



Ask a Question



Introduction to Graphs

Let's go over the Graphs pattern, its real-world applications, and some problems we can solve with it.

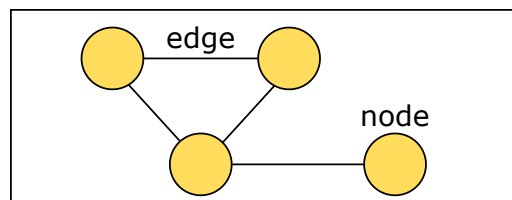
We'll cover the following



- About the pattern
- Examples
- Does your problem match this pattern?
- Real-world problems
- Strategy time!

About the pattern

A **graph** is a nonlinear data structure that represents connections between entities. In graph theory, entities are represented as vertices (or nodes), and the connections between them are expressed through edges.



Let's go over some basic components and common properties of graph






Name	Description
Vertex (node)	It is the fundamental unit of a graph, usually represented as a point or data.
Edge	It is a connection between two nodes. An edge may have a direction (directed edge) or not (undirected edge).
Weight	Each edge has a weight assigned to it, which indicates a cost or value of the connection.
Degree	It is the number of edges incident to a node.
In-degree	It is the number of edges coming towards the node (In directed graphs).
Out-degree	It is the number of edges going away from the node (In directed graphs).
Adjacent nodes	Those nodes that are directly connected to each other by an edge.
Path	It is a sequence of nodes where each adjacent pair is connected by an edge.
Cycle	It is a path that starts and ends at the same node.

Types of graphs

There are various types of graphs, each tailored to represent different relationships and scenarios. Let's explore the key types:

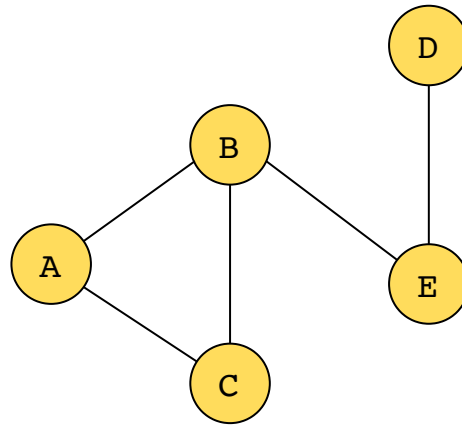
- **Undirected graph:** A graph in which the edges have no direction, representing a two-way relationship between nodes. For example, if we consider each person on Instagram as a node and their connection with other users as edges, a two-way relationship occurs when person A follows person B and person B also follows person A.



- 
- **Directed graph:** A graph in which the edges have a direction, indicating a one-way relationship between nodes. For example, if person A follows person B but person B doesn't follow person A, it's a one-way relationship.
 - **Weighted graph:** A graph in which each edge has a numerical value assigned to it, indicating the cost, distance, or some other relevant measure associated with that connection. For example, if each person's connection with the other person has some weight assigned to it, then the weight of 8 represents a strong connection between person A and person B and the weight of 3 indicates a weaker connection between person A and person C.
 - **Cyclic graph:** A graph that contains at least one cycle, which is a path that starts and ends at the same node. For example, person A is friends with person B, who is friends with person C, and finally, person C is friends with person A, completing the cycle.
 - **Acyclic graph:** A graph that contains no cycles, that is, there is no path that starts and ends at the same node. For example, person A is friends with person B, who is friends with person C, and finally, person C is friends with person D, not creating any cycle.

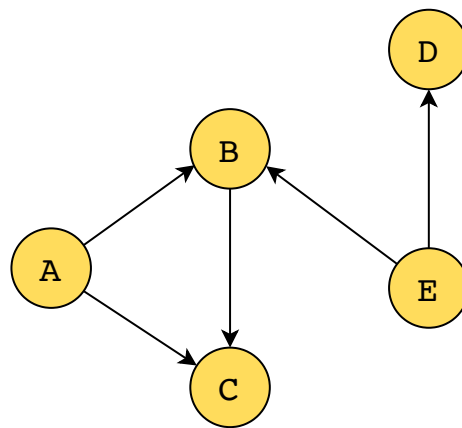
Here's the visual representation of each of the above-discussed graphs:





Undirected graph

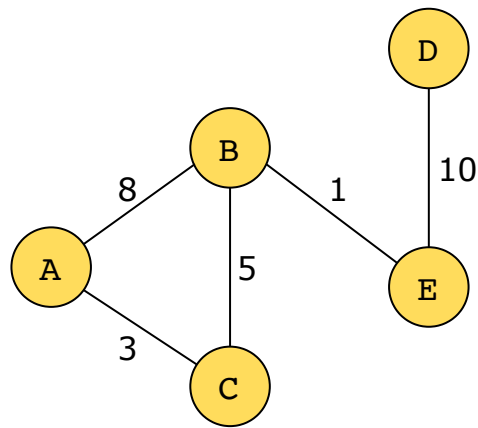
1 of 5



Directed graph

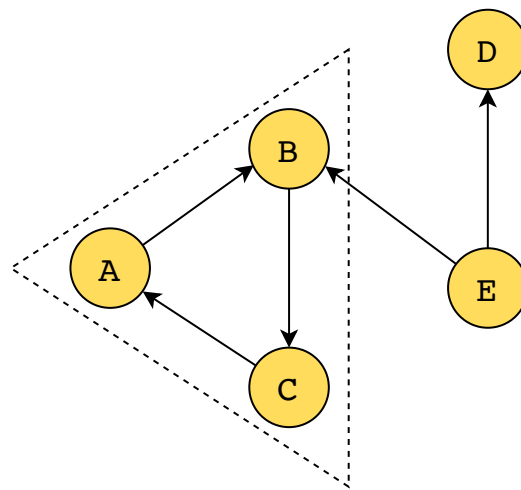
2 of 5





Weighted graph

3 of 5



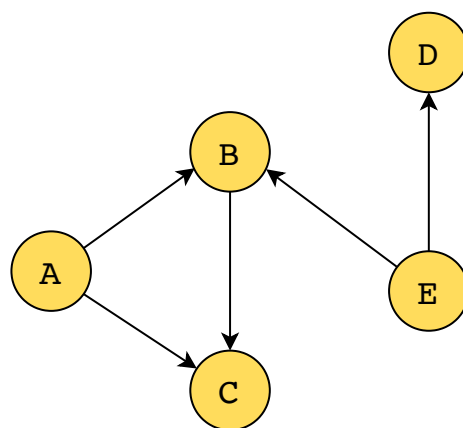
Cyclic graph

4 of 5

?

Tt





Acyclic graph

5 of 5

—

[]

Graph representation

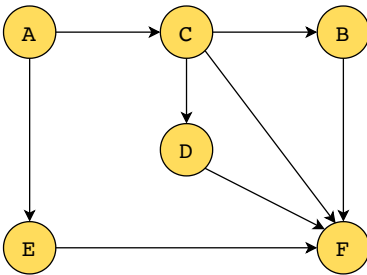
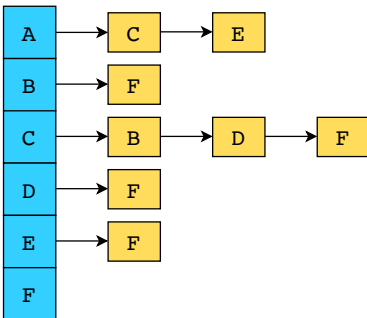
Graphs are usually represented as either an adjacency list or an adjacency matrix in order to solve graph-related problems.

- **Adjacency list:** A collection of lists, where each list corresponds to a node and contains its neighbors. In the case of weighted graphs, the weight is stored along with the corresponding nodes. Otherwise, a default weight of 1 is assumed and is often omitted from the lists.
- **Adjacency matrix:** A 2D array where each cell, $matrix[i][j]$, represents the edge between nodes i and j . The value of $matrix[i][j]$ equals 1 (or the weight in the case of a weighted graph) if there's an edge between nodes i and j , and 0 otherwise.

?

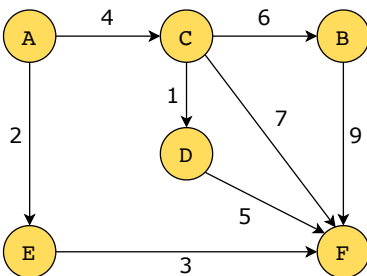
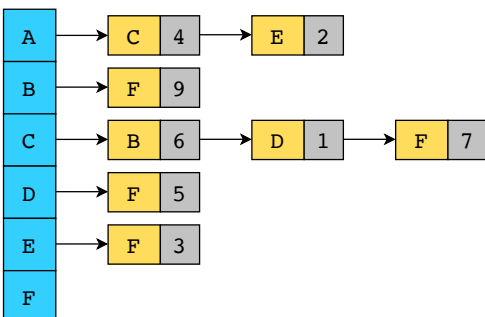
TT



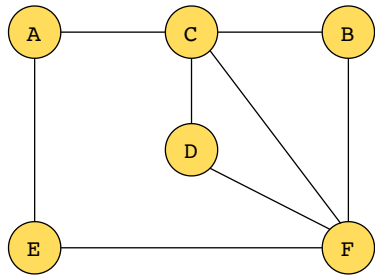
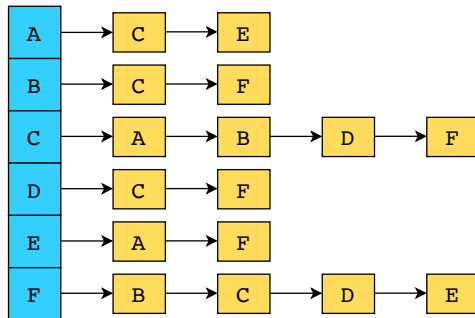
Unweighted directed graph**Represented as an adjacency list****Represented as an adjacency matrix**

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

1 of 4

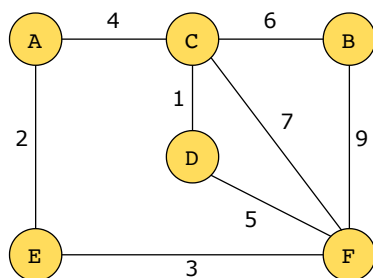
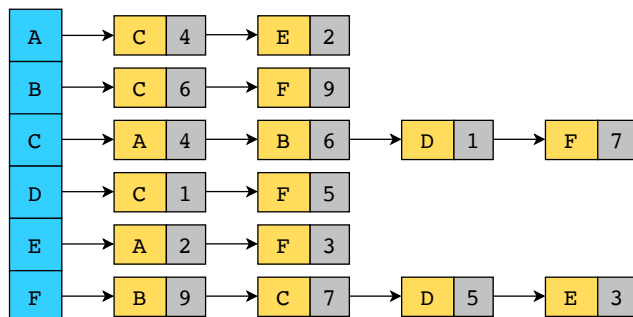
Weighted directed graph**Represented as an adjacency list****Represented as an adjacency matrix**

	A	B	C	D	E	F
A	0	0	4	0	2	0
B	0	0	0	0	0	9
C	0	6	0	1	0	7
D	0	0	0	0	0	5
E	0	0	0	0	0	3
F	0	0	0	0	0	0

Unweighted undirected graph**Represented as an adjacency list****Represented as an adjacency matrix**

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



Weighted undirected graph**Represented as an adjacency list****Represented as an adjacency matrix**

	A	B	C	D	E	F
A	0	0	4	0	2	0
B	0	0	6	0	0	9
C	4	6	0	1	0	7
D	0	0	1	0	0	5
E	2	0	0	0	0	3
F	0	9	7	5	3	0

4 of 4

To solve the coding problems related to graphs, one should have a good understanding of graph traversals and based on these traversals some graph algorithms.

Graph traversal

There are two fundamental graph traversal techniques: breadth-first search (BFS) and depth-first search (DFS). Let's dive deep into these traversals.

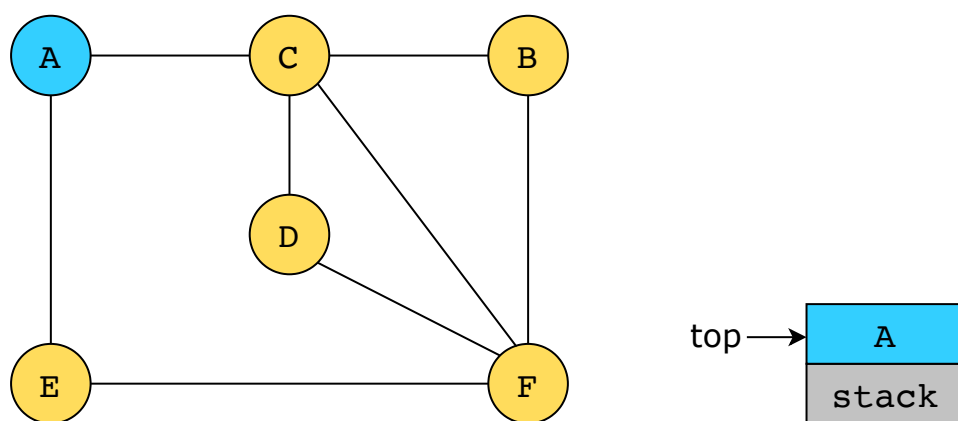
Depth-first search (DFS)

In DFS, the strategy is to explore as far as possible along one path before turning back. The algorithm starts with a chosen source node and proceeds to one of its neighbor nodes while marking the source node as

visited. From there, it goes to one of its neighbors and marks this newly traversed node as visited. This continues until it reaches a node with no neighbors. Now, the algorithm starts backtracking. In backtracking, the algorithm goes one step back and checks for the remaining neighbor nodes that are yet to be explored. This process continues until all the nodes in the graph, that are reachable from the source node, have been visited.

We can implement this algorithm using a stack. Initialize an empty stack and choose a source node, push it onto the stack, and mark it as visited. While the stack is not empty, pop a node, explore its unvisited neighbors, and push them onto the stack, marking them as visited. Continue this process until reaching a node with no unvisited neighbors, then backtrack by popping from the stack. Repeat until the stack is empty, ensuring all connected nodes are visited.

The following illustration shows how a DFS works on a graph as discussed above:



Unvisited To be traversed Visited

Traverse the given graph using DFS. The chosen source node is A, so push it onto the stack.



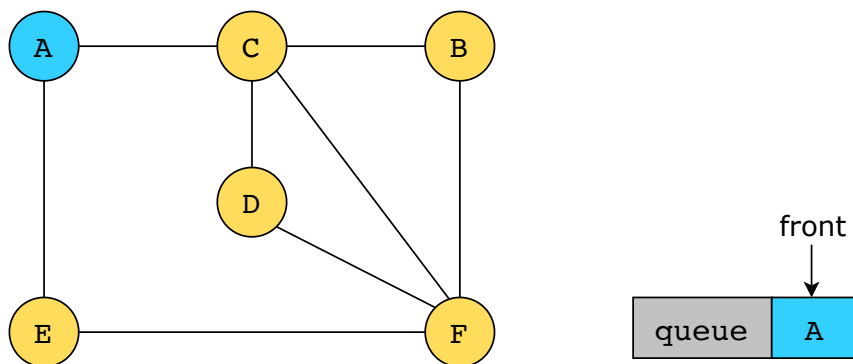
Breadth-first search (BFS)

In BFS, the strategy is to explore the graph in layers, one level at a time. The algorithm begins at a chosen source node and visits all its immediate neighbor nodes while marking them as visited. It then moves on to visit the neighbors of those nodes before proceeding to the next level of neighbors. This process continues until all the nodes in the graph, that are reachable from the source node, have been visited.

We can implement this algorithm using a queue. Initialize an empty queue and choose a source node, enqueue it, and then enter a loop. Within this loop, the algorithm dequeues a node from the front of the queue, visits its immediate neighbors, and marks them as visited. These neighbors are subsequently enqueued into the queue. The queue plays a crucial role in determining the order of exploration, ensuring that nodes at the current level are processed before progressing to the next. This iterative process continues until the queue is empty, signifying that all reachable nodes from the source have been visited.

The following illustration shows how a BFS works on a graph as discussed above:





Unvisited

To be traversed

Visited

Traverse the given graph using BFS. The chosen source node is A, so enqueue it.

1 of 13



Graph algorithms

Traversals can be used to find paths, cycles, or connected components within graphs. However, when aiming for specific objectives, such as determining the shortest path rather than any path, or identifying the minimum spanning tree instead of connected components, traversals can be adapted to address particular optimization problems associated with the graph.

There are various algorithms that help us solve these specific graph problems. Let's go over a few:

- **Dijkstra's algorithm:** It's a variation of DFS and finds the shortest path between two nodes in a weighted graph.



- **Bellman-Ford algorithm:** It's a variation of BFS and finds the shortest paths in a weighted graph, even when negative edge weights are present.
- **Floyd-Warshall algorithm:** It's a variation of BFS and finds the shortest paths between all pairs of nodes in a weighted graph.
- **Topological sorting:** It's similar to DFS and orders nodes in a directed acyclic graph (DAG) to satisfy dependencies.
- **Prim's algorithm:** It finds the minimum spanning tree of a connected, undirected graph.
- **Kruskal's algorithm:** It also finds the minimum spanning tree of a connected, undirected graph.

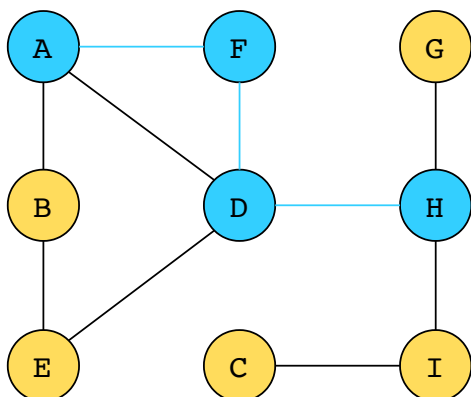
Examples

The following examples illustrate some problems that can be solved with this approach:

1. **Find if a path exists in the graph:** Given the source and destination nodes in a graph, determine whether there is a valid path between them. The graph can be either directed or undirected. Return TRUE if there is a valid path between the source and destination, and FALSE otherwise.



Undirected graph



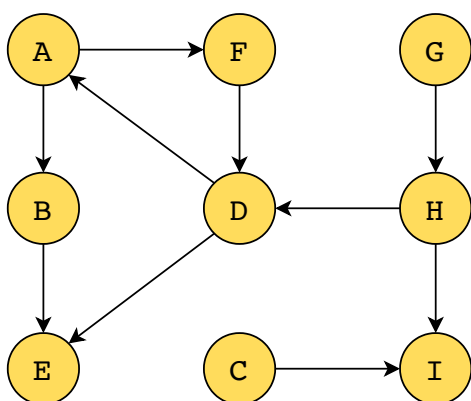
Source	A
Destination	H

Path exists?	TRUE
--------------	------

Perform the breadth-first search (BFS) by adding the source vertex to a queue. Dequeue the vertex from the queue, and add its adjacent vertices into the queue. Repeat the above process until the queue becomes empty. During this process, if the destination vertex is found in the queue, return **TRUE**. Otherwise, if the queue becomes empty and the destination vertex is not found, return **FALSE**.

1 of 2

Directed graph



Source	A
Destination	H

Path exists?	FALSE
--------------	-------

Perform the breadth-first search (BFS) by adding the source vertex to a queue. Dequeue the vertex from the queue, and add its adjacent vertices into the queue. Repeat the above process until the queue becomes empty. During this process, if the destination vertex is found in the queue, return **TRUE**. Otherwise, if the queue becomes empty and the destination vertex is not found, return **FALSE**.

?

T

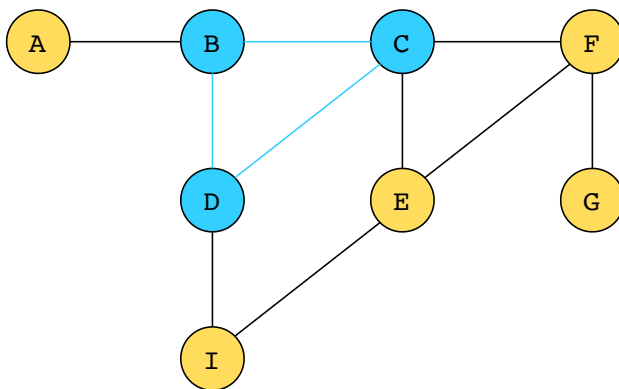
2 of 2





2. **Find if a cycle exists in the graph:** Given a graph, determine whether it contains a cycle or not. The graph can be either directed or undirected. Return TRUE if there is a cycle in the graph, and FALSE otherwise.

Undirected graph



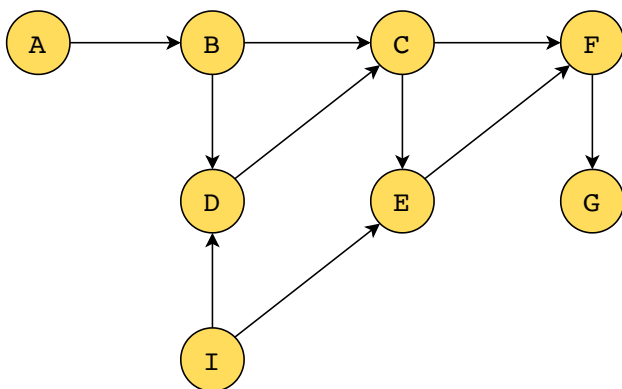
Cycle exists?

TRUE

Start the depth-first search (DFS). Push any starting vertex onto the stack, and mark it as visited. Pop the vertex from the stack and for each of its adjacent vertices, if the adjacent vertex is not visited, push it onto the stack and mark it as visited. If the adjacent vertex is already visited, then a cycle exists, so return TRUE. Repeat the above process for all vertices, and if all the vertices are processed without finding a cycle, return FALSE.

1 of 2



Directed graph

Cycle exists?

FALSE

Start the depth-first search (DFS). Push any starting vertex onto the stack, and mark it as visited. Pop the vertex from the stack and for each of its adjacent vertices, if the adjacent vertex is not visited, push it onto the stack and mark it as visited. If the adjacent vertex is already visited, then a cycle exists, so return TRUE. Repeat the above process for all vertices, and if all the vertices are processed without finding a cycle, return FALSE.

2 of 2

—

[]

Does your problem match this pattern?

Yes, if the following conditions are fulfilled:

- **Relationships between elements:** There is a network of interconnected objects with some relationship between them; that is, the data can be represented as a graph.


Real-world problems

Many problems in the real world use the graphs pattern. Let's look at some examples.

?

Tt

C

- 
- **Routing in computer networks:** The graph representation helps to visualize the computer network, where nodes represent devices such as computers or servers and edges signify connections. Then, graph algorithms such as Dijkstra's can be used to find the shortest and optimal path between any computer and the server. This is to ensure that data travels quickly from our computer to a faraway server on the internet without getting stuck or slowed down.
 - **Flight route optimization:** Airlines use graph-based algorithms to optimize flight routes. The airport network can be represented as a graph, where nodes represent airports, and edges represent flights connecting them. The algorithms can be used to find the shortest route between two airports, reduce fuel consumption, and minimize flight time.
 - **Epidemic spread modeling:** Graph algorithms can be used to predict how a virus might spread in a community. Imagine people as nodes and their interactions as edges. Algorithms simulate how a virus could move through these connections to help plan prevention strategies.
 - **Recommendation systems:** Online streaming services such as Netflix suggest movies we might like based on what we've watched using graph algorithms. Imagine all the movies and viewers as nodes, with edges connecting similar movies or people who like the same things. Graph algorithms then analyze these connections, finding

