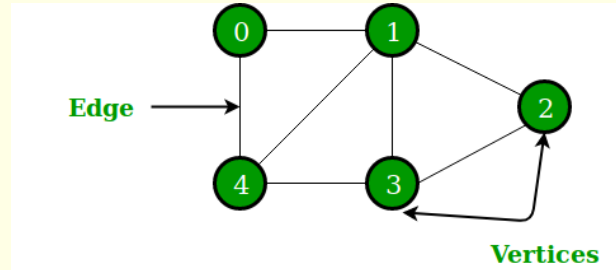


Graph Terminology

GRAPH



Modeling connections or relationships between items

Vertex /node

Edge is the line

Each vertex has a '**degree**' – number of connections

NO UNIQUE ROOT

Type of graphs:
Undirected graph
Directed graph
Weighted graph

Breadth-First and Depth-First Traversal

The Applications of Graphs

Graphs have many uses in the world of Computer Science, for example:

- Social Networks
- Transport Networks
- Operating Systems

Key words

Keyword	Definition
Directed Graph	A directed graph is a set of objects that are connected together, where the edges are directed from one vertex to another.
Undirected Graph	An undirected graph is a set of objects that are connected together, where the edges do not have a direction.
Weighted Graph	A weighted graph is a set of objects that are connected together, where a weight is assigned to each edge.
Adjacency Matrix	Also known as the connection matrix is a matrix containing rows and columns which is used to present a simple labelled graph.
Adjacency List	This is a collection of unordered lists used to represent a finite graph.
Vertices/Nodes	A vertex (or node) of a graph is one of the objects that are connected.
Edges/Arcs	An edge (or arc) is one of the connections between the nodes (or vertices) of the network.
Dictionary	A collection of names, definitions, and attributes about data elements that are being used or captured in a database, information system, or part of a research project.



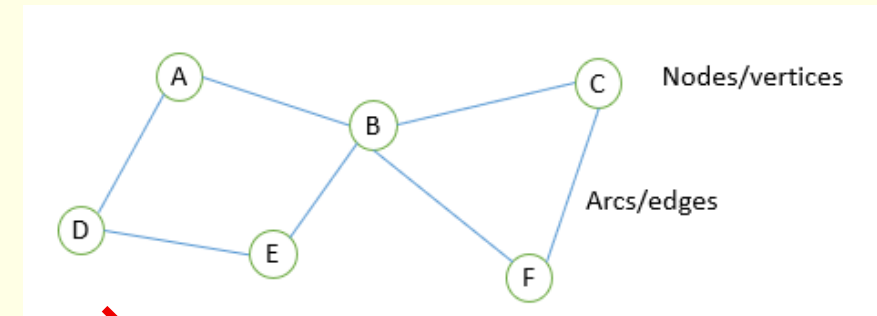
Implementing a graph

Two possible implementations of a graph are the **adjacency matrix** and the **adjacency list**.

Adjacency Matrix

- A two-dimensional array can be used to store information about a directed or undirected graph.
- Each of the rows and columns represents a node, and a value stored in the cell at the intersection of row i , column j indicates that there is an edge connecting node i and node j .
- Graphs with no weights are given a value of 1 for connected nodes
- In the case of an undirected graph, the adjacency matrix will be symmetric, with the same entry in $(0,1)$ as in $(1,0)$, for example.

No Weighting, Undirected graph



	A	B	C	D	E	F
A	-	1	-	1	-	-
B	1	-	1	-	1	1
C	-	1	-	-	-	1
D	1	-	-	-	1	-
E	-	1	-	1	-	-
F	-	1	1	-	-	-

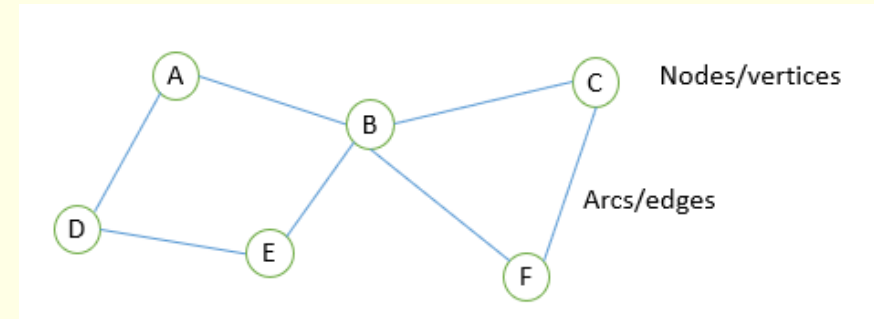
Implementing a graph

Two possible implementations of a graph are the **adjacency matrix** and the **adjacency list**.

A list of all the nodes is created, and each node points to a list of all the adjacent nodes to which it is directly linked.

The adjacency list can be implemented as a list of dictionaries, with the key in each dictionary being the node and the value, the edge weight.

No Weighting, Undirected graph



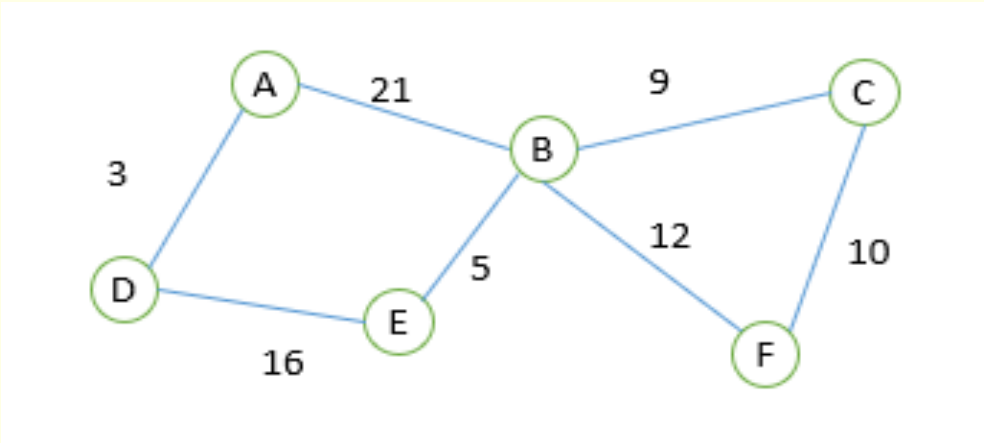
A	[B, D]
B	[A, C, E, F]
C	[B, F]
D	[A, E]
E	[B, D]
F	[B, C]

The next nodes should be in alphabetical order

Weighted graph

Weighted graphs add a value to an arc.

This might represent the distance between places or the time taken between train stations



Adjacency List

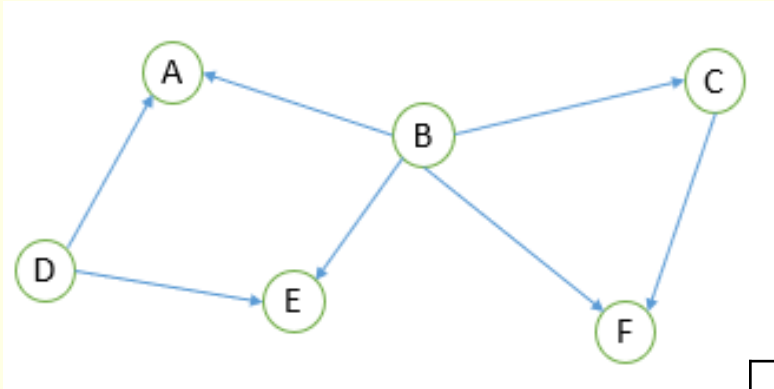
A	{B:21, D:3}
B	{A:21, C:9, E:5, F:12}
C	{B:9, F:10}
D	{A:3, E:16}
E	{B:5, D:16, }
F	{B:12, C:10}

Adjacency Matrix

	A	B	C	D	E	F
A	-	21	-	3	-	-
B	21	-	9	-	5	12
C	-	9	-	-	-	10
D	3	-	-	-	16	-
E	-	5	-	16	-	-
F	-	12	10	-	-	-

Directed Graph

- Directed graphs only apply in one direction and are represented with edges with arrow heads on one end.



Adjacency List

A	
B	[A, C, E, F]
C	[F]
D	[A, E]
E	
F	

Adjacency Matrix

		To					
From		A	B	C	D	E	F
	A						
	B	1		1		1	1
	C						1
	D	1				1	
	E						
	F						

Advantages and Drawbacks of Adjacency lists vs Adjacency matrix

Quick exam comparison sentence

Adjacency lists are more memory-efficient for sparse graphs, while adjacency matrices allow faster edge lookup but use more memory.

Adjacency List

Advantages

- Uses **less memory** for graphs with few edges (sparse graphs)
- Faster to **list neighbours** of a vertex
- Good for **large graphs** with many vertices

Drawbacks

- Slower to check if **a specific edge exists**
- Slightly more complex to understand and implement
- Not ideal for **dense graphs**

Adjacency Matrix

Advantages

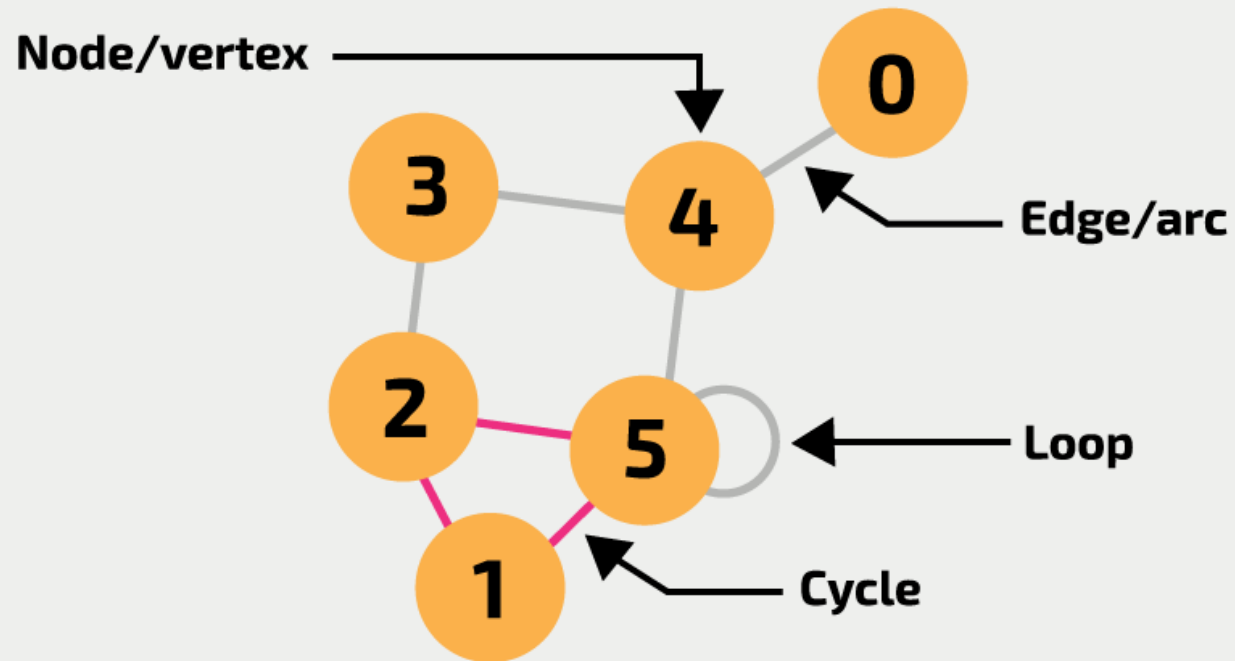
- Very quick to **check if an edge exists** (constant time)
- Good for **dense graphs** (many edges)
- Easy to represent **weighted graphs**

Drawbacks

- Uses **a lot of memory**, even if few edges exist
- Adding or removing vertices is inefficient
- Wastes space for **sparse graphs**

Graphs allow cycles and loops

- A **cycle** is a closed path, i.e. a path that starts and ends at the same node (and no node is visited more than once)
- A **loop** is an edge that connects a node to itself



Loop Example.

Web Page Navigation

A loop represents a page linking back to itself

Example: Clicking “refresh”

Traffic Systems

A loop can represent **waiting at the same junction**

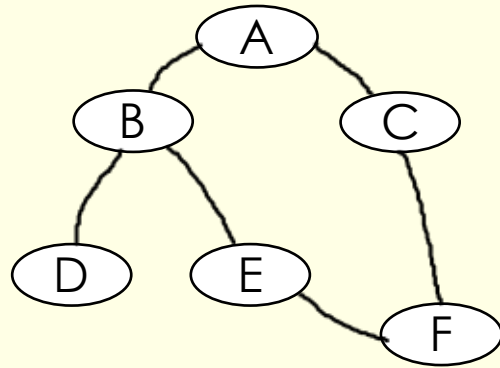
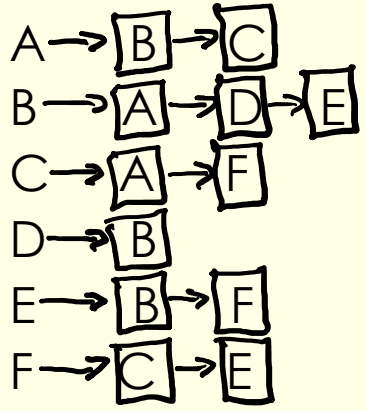
Example: Traffic lights that remain red for multiple cycles

Process Scheduling (Operating Systems)

A loop shows a process **waiting or idle**

Graph representation

Adjacency List



Adjacency Matrix

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

Adjacency list can be coded as a dictionary

```
graph = {  
  'A': ['B', 'C'],  
  'B': ['A', 'D', 'E'],  
  'C': ['A', 'F'],  
  'D': ['B'],  
  'E': ['B', 'F'],  
  'F': ['C', 'E']}
```

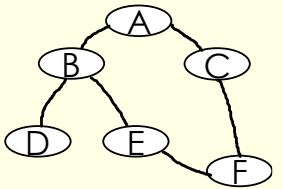
Adjacency Matrix can be coded as a 2D list

```
graph =  
[[0,1,1,0,0,0],  
 [1,0,0,1,1,0],  
 [1,0,0,0,0,1],  
 [0,1,0,0,0,0],  
 [0,1,0,0,0,1],  
 [0,0,1,0,1,0]]
```

Depth-first search

Uses a stack to manage traversal
Visits nodes by exploring as far down one branch as possible before backtracking Typically explores neighbours in a depth-wise manner

Current node	Stack	Visited
A	C B	A
B	C E D	A B
D	C E	A B D
E	CF	A B D E
F	C	A B D E F
C		A B D E F C



A B D E F C

- This method makes use of a stack data structure to push each visited node onto the stack and then pop them from the stack when there are no nodes left to visit
1. Set the **current node** to the **start node**.
 2. If the **current node** has **unvisited neighbours**, **push** them to the stack **in reverse order**.
 3. If the **current node** is **not in visited**, add it to visited.
 4. **Pop** the next node from the stack and set it as the new **current node**.
 5. Repeat until the **stack is empty**.

Procedure depth_first_search (graph, start_node)

```
visited = []
s = Stack()
current_vertex = start_node
while current_vertex != None #while there is a node in the stack

    for vertex in reversed(graph[current_vertex])
        if not vertex in visited then
            s.push(vertex) #adds the vertex to the stack if not already in visited
        end if
    next vertex

    if not current_vertex in visited then
        visited.append(current_vertex) # current_vertex is added to visited
    endif

    current_vertex = s.pop() #next current_vertex
endwhile

print(visited)

end procedure
```

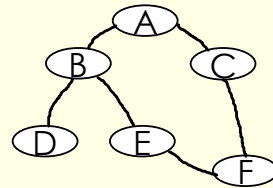
Trinket Python Implementation:
<https://trinket.io/python3/90a4a3516460>

```
graph = {
'A': ['B', 'C'],
'B': ['A', 'D', 'E'],
'C': ['A', 'F'],
'D': ['B'],
'E': ['B', 'F'],
'F': ['C', 'E']
}
```

Breadth-First search

Uses a queue to manage traversal
Visits nodes level-by-level, exploring all neighbours before moving to the next level. Ideal for finding the shortest path in an unweighted graph

Current node	Queue	Visited
	A	
A	BC	A
B	C D E	A B
C	D E F	A B C
D	E F	A B C D
E	F	A B C D E
F		A B C D E F



A B C D E F

1. Set **current node** to **start node**
2. If the **current node** has **unvisited neighbours**, enqueue **unvisited neighbour** nodes to the **queue**
3. If the **current node** is **not in visited**, add it to visited.
4. **Dequeue** node from queue and set to **current node**.
5. Repeat until the **queue is empty**

Algorithm

Procedure bfs(graph,start_node)

visited = []

q = queue() #create queue

current_vertex = start_node# currentNode

while current_vertex != None #while the queue is not empty

for vertex in graph[current_vertex] #get all nodes that are adjacent nodes
if not vertex in visited then # if vertex is in visited do not add to queue
q.enqueue(vertex) #add vertex to queue
next vertex

if not current_vertex in visited then #if the current node has not been visited
visited.append(current_vertex) #add node to visited

current_vertex = q.dequeue() #next current_vetex

print(visited)

End procedure

Trinket Python Implementation:

<https://trinket.io/python3/6bf76ad43d10>

```
graph = {  
'A': ['B', 'C'],  
'B': ['A', 'D', 'E'],  
'C': ['A', 'F'],  
'D': ['B'],  
'E': ['B', 'F'],  
'F': ['C', 'E']  
}
```

```
procedure dfs(graph, current_vertex)
```

```
    visited.append(current_vertex)
```

```
    for vertex in graph[current_vertex]
```

```
        if not vertex in visited then
            dfs(graph, vertex)
```

```
        end if
```

```
    next vertex
```

```
end procedure
```

#Main program starts here

```
graph = {'A': ['B', 'C'],
        'B': ['A', 'D', 'E'],
        'C': ['A', 'F'],
        'D': ['B'],
        'E': ['B', 'F'],
        'F': ['C', 'E']}
```

```
visited = []
```

```
dfs(graph, "A")
```

```
print visited
```

No stack used.

Backtracking

- The algorithm used to demonstrate a DFS does not show step-wise backtracking.
- It is an algorithm that shows you the order in which nodes will be visited in the graph.
- If you wanted to show step-wise backtracking the algorithm would need to be extended.
- An example of a Graph DFS and backtracking is available for you to look at on Trinket.
- This approach used to implement the graph – it is using a graph class.

Explain the differences between depth-first search and breadth-first search in terms of data structures and traversal order.

Feature	DFS	BFS
Data Structure	Stack (or recursion)	Queue
Traversal Order	Goes deep before backtracking	Visits all neighbors level by level
Memory Usage	Can use less memory in wide graphs	Uses more memory for wide graphs
Use Case	Solving mazes, topological sort	Shortest path in unweighted graph