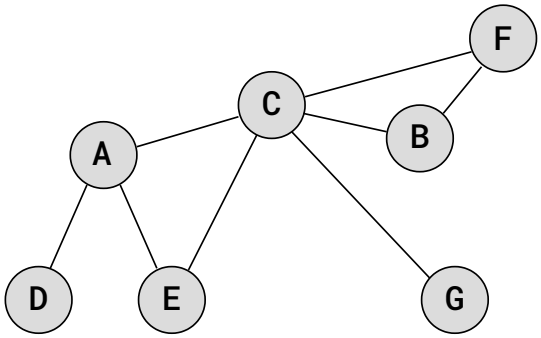
DSA Graphs

Previous Next

Graphs

A Graph is a non-linear data structure that consists of vertices (nodes) and edges.



A vertex, also called a node, is a point or an object in the Graph, and an edge is used to connect two vertices with each other.

Graphs are non-linear because the data structure allows us to have different paths to get from one vertex to another, unlike with linear data structures like Arrays or Linked Lists.

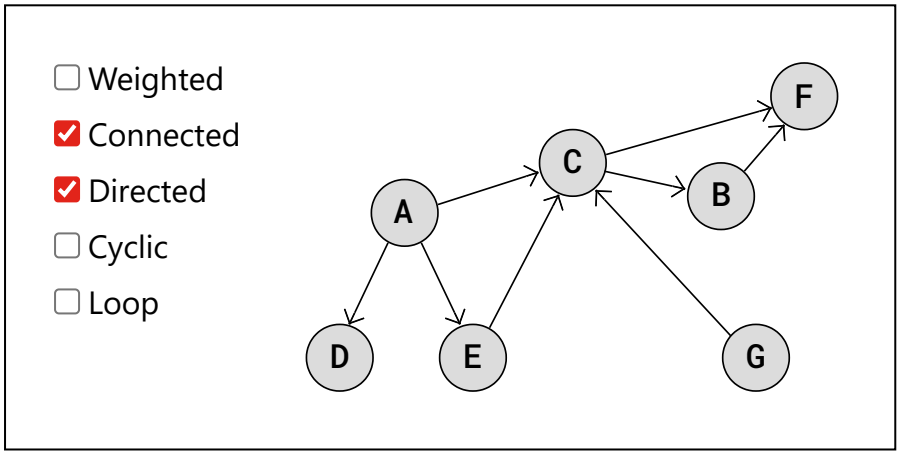
Graphs are used to represent and solve problems where the data consists of objects and relationships between them, such as:

locations when stored as a Graph.

- Internet: Can be represented as a Graph, with web pages as vertices and hyperlinks as edges.
- Biology: Graphs can model systems like neural networks or the spread of diseases.

Graph Properties

Use the animation below to get an understanding of the different Graph properties, and how these properties can be combined.



A **weighted** Graph is a Graph where the edges have values. The weight value of an edge can represent things like distance, capacity, time, or probability.

A **connected** Graph is when all the vertices are connected through edges somehow. A Graph that is not connected, is a Graph with isolated (disjoint) subgraphs, or single isolated vertices.

A **directed** Graph, also known as a digraph, is when the edges between the vertex pairs have a direction. The direction of an edge can represent things like hierarchy or flow.

A cyclic Graph is defined differently depending on whether it is directed or not:

- A **directed cyclic** Graph is when you can follow a path along the directed edges that goes in circles. Removing the directed edge from F to G in the animation above makes the directed Graph not cyclic anymore.
- An **undirected cyclic** Graph is when you can come back to the same vertex you started at without using the same edge more than once. The undirected Graph above is cyclic because we can start and end up in vertes C without using the same edge twice.

Graph Representations

A Graph representation tells us how a Graph is stored in memory.

Different Graph representations can:

- take up more or less space.
- be faster or slower to search or manipulate.
- be better suited depending on what type of Graph we have (weighted, directed, etc.), and what we want to do with the Graph.
- be easier to understand and implement than others.

Below are short introductions of the different Graph representations, but Adjacency Matrix is the representation we will use for Graphs moving forward in this tutorial, as it is easy to understand and implement, and works in all cases relevant for this tutorial.

Graph representations store information about which vertices are adjacent, and how the edges between the vertices are. Graph representations are slightly different if the edges are directed or weighted.

Two vertices are adjacent, or neighbors, if there is an edge between them.

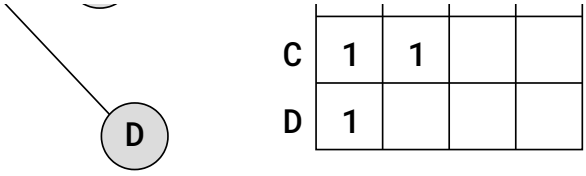
Adjacency Matrix Graph Representation

Adjacency Matrix is the Graph representation (structure) we will use for this tutorial.

How to implement an Adjacency Matrix is shown on the next page.

The Adjacency Matrix is a 2D array (matrix) where each cell on index `(i,j)` stores information about the edge from vertex `i` to vertex `j`.

Below is a Graph with the Adjacency Matrix representation next to it.

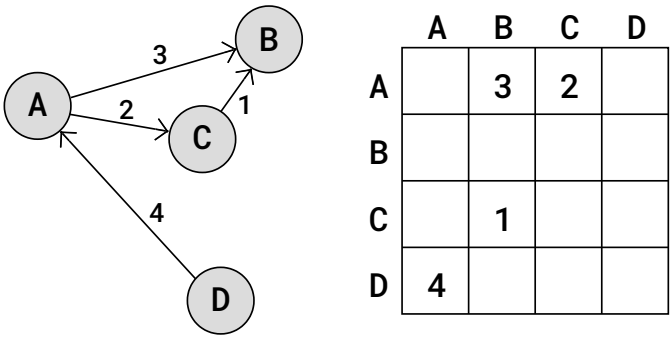


An undirected Graph
and the adjacency matrix

The adjacency matrix above represents an undirected Graph, so the values '1' only tells us where the edges are. Also, the values in the adjacency matrix is symmetrical because the edges go both ways (undirected Graph).

To create a directed Graph with an adjacency matrix, we must decide which vertices the edges go from and to, by inserting the value at the correct indexes `(i,j)`. To represent a weighted Graph we can put other values than '1' inside the adjacency matrix.

Below is a directed and weighted Graph with the Adjacency Matrix representation next to it.



A directed and weighted Graph,
and its adjacency matrix.

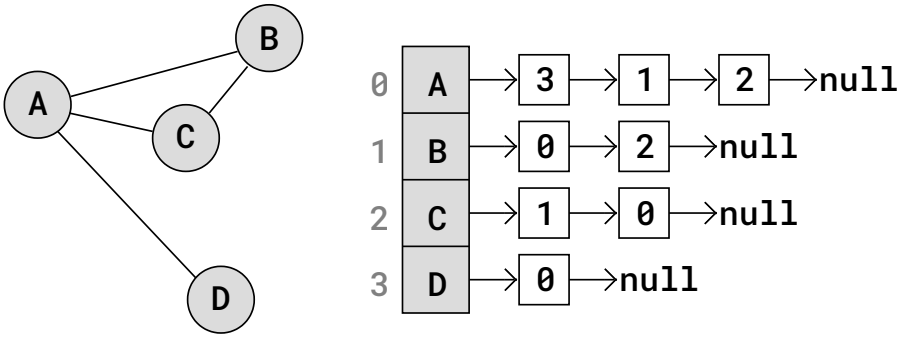
In the adjacency matrix above, the value `3` on index `(0,1)` tells us there is an edge from vertex A to vertex B, and the weight for that edge is `3`.

As you can see, the weights are placed directly into the adjacency matrix for the correct edge, and for a directed Graph, the adjacency matrix does not have to be symmetric.

Adjacency List Graph Representation

In case we have a 'sparse' Graph with many vertices, we can save space by using an Adjacency List compared to using an Adjacency Matrix, because an Adjacency Matrix would reserve a lot of memory on empty Array elements for edges that don't exist.

a Linked List (or Array) with the vertex's edges.



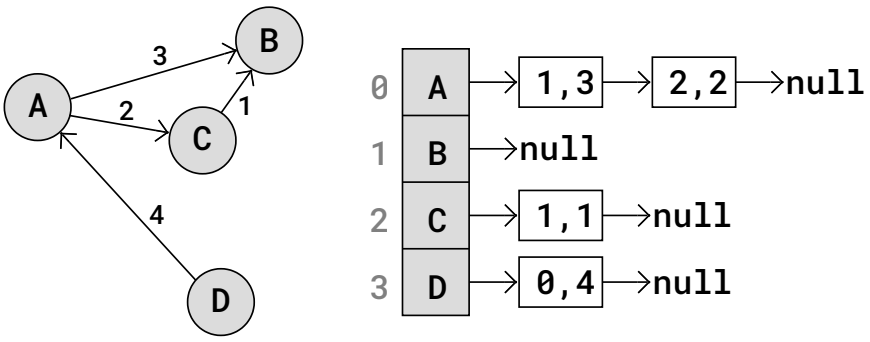
An undirected Graph and its adjacency list.

In the adjacency list above, the vertices A to D are placed in an Array, and each vertex in the array has its index written right next to it.

Each vertex in the Array has a pointer to a Linked List that represents that vertex's edges. More specifically, the Linked List contains the indexes to the adjacent (neighbor) vertices.

So for example, vertex A has a link to a Linked List with values 3, 1, and 2. These values are the indexes to A's adjacent vertices D, B, and C.

An Adjacency List can also represent a directed and weighted Graph, like this:



A directed and weighted Graph and its adjacency list.

In the Adjacency List above, vertices are stored in an Array. Each vertex has a pointer to a Linked List with edges stored as `i,w`, where `i` is the index of the vertex the edge goes to, and `w` is the weight of that edge.

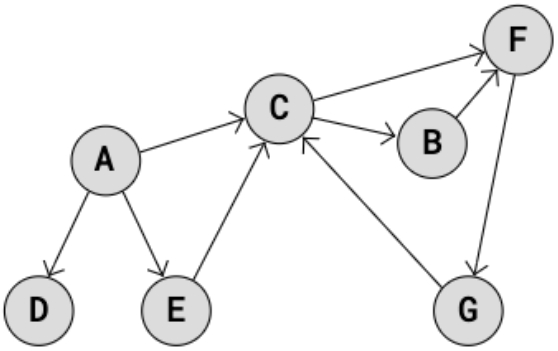
Node D for example, has a pointer to a Linked List with an edge to vertex A. The values `0,4` means that vertex D has an edge to vertex on index `0` (vertex A), and the weight of that

DSA Exercises

Test Yourself With Exercises

Exercise:

How can the Graph below be described?



The Graph is cyclic, connected, and .

Submit Answer »

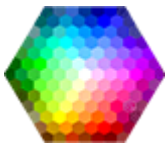
Start the Exercise

‹ Previous

Next ›

Sign in to track progress

COLOR PICKER



Top Tutorials

HTML Tutorial
CSS Tutorial
JavaScript Tutorial
How To Tutorial
SQL Tutorial
Python Tutorial
W3.CSS Tutorial
Bootstrap Tutorial
PHP Tutorial
Java Tutorial
C++ Tutorial
jQuery Tutorial

Top References

HTML Reference
CSS Reference
JavaScript Reference
SQL Reference
Python Reference
W3.CSS Reference
Bootstrap Reference
PHP Reference
HTML Colors
Java Reference
AngularJS Reference
jQuery Reference

Top Examples

HTML Examples
CSS Examples
JavaScript Examples
How To Examples
SQL Examples
Python Examples
W3.CSS Examples
Bootstrap Examples
PHP Examples
Java Examples
XML Examples
jQuery Examples

Get Certified

HTML Certificate
CSS Certificate
JavaScript Certificate
Front End Certificate
SQL Certificate
Python Certificate
PHP Certificate
jQuery Certificate
Java Certificate
C++ Certificate
C# Certificate
XML Certificate

W3Schools is optimized for learning and training. Examples might be simplified to improve reading and learning.
Tutorials, references, and examples are constantly reviewed to avoid errors, but we cannot warrant full correctness
of all content. While using W3Schools, you agree to have read and accepted our terms of use, cookies and privacy policy.

Tutorials References Exercises Sign In

SS JAVASCRIPT SQL PYTHON JAVA PHP HOW TO W3.CSS C C