number of nodes that is equal to the number of edges.



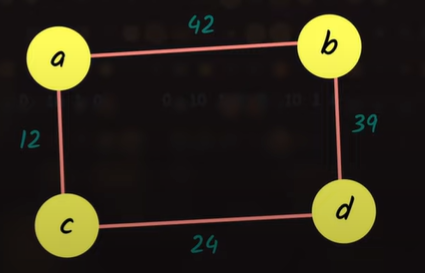
Connected Graph: A Graph in which any node can be visited from any other node.



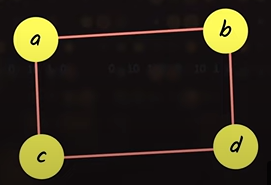
Completed Graph: A Graph in which any node is connected to any other node with exactly one edge.



Weighted Graph: A Graph in which each edge has a value (weight) assign to it.



Simple Graph: A Graph having no loops nor parallel edges.



Graph Programmatic Representation

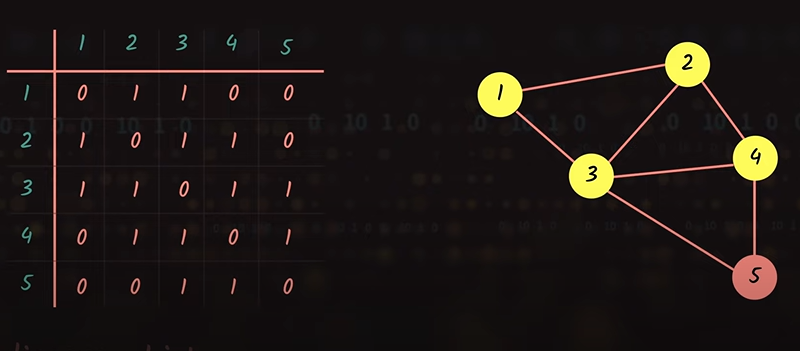
A Graph can be represented eight with an Adjacency Matrix or an Adjacency List.

Adjacency Matrix

A 2D array or a Hash Map to represent the graph.

This is a Binary Matrix in which 1 represents an edge and 0 the absence of it.

One special case is a Weighted Graph, in which the values in the matrix are the weight of the edge or infinity (∞) if there is no edge .

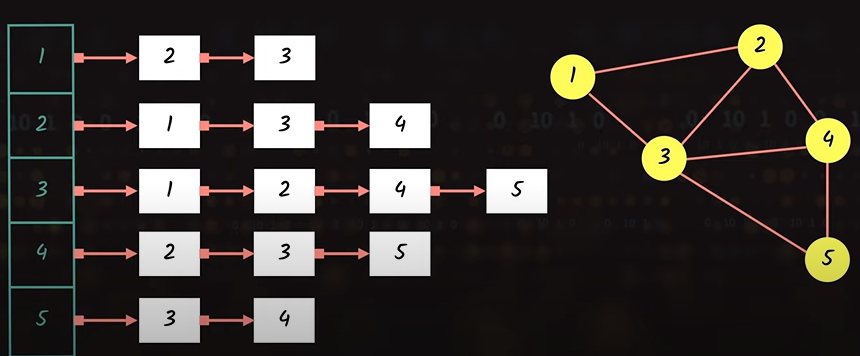


Complexities of Adjacency Matrix Representation Operations

* Removal of an edge can be done in O(1)
* Addition of an edge can be done in O(1)
* Edges can be queried in O(1)
* Removal of vertex has O(n²)
* Addition of vertex has O(n²)
* Space complexity is quadratic O(n²)

Adjacency List

Use an array of Linked List to represent the graph.



Complexities of Adjacency List Representation Operations

* Adding a vertex takes O(1)
* Adding an edge takes O(1)
* Removing a vertex takes O(n)
* Removing an edge takes O(n)
* Querying is O(n)
* Space complexity is O(n)

Strengths and Weaknesses

**Strengths**

**Versatility:** Graphs can represent a wide range of relationships and structures, making them applicable in numerous domains, including social networks, transportation systems, recommendation systems, and more.

**Complex Relationships:** They can model complex relationships and dependencies, making them suitable for representing real-world scenarios where entities have intricate interactions.

**Data Analysis:** Graphs are essential for network analysis, social network analysis, and various data science applications. They help discover patterns, communities, and anomalies in data.

**Efficient Algorithms:** Many graph algorithms are available to solve specific problems efficiently, such as shortest path finding, network flow, and matching problems.

**Hierarchical Structures:** Graphs can represent hierarchical structures, such as organizational charts or file systems, making them valuable in data management.

**AI and Machine Learning:** Graph neural networks and other graph-based techniques are increasingly important in artificial intelligence and machine learning for tasks like recommendation, knowledge graph reasoning, and natural language processing.

**Weaknesses**

**Complexity:** While graphs are powerful, they can be complex to work with, especially in terms of algorithm design and analysis. Some graph problems are NP-hard and computationally intensive.

**Storage and Memory:** Storing large graphs can consume a significant amount of memory. This can be a concern in applications dealing with massive datasets.

**Traversal Overhead:** Traversing graphs can be time-consuming, and the efficiency of traversal depends on the graph's structure and the chosen algorithm.

**Complexity of Implementation:** Implementing graph algorithms and data structures can be challenging, and errors in code can be difficult to spot.

**Practical Use Cases:** While graphs have numerous applications, they are not always the best choice for every problem. In some cases, simpler data structures may be more efficient.

**Scalability:** Graph algorithms and data structures may not scale well for extremely large graphs, requiring specialized techniques and hardware to handle such cases.

Graphs Use Cases

0 
008 
GPS 
SOCIAL 
SYSTEMS 
NETWORKS 
SEARCH 
ENGINES 

GPS Systems: To find the shortest path between two locations.

Social Networks: To represent social connections.

Search Engines: To rank the result to show them in order.

Chat GPT's recommendation to study regarding graphs

1. ~~Graph Fundamentals:~~
   * ~~Types of graphs: directed, undirected, weighted, unweighted.~~
   * ~~Basic terminology: nodes/vertices, edges, neighbors, degree, etc.~~

1. ~~Graph Representation:~~
   * ~~Adjacency matrix~~
   * ~~Adjacency list~~
   * ~~Incidence matrix (for bipartite graphs)~~
   * ~~Graph data structures in programming (e.g., dictionaries, lists, sets)~~

1. ~~Graph Traversals:~~
   * ~~Breadth-First Search (BFS)~~
   * ~~Depth-First Search (DFS)~~

1. ~~Shortest Path Algorithms:~~
   * ~~Dijkstra's algorithm~~
   * ~~Bellman-Ford algorithm~~
   * ~~A\* algorithm (useful in AI applications)~~

1. Minimum Spanning Tree:
   * Prim's algorithm
   * Kruskal's algorithm

1. Graph Algorithms in AI and Data Science:
   * Graph-based data structures for various applications.
   * Network analysis and centrality measures (e.g., betweenness centrality, closeness centrality).
   * Graph-based algorithms for recommendation systems.
   * Graph-based machine learning algorithms (e.g., graph neural networks).

1. Real-world Applications:
   * Study real-world use cases of graphs in data science and AI, such as social network analysis, recommendation systems, and natural language processing.

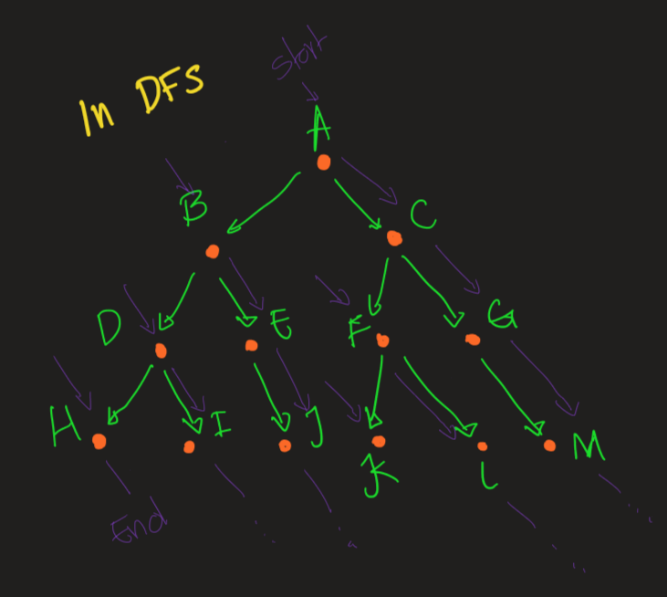
1. Graph Libraries and Tools:
   * Familiarize yourself with graph libraries in your preferred programming language, such as NetworkX for Python.

1. Practice and Projects:
   * Implement various graph algorithms and work on projects that involve graph analysis. Practical experience is essential for a deep understanding.

Graphs Traversal

There are several ways to traverse graphs but, initially there are two approaches: Depth First and Breadth first.

Depth-First Search (DFS): DFS explores as far down a branch as possible before backtracking. It can be implemented using either recursion or an explicit stack data structure.



Key characteristics of DFS

* DFS can be implemented using recursion or an explicit stack data structure.

* It **does not** guarantee the shortest path between nodes, making it unsuitable for certain path-finding problems.

* DFS is memory efficient when dealing with deep graphs or trees, as it only needs to store information about the current path being explored.

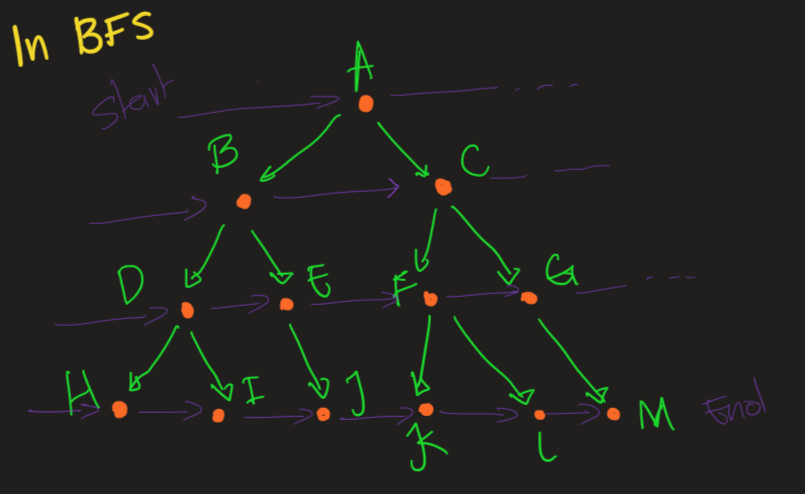
* It's well-suited for problems involving cycle detection, topological sorting in directed acyclic graphs (DAGs), and exploring all possibilities in a search tree.

* DFS can handle complex problems by exploring and then reversing decisions (backtracking) when needed.

Complexity T: O(V+E) / S: O(V)

ref: [Depth-first search in 4 minutes](https://www.youtube.com/watch?v=Urx87-NMm6c)

Breadth-First Search (BFS): BFS explores all neighbors of a node before moving on to their children. It uses a queue data structure to maintain the order of exploration.



Key characteristics of BFS

* BFS guarantees that it will find the shortest path to all reachable nodes from the source node in an unweighted graph.

* It's typically implemented using a queue or a deque data structure to maintain the order of exploration.

* BFS is often used for tasks like finding the shortest path, solving puzzles and mazes, connectivity analysis, and network traversal.

* It is not well-suited for graphs with very deep levels, as it can consume significant memory when exploring deeply before backtracking.

Complexity T: O(V+E) / S: O(V)

ref: [Breadth-first search in 4 minutes](https://www.youtube.com/watch?v=HZ5YTanv5QE)

Other ways to traverse

Topological Sort

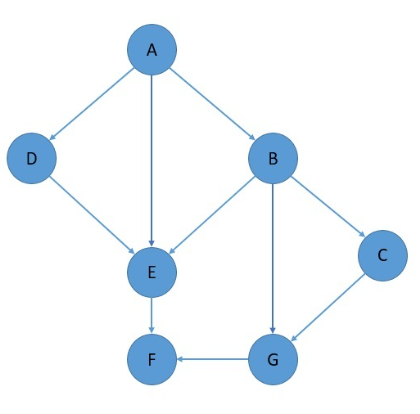
This specific kind of sorting takes the spirit of the DFS (Meaning that go mainly in depth) to build the resulting sequence.

Topological sorting is a technique used to order the vertices of a directed acyclic graph (DAG) in such a way that for every directed edge (u, v), vertex u comes before vertex v in the ordering. In simpler terms, it is a way to linearly order the nodes of a directed graph in a way that all dependencies are satisfied. This is commonly used in various applications, including build systems, task scheduling, pre-requisites tasks management and resolving dependencies in a directed acyclic graph.

Topological sorting is a fundamental concept in computer science and is essential in various applications where you need to establish a clear order of execution or dependency resolution. If the graph has cycles, it's not possible to perform a topological sort, as the dependencies become ambiguous. Therefore, it's crucial to ensure that the graph is a DAG before attempting a topological sort.

ref: [Topological Sort Visualized and Explained](https://www.youtube.com/watch?v=7J3GadLzydI)

DAG Example



Dijkstra's Algorithm

Dijkstra's algorithm is a graph search algorithm that is used to find the shortest path between nodes in a graph, particularly in graphs with non-negative edge weights. It was developed by Dutch computer scientist Edsger W. Dijkstra in 1956.

Dijkstra's algorithm guarantees the shortest path for each node as long as the edge weights are non-negative. It is commonly used in various applications, such as network routing, GPS navigation, and finding the shortest path in a map or a transportation network. However, it may not work correctly with graphs containing negative edge weights, and for such cases, other algorithms like the Bellman-Ford algorithm are more appropriate.

Both BFS and Dijkstra's follow the Breadth First traversal pattern.

ref: [Dijkstra's Algorithm Explained and Implemented in Java | Shortest Path | Graph Theory | Geekific](https://www.youtube.com/watch?v=BuvKtCh0SKk)

A\* Search\*

A\* is another algorithm used to find the shortest path, often used in applications like pathfinding in games or GPS navigation. It combines elements of Dijkstra's algorithm and heuristics to prioritize nodes likely to lead to the goal.

Note: When the heuristic function is set to return 0, the algorithm would behave just like Dijkstra's.

ref: [A Star algorithm | Example | Informed search | Artificial intelligence | Lec-21 | Bhanu Priya](https://www.youtube.com/watch?v=PzEWHH2v3TE)

Bellman-Ford Algorithm

The Bellman-Ford algorithm is a popular and widely used algorithm for finding the shortest path from a single source vertex to all other vertices in a weighted directed graph. It can handle graphs with negative edge weights, but it can detect and report negative weight cycles.

The Bellman-Ford algorithm is a versatile algorithm and is particularly useful when dealing with graphs that contain negative edge weights, which some other algorithms like Dijkstra's algorithm cannot handle without modification. However, it is important to note that the Bellman-Ford algorithm has a time complexity of O(V \* E), where V is the number of vertices and E is the number of edges. This makes it less efficient than Dijkstra's algorithm for graphs with non-negative edge weights, but it can handle a wider range of scenarios.

If a negative weight cycle is detected, the Bellman-Ford algorithm is often used to find the vertices within that cycle, which can be useful in various applications like network routing and detecting arbitrage opportunities in financial markets.

ref: [Bellman-Ford in 5 minutes — Step by step example](https://www.youtube.com/watch?v=obWXjtg0L64)

Iterative Deepening Depth-First Search (IDDFS)

IDDFS is a combination of two popular graph traversal techniques: Depth-First Search (DFS) and Breadth-First Search (BFS). It's often used to explore and search trees or graphs efficiently while consuming minimal memory.

IDDFS combines the benefits of both DFS and BFS: Like DFS, it explores as deeply as possible along each branch, making it memory-efficient because it doesn't store the entire search tree in memory. Like BFS, it ensures that shallower levels are explored before deeper levels. This means that it's guaranteed to find the shortest path to the target if one exists.

IDDFS is especially useful in situations where you have limited memory resources and want to find the shortest path in a tree or graph. It's worth noting that IDDFS can be less efficient than more specialized algorithms like A\* search in terms of time complexity, but it has the advantage of using less memory.

ref: [Iterative Deepening Search | IDS Search | DFS Algorithm in Artificial Intelligence by Mahesh Huddar](https://www.youtube.com/watch?v=BK8cEWKHCkY)

Bi-Directional Search

Bidirectional Search is a graph search algorithm that simultaneously explores the graph from two directions: one starting from the initial state (often denoted as the "start" state), and the other from the goal state. The primary goal of this approach is to reduce the overall search space and improve the efficiency of the search, especially in scenarios where traditional search algorithms might be computationally expensive.

The key advantage of Bidirectional Search is that it can potentially reduce the time and space complexity of the search, as it explores the graph from two directions and meets in the middle. This can be particularly beneficial in large graphs or networks.

However, it's essential to note that Bidirectional Search is not suitable for all scenarios. It works best when you have a well-defined initial state and a well-defined goal state and there is a relatively clear path between them. It's also important to consider the cost of verifying the overlap between the two search trees.

In summary, Bidirectional Search is a graph traversal technique that can be more efficient than traditional search algorithms in specific situations. It's often used in domains like route planning and certain artificial intelligence applications.

Graph used in the implementation

Gráfico, Gráfico radial

Descripción generada automáticamente

Randomized Search Algorithms

Some algorithms, like Random Walks, use randomness to explore the graph. These are often used in approximate solutions and sampling problems.