CHAPTER 5

Graphs and Elementary Graph Algorithms

V.1. Introduction

In the real world, many problems are represented in terms of objects and the relationships among them. For example, in an airline route map, we might be interested in questions like: "**What's the fastest way to go from Alger to M'sila?**" or "**What is the cheapest way to go from Alger to M'sila?**".

To answer these questions, we need information about connections (airline routes) between objects (towns). Graph technologies and analytics provide powerful tools for connected data that are used to solve these types of problems.

In this chapter, we consider how graphs can be represented, and take a look at algorithms for some basic operations such as searching and traversal.

V.2. Graph Definition

Graph is a Non-Linear data structure comprising a finite set of nodes and edges.

Generally, a $\boldsymbol{G}$ graph is a pair $\boldsymbol{G = (V, E)}$, where $\boldsymbol{V = \{v_1, v_2, \cdots, v_n\}}$ is a finite set of nodes, called **vertices**, and $\boldsymbol{E = \{e_1, e_2, \cdots, e_m\}}$ is a finite set of collections of vertex pairs, called **edges**.

- We will often denote $\boldsymbol{n = |V|}$, and $\boldsymbol{e = |E|}$.
- Vertices and edges are positions and storage elements.

Example:

Graph $\boldsymbol{G}$ can be defined as $\boldsymbol{G = (V, E)}$ where :

$\boldsymbol{V = \{A, B, C, D, E\}}$, and $\boldsymbol{E = \{\{A, B\}, \{A, C\}, \{A, D\}, \{B, D\}, \{C, D\}, \{B, E\}, \{E, D\}\}}$.

A graph is generally displayed as figure 5.1, in which the vertices are represented by circles and the edges by lines.

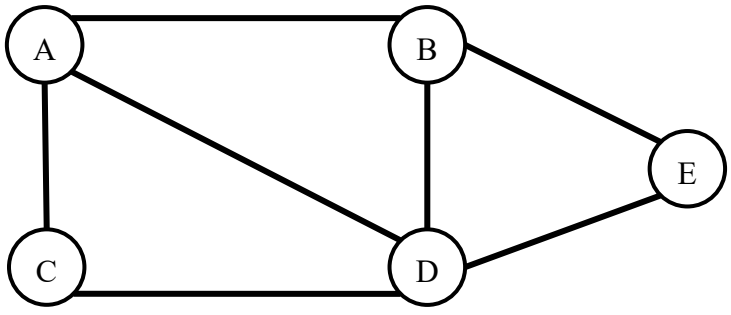


Figure 5.1 : A Graph.

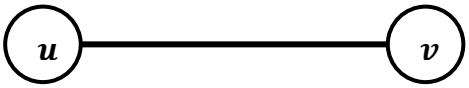
V.3. Graph Terminology

Vertex : Individual data element of a graph is called as Vertex. **Vertex** is also known as **node**. In above example graph, A, B, C, D , and E are known as vertices.

Edge : An edge is a connecting link between two vertices. Edges are three types :

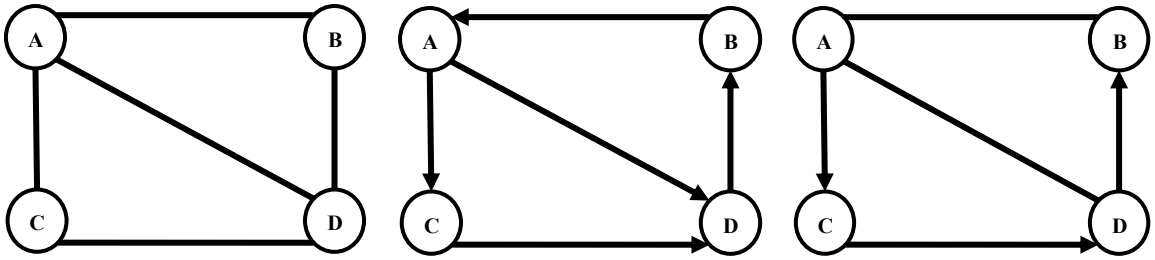
- **Directed Edge :** A directed edge is a unidirectional edge. If there is directed edge between vertices u and v then edge (u, v) is not equal to edge (v, u).
  - ordered pair of vertices $(u, v) \neq (v, u)$
  - first vertex u is the origin
  - second vertex v is the destination
  - **Example:** one-way road traffic

- **Undirected Edge :** An undirected edge is a bidirectional edge. If there is undirected edge between vertices u and v then edge $\{u, v\}$ is equal to edges (u, v) and (v, u)
  - unordered pair of vertices $\{u, v\}$
  - **Example:** railway lines



- **Weighted Edge :** A weighted edge is an edge with value (cost) on it.

- **Undirected Graph :** A graph with only undirected edges is said to be undirected graph.
- **Directed Graph** : A graph with only directed edges is said to be directed graph.
- **Mixed Graph** : A graph with both undirected and directed edges is said to be mixed graph



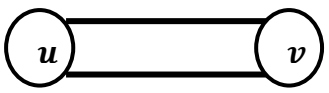
a) Undirected Graph **b) Directed Graph** **c) Mixed Graph**

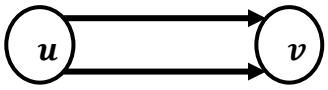
- **End vertices or Endpoints :** The two vertices joined by edge are called end vertices (or endpoints) of that edge.
- **Origin :** If an edge is directed, its first endpoint is said to be the origin of it.
- **Destination :** If an edge is directed, its first endpoint is said to be the origin of it and the other endpoint is said to be the destination of that edge.

Origin **Destination**

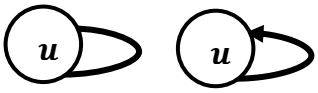
- **Adjacent Nodes :** If there is an edge between vertices u and v then both u and v are said to be adjacent. In other words, vertices u and v are said to be adjacent if there is an edge between them.
- **Incident :** Edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.
- **Outgoing Edge** : A directed edge is said to be outgoing edge on its origin vertex.

- **Incoming Edge :** A directed edge is said to be incoming edge on its destination vertex.
- **Degree** : Total number of edges connected to a vertex is said to be degree of that vertex.
- **Indegree :** Total number of incoming edges connected to a vertex is said to be indegree of that vertex.
- **Outdegree :** Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.
- **Parallel edges or Multiple edges :** If there are two undirected edges with same end vertices and two directed edges with same origin and destination, such edges are called parallel edges or multiple edges.





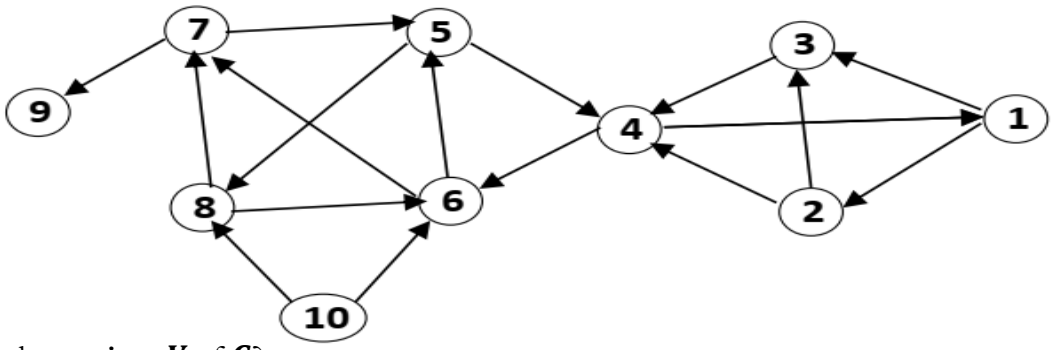
- **Self-loop :** Edge (undirected or directed) is a self-loop if its two endpoints coincide with each other.



- **Simple Graph :** A graph is said to be simple if there are no parallel and self-loop edges.
- **Path** : A path is a sequence of alternate vertices and edges that starts at a vertex and ends at another vertex such that each edge is incident to its predecessor and successor vertex. Simple path is a path with no repeated vertices.
- **Cycle** : A Cycle in a graph is a **path that starts and ends at the same vertex**, with no repetitions of vertices (except the starting and ending vertex, which are the same).
- **A graph with no cycles** is called a **tree**. A tree is an acyclic connected graph.

Example :

Consider the following Graph $\boldsymbol{G = (V, E)}$.



What are the **vertices** $\boldsymbol{V}$ of $\boldsymbol{G}$?

- $\boldsymbol{V = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}}$

What are the **edges** $\boldsymbol{E}$ of $\boldsymbol{G}$?

- $\boldsymbol{E = \{(1,2), (1,3), (2,3), (2,4), (3,4), (4,1), (4,6), (5,4), (5,8), (6,5), (6,7), (7,5), (7,9), (8,6), (8,7), (10,6), (10,8)\}}$

What is the **order** and **size** of the graph $\boldsymbol{G}$?

- Order of Graph G : n=|V|=10
- Size of Graph G : e=|E|=17

What is the type of graph $\boldsymbol{G}$?

- Directed Graph

Find the **number of outgoing** and **incoming edges** of vertex **6**.

- The number of **outgoing** edges from vertex 6 is 2
- The number of **incoming** edges from vertex 6 is 3

Find the **degree , indegree , and outdegree** of vertex **4**.

- The degree of vertex 4 is 5
- The indegree of vertex 4 is 3
- The outdegree of vertex 4 is 2

Find a path between vertices **4** and **6**, Is it simple?

- C1 = (4, 6) is a simple path
- C2 = (4, 6, 7, 5, 8, 6) is not a simple path

Does the graph have a cycle?

- Yes, the graph have a cycle ➔ C3 = (6, 7, 5, 8, 6)

V.4. Graph Representations

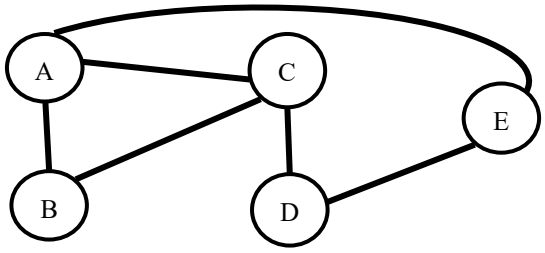
The graphs are **non-linear, non-regular** structures. Two important graph abstract data types (ADTs) that can be used to represent a graph in memory are the **adjacency matrix** and **adjacency list**. The benefits of choosing one implementation over the other depend on the type of graph problem you are working on.

V.4.1. Adjacency Matrix

An adjacency matrix is a matrix M where the cell $M[i][j]$ represents whether an edge exists from vertex i to vertex j.

- In an unweighted graph, the element at row i, column j of the adjacency matrix is **1** if and only if the edge (v_i, v_j) exists in the set of edges E, and **0** otherwise.
- For **undirected graphs**, an adjacency matrix is always symmetric across the diagonal (since a connection between i and j also implies a connection between j and i).

An example of an adjacency matrix for an undirected graph is shown below :



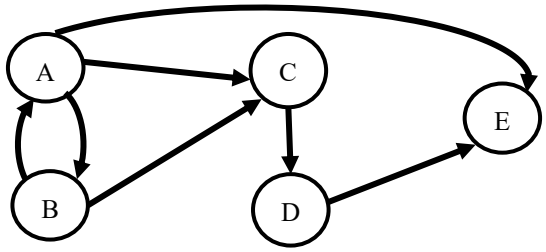
	A	B	C	D	E
A	0	1	1	0	1
B	1	0	1	0	0
C	1	1	0	1	0
D	0	0	1	0	1
E	1	0	0	1	0

Because the adjacency matrix of an undirected graph is always symmetric, we can save time and space by only filling out half of the matrix. If we do this, we must make sure to access the rows and columns in the correct order (such as making sure that the row index is never larger than the column index).

	A	B	C	D	E
A	0	1	1	0	1
B	-	0	1	0	0
C	-	-	0	1	0
D	-	-	-	0	1
E	-	-	-	-	0

However, in a **directed graph**, we will have to fill out the entire matrix. This is because the existence of a path from vertex $\boldsymbol{A}$ to $\boldsymbol{B}$ does not mean that there is one the other way around.

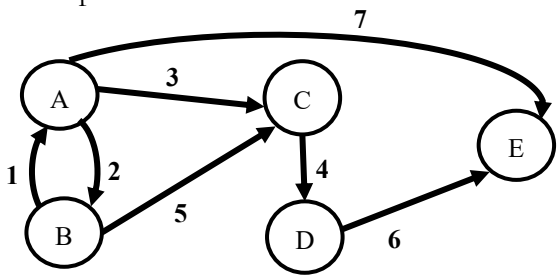
An example of an adjacency matrix for a directed graph is shown below:



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

An adjacency matrix can also be used to represent weighted graphs. If a graph is weighted, the cell $\boldsymbol{M[i][j]}$ would store the weight of the edge from vertex $\boldsymbol{i}$ to vertex $\boldsymbol{j}$. If no edge exists between vertex $\boldsymbol{i}$ and vertex $\boldsymbol{j}$, then the cell $\boldsymbol{M[i][j]}$ is assigned an edge weight of infinity.

An example is shown below:



	A	B	C	D	E
A	∞	2	3	∞	7
B	1	∞	5	∞	∞
C	∞	∞	∞	4	∞
D	∞	∞	∞	∞	6
E	∞	∞	∞	∞	∞

Remark :

To initialize a value to ∞, you can use **`std::numeric_limits<double>::infinity()`** from the **`<limits>`** library.

Adjacency Matrix Declarations

```
int N, M;                    // N vertices and M Edges
int arr_edges[][2];          // Given Edges
int Adj[N][N];               // For Adjacency Matrix
```

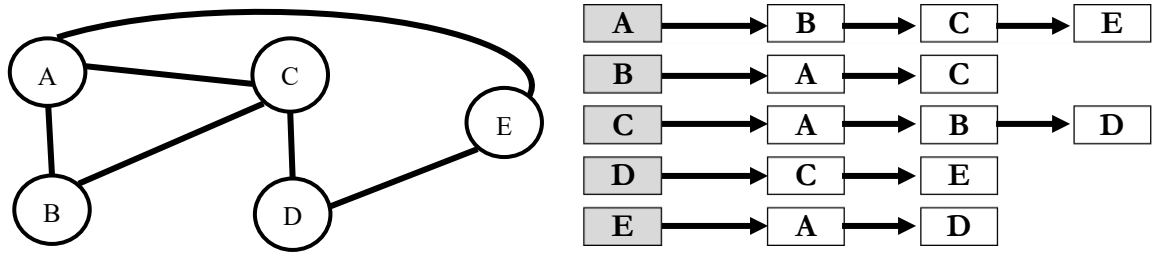
Complexity

Given a graph with $|\boldsymbol{V}|$ vertices, the time complexity to build an adjacency matrix is $\boldsymbol{\mathcal{O}(|V^2|)}$, since you have to construct a $|\boldsymbol{V}| \times |\boldsymbol{V}|$ matrix. The adjacency matrix representation requires a lot of space: for a graph with $\boldsymbol{V}$ vertices we must allocate space in $\boldsymbol{\mathcal{O}(V^2)}$. The time required to iterate over **all adjacent** vertices of a vertex in an adjacency matrix is $\boldsymbol{\mathcal{O}(|V|)}$, and the time required to iterate over **all edges** is $\boldsymbol{\mathcal{O}(|V^2|)}$**.** However, the benefit of the adjacency matrix representation is that **adding** an edge and **checking** for the existence of an edge are both $\boldsymbol{\mathcal{O}(1)}$ operations.

V.4.2. Adjacency List

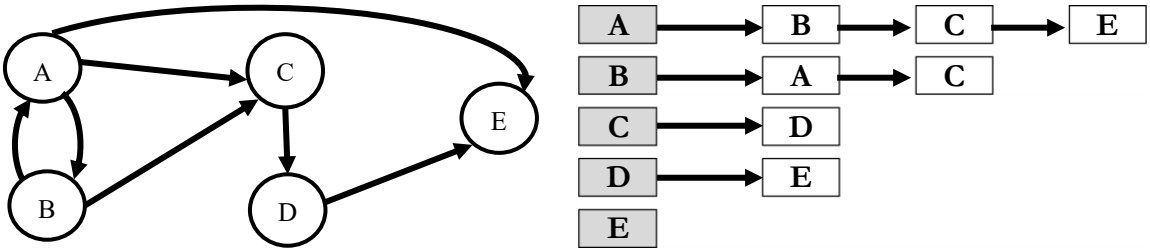
Instead, if a graph is sparse, a better alternative is to use an **adjacency list** as the underlying representation. In an adjacency list, each vertex $\boldsymbol{v}$ is mapped to a list of edges that originate from $\boldsymbol{v}$.

For example, the following depicts an adjacency list for an undirected graph, where the presence of $\boldsymbol{A \rightarrow [B, C, E]}$ indicates that vertex A has a connection to vertices $\boldsymbol{B, C}$, and $\boldsymbol{E}$.

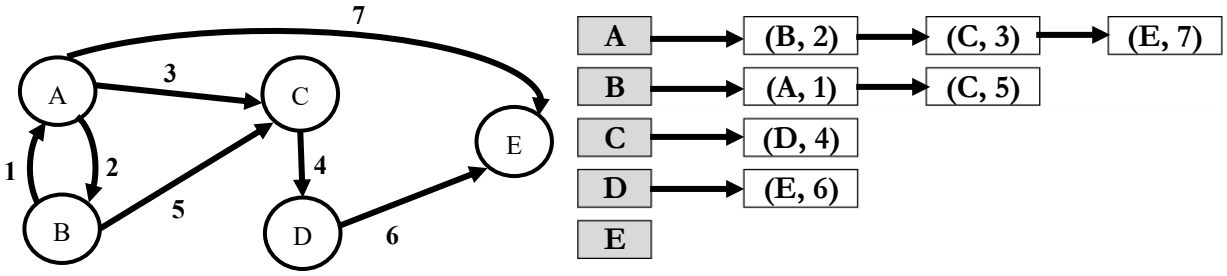


Adjacency lists also work with **directed graphs** and **weighted graphs**.

The following depicts an adjacency list for a directed graph (which is similar to the undirected representation, with the exception that $\boldsymbol{X} \rightarrow \boldsymbol{Y}$ does not necessarily guarantee $\boldsymbol{Y} \rightarrow \boldsymbol{X}$).



The following depicts an adjacency list for a weighted graph. When dealing with weighted graphs, each edge in the adjacency list is paired with its weight (e.g., $\boldsymbol{A} \rightarrow (\boldsymbol{B}, \mathbf{2})$ indicates that there is an edge from $\boldsymbol{A}$ to $\boldsymbol{B}$ with weight $\mathbf{2}$).



Adjacency List Declarations

```
struct GNode{
    int vertex;
    GNode* next;
};
struct Graph {
    unsigned int numVertices;
    GNode** adjLists;
    bool isDirected;
};
```

Complexity

Given a graph with $|\boldsymbol{V}|$ vertices, the time complexity to build an adjacency list is $\boldsymbol{\mathcal{O}}(|\boldsymbol{E}|)$, and the space complexity is $\boldsymbol{\mathcal{O}}(|\boldsymbol{V}| + |\mathbf{E}|)$. The time complexity is $\boldsymbol{\mathcal{O}}(|\boldsymbol{E}|)$ because you have to iterate through each edge and insert it into the adjacency list, where each insertion takes $\boldsymbol{\mathcal{O}}(\mathbf{1})$ time (assuming the adjacency list is stored in a **2-D** vector or a hash table). The space complexity is $\boldsymbol{\mathcal{O}}(|\boldsymbol{V}| + |\mathbf{E}|)$ because storing all the vertices requires $\boldsymbol{\mathcal{O}}(|\boldsymbol{V}|)$ space, and storing all the edges in the lists requires $\boldsymbol{\mathcal{O}}(|\mathbf{E}|)$ space.

V.5. Graph Traversal

Perhaps the most fundamental problem with graphs is to systematically visit every edge and vertex of a graph. Indeed, all basic graph accounting operations (such as printing or copying graphs, and converting between alternative representations) are applications of graph traversal.

To solve these problems on graphs, we need a mechanism for traversing the graphs. Graph traversal algorithms are also called graph search algorithms. Like trees traversal algorithms (In-order, Pre-order, Post-order, and Level-Order traversals), graph search algorithms can be thought of as starting at some source vertex in a graph and "searching" the graph by going through the edges and marking the vertices. Now, we will discuss two such algorithms for traversing the graphs.

- Depth First Search (DFS)
- Breadth First Search (BFS)

V.5.1. Depth First Search (DFS)

The depth-first search (DFS) algorithm works similarly to preordered tree traversal, but this algorithm also uses the stack. The DFS algorithm is a core graph traversal algorithm that can be used to systematically explore the vertices and edges of a graph that are reachable from a given source vertex.

In a DFS, the algorithm starts at a source vertex and explores each branch of the graph as far as possible until there are no more undiscovered vertices along its path, at this point, it backtracks and continues along another branch.

Implementing a DFS generally relies on recursion or the **`stack`**<> data structure. An outline of the DFS algorithm is as follows:

1) Mark the source vertex as visited, and then push its neighbors (i.e., adjacent vertices) into a stack. This can be done by pushing all neighbors into an actual **`stack<>`**, or by performing a recursive call on every neighbor (which pushes neighbors onto the program call stack).
2) Pop the vertex at the top of the **`stack`** and set it as the current vertex (this behavior is automatically done by a recursive call if a recursive approach is used).
3) Push the unvisited neighbors of the current vertex into the **`stack`** (or perform a recursive call on each neighbor, if a recursive approach is used) and mark them as visited.
4) Repeat **steps 2** and **3** until the **`stack`** is empty.

Example : (Algorithm Depth-First Search (DFS))

A depth-first search can be done both iteratively (using a **`stack<>`**) or recursively (using the program stack).

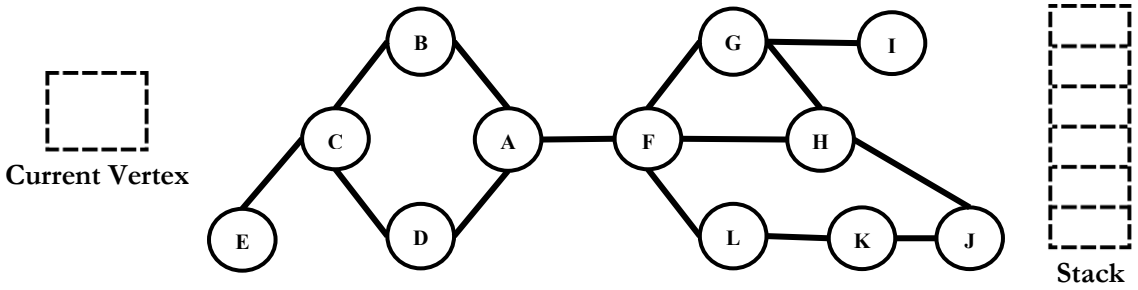
For example, consider the following graph, which we will traverse using an **iterative** depth-first search starting at vertex **A**.

Here, we will classify visited nodes into two groups: **discovered** and **processed**.

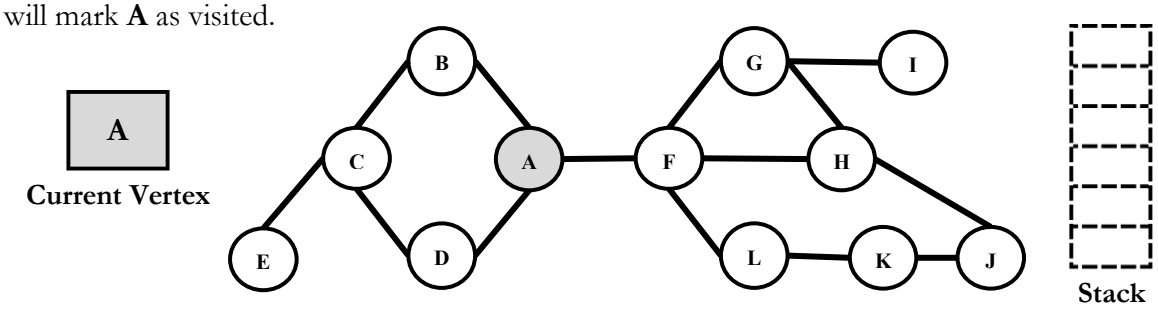
- A **discovered** vertex is a vertex that has been encountered and pushed into the **`stack`**; discovered vertices will be represented with a **light gray** shading.

- A **processed** vertex is a vertex that has already been taken out of the **`stack`** and had its adjacency list fully examined; processed vertices will be represented with a **dark gray** shading.

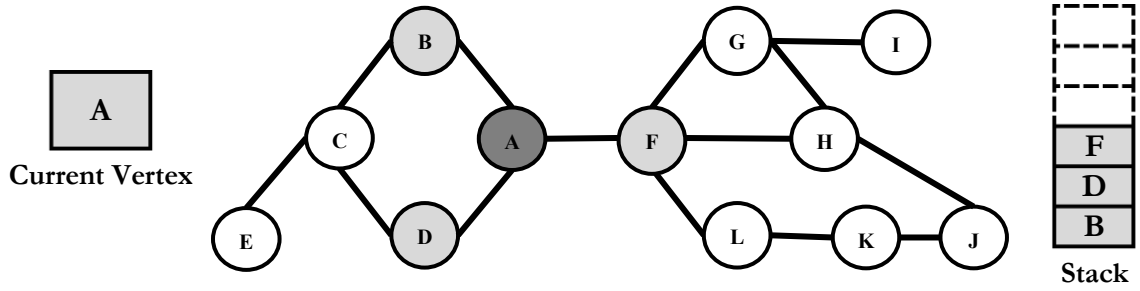
We start by initializing a **`stack<>`** to track vertices yet to be explored.



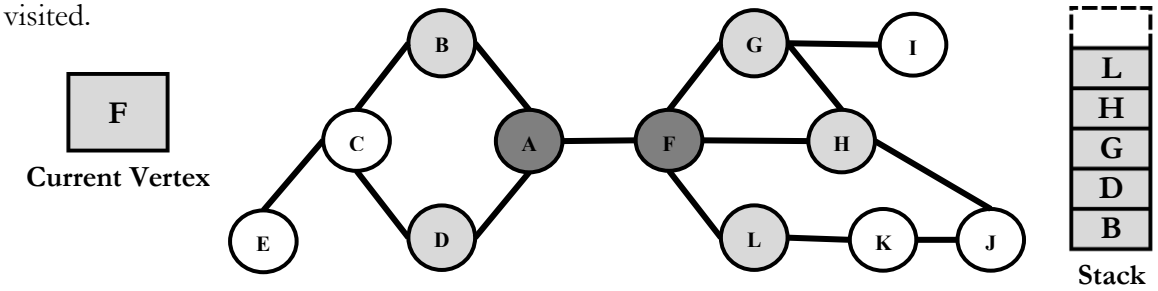
First, we will mark the starting vertex as visited. In this case, our starting vertex is vertex **A**, so we will mark **A** as visited.



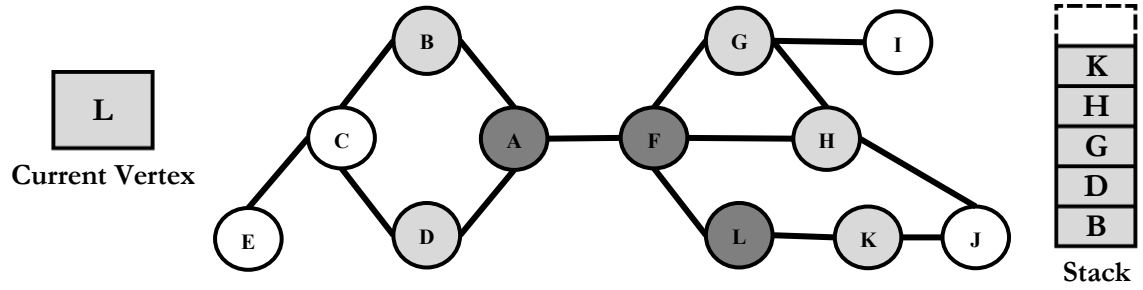
Next, we look at the vertices that vertex **A** is connected to, mark them as visited, and add them to the **`stack`**. The order we add these vertices doesn't matter, but for our example, we will add them in alphabetical order (i.e., **B** is inserted first, then **D**, then **F**).



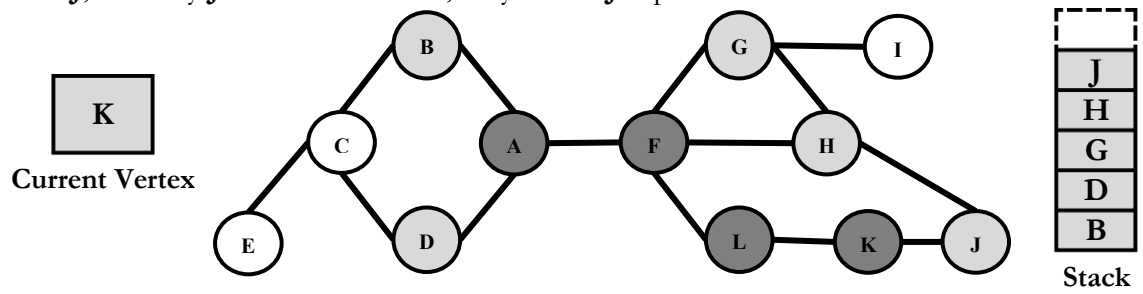
Next, we take out the vertex at the top of the stack and set it as our current vertex. Here, vertex **F** is at the top of the **`stack`**, so we pop it and set it as our current vertex. We then push all of **F** 's unvisited neighbors into the **`stack`**. Vertex F is connected to vertices **A, G, H**, and **L**, but only **G, H**, and **L** are unvisited. Thus, vertices **G, H,** and **L** are pushed into the **`stack`** and marked as visited.



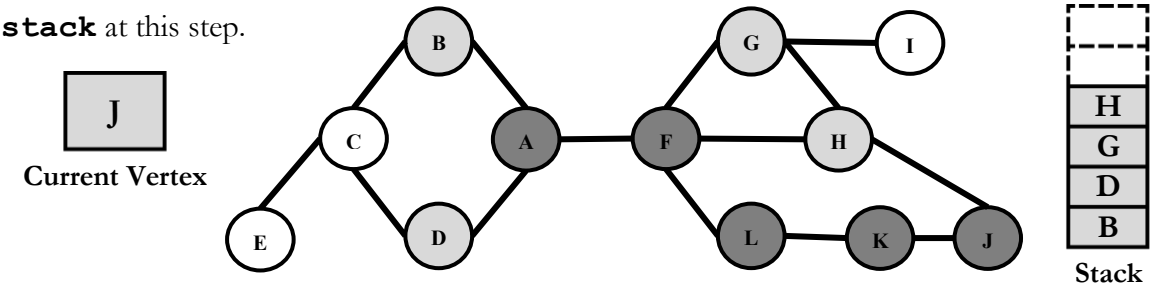
Vertex **L** is now at the top of the **`stack`**, so we pop it and set it as our current vertex. We then push all of vertex **L**'s unvisited neighbors into the **`stack`**. Vertex **L** is connected to vertices **F** and **K**, but only **K** is unvisited. Thus, only vertex **K** is pushed into the **`stack`** and marked as visited.



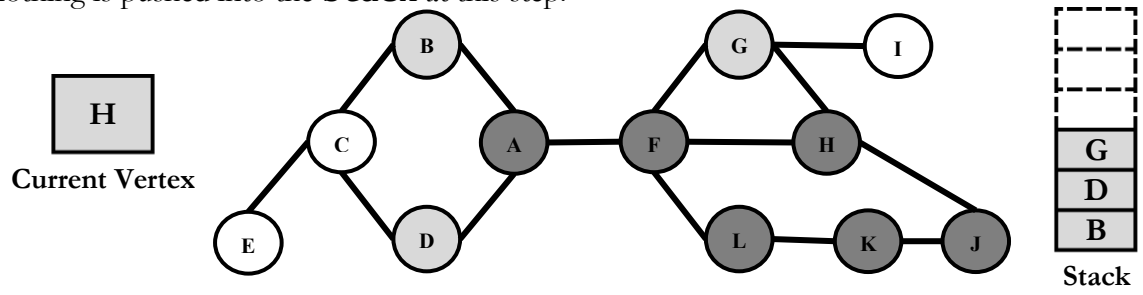
Vertex **K** is now at the top of the **`stack`**, so we pop it off and set it as our current vertex. We then push all of vertex **K**'s unvisited neighbors into the **`stack`**. Vertex **K** is connected to vertices **L** and **J**, but only **J** is unvisited. Thus, only vertex **J** is pushed into the stack and marked as visited.



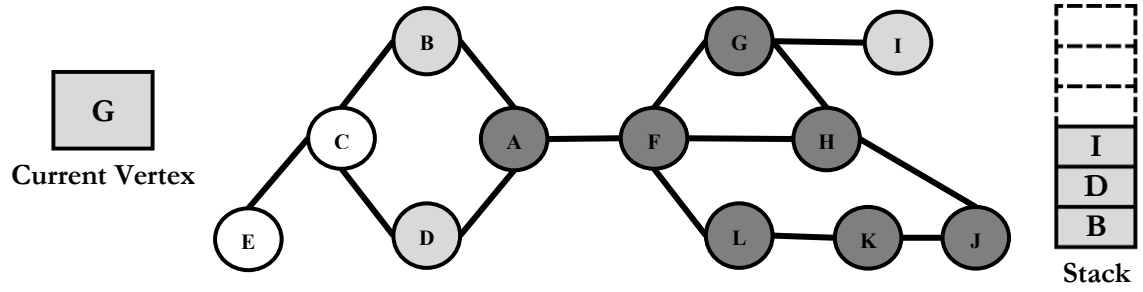
Vertex **J** is now at the top of the **`stack`**, so we pop it off and set it as our current vertex. We then push all of vertex **J**'s unvisited neighbors into the **`stack`**. Vertex **J** is connected to vertices **H** and **K** however, both **H** and **K** have already been marked as visited. Thus, nothing is pushed into the **`stack`** at this step.



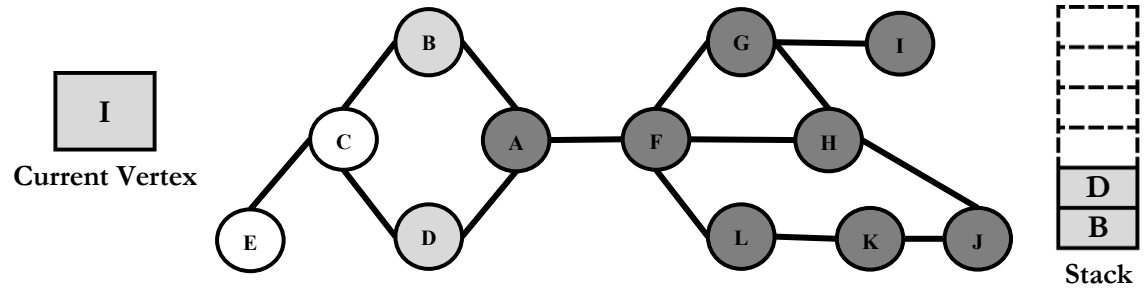
Vertex **H** is now at the top of the **`stack`**, so we pop it off and set it as our current vertex. We then push all of vertex **H**'s unvisited neighbors into the **`stack`**. Vertex H is connected to vertices **F, G**, and **J**, but similar to before, all three of these vertices have been marked as visited. Thus, nothing is pushed into the **`stack`** at this step.



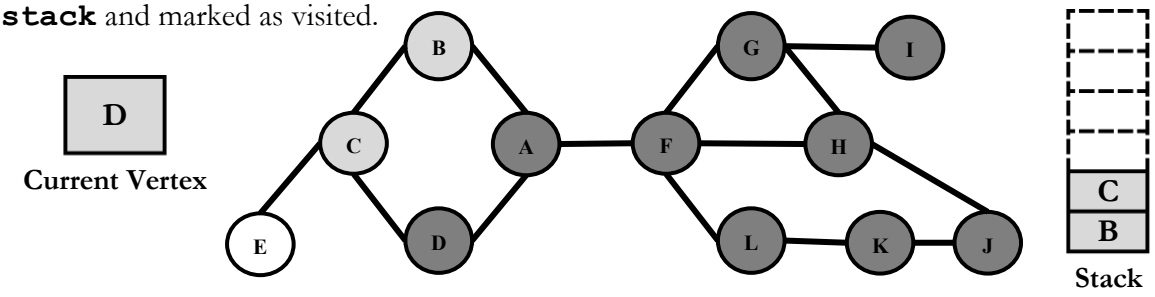
Vertex **G** is at the top of the **`stack`**, so we pop it and set it as our current vertex. We then push all of vertex **G**'s unvisited neighbors into the **`stack`**. Vertex **G** is connected to vertices **F , H**, and **I**, but only vertex **I** is unvisited. Thus, vertex **I** is pushed into the **`stack`** and marked as visited.



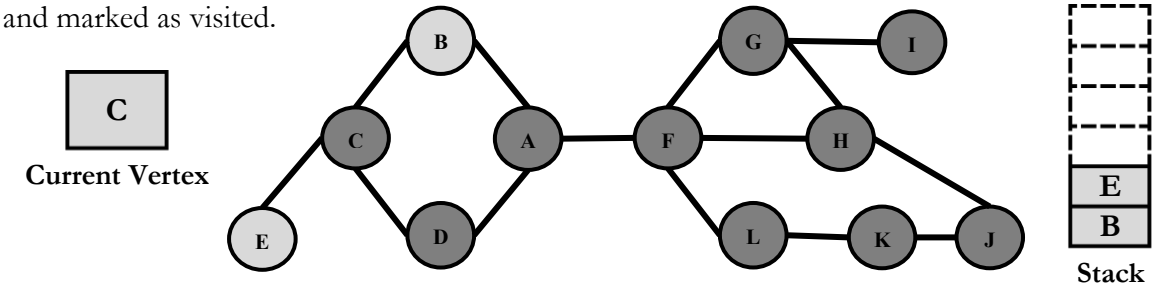
Vertex **I** is now at the top of the **stack**, so we pop it off and set it as our current vertex. We then push all of vertex **I**'s unvisited neighbors into the **stack**. Vertex **I** is connected to vertex **G**, but vertex **G** has already been marked as visited. Thus, nothing is pushed into the **stack** at this step.



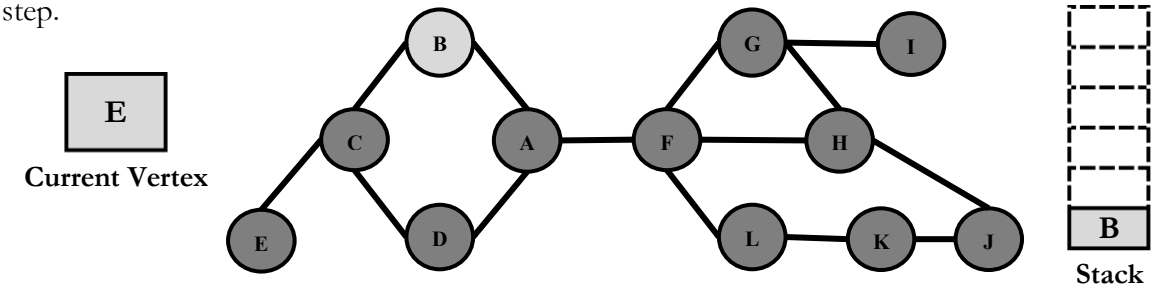
Vertex **D** is now at the top of the **stack**, so we pop it off and set it as our current vertex. We then push all of vertex **D**'s unvisited neighbors into the **stack**. Vertex **D** is connected to vertices **A** and **C**, but vertex **A** has already been marked as visited. Thus, only vertex **C** is pushed into the **stack** and marked as visited.



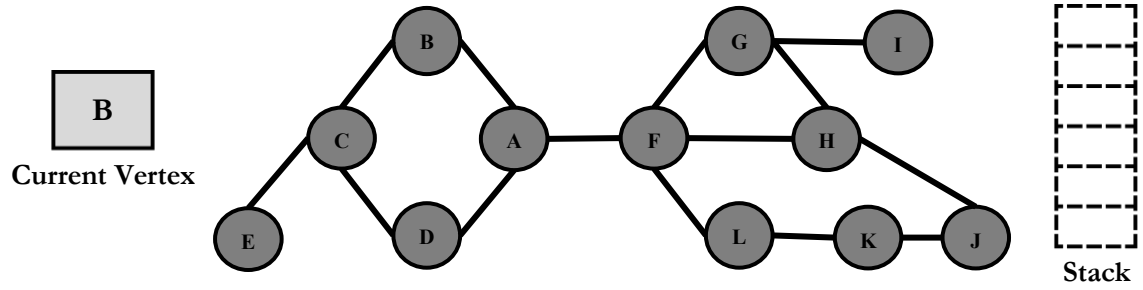
Vertex **C** is now at the top of the **stack**, so we pop it off and set it as our current vertex. We then push all of vertex **C**'s unvisited neighbors into the **stack**. Vertex **C** is connected to vertices **B, D**, and **E**, but only vertex **E** has not yet been visited. Thus, vertex **E** is pushed into the **stack** and marked as visited.



Vertex **E** is now at the top of the **stack**, so we pop it and set it as our current vertex. We then push all of vertex **E**'s unvisited neighbors into the **stack**. Vertex **E** is connected to vertex **C**, but vertex **C** has already been marked as visited. Thus, nothing gets pushed into the **stack** at this step.



Vertex **B** is now at the top of the **stack**, so we pop it off and set it as our current vertex. We then push all of vertex **B**'s unvisited neighbors into the **stack**. Vertex **B** is connected to vertices **A** and **C**, but both of these vertices have already been marked as visited. Thus, nothing gets pushed into the **stack** at this step.

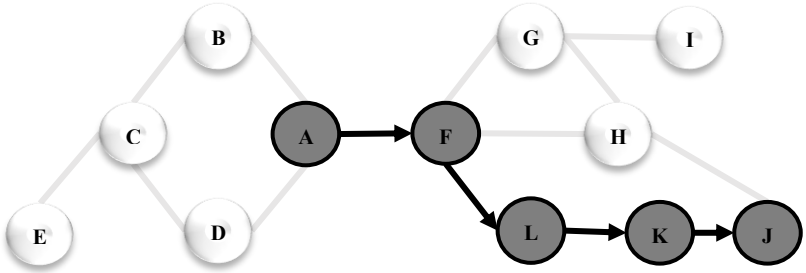


The **stack** is now empty, so our **depth-first search (DFS)** is complete, and we have processed every vertex reachable from vertex **A**. **The DSF traversal order of the vertices is as follows:**

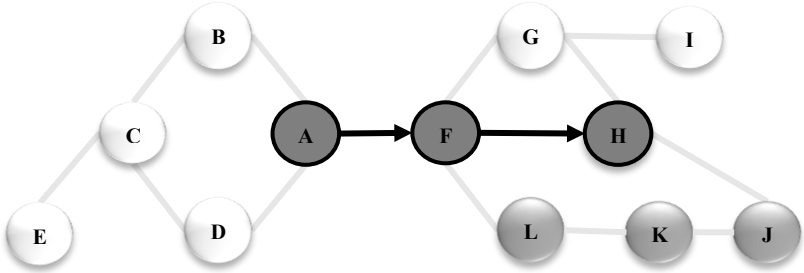
A →F → L → K → J → H → G → I → D → C → E → B

Notice that, because a depth-first search relies on a **stack** to process the nodes in a graph, the algorithm explores and processes vertices in the graph one branch at a time in **LIFO** order.

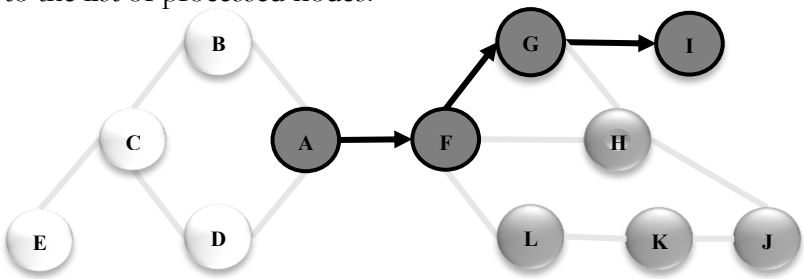
Here, the depth-first search first processed the branch $\boldsymbol{A \rightarrow F \rightarrow L \rightarrow K \rightarrow J}$:



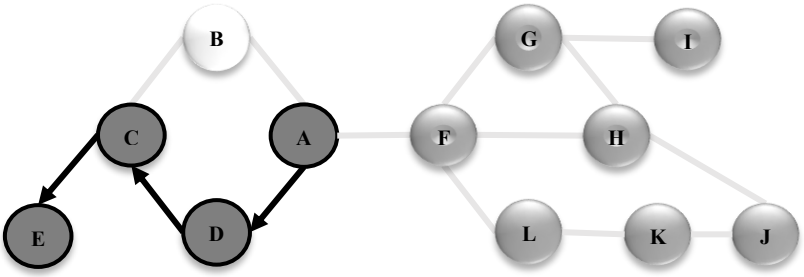
After processing these five vertices, the search then backtracks to vertex **F** and explores the new branch $\boldsymbol{A \rightarrow F \rightarrow H}$, which adds vertex **H** to the list of processed nodes.



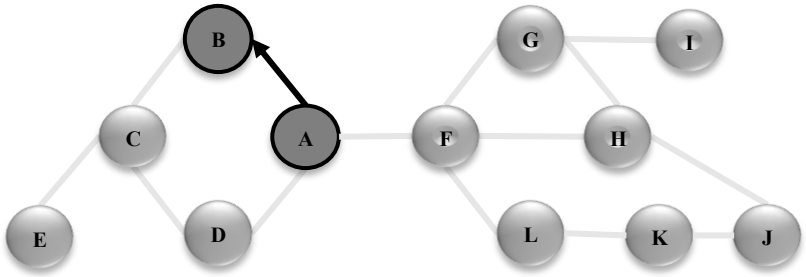
The search then backtracks to vertex **F** and explores the branch $\boldsymbol{A \rightarrow F \rightarrow G \rightarrow I}$, which adds vertices **G** and **I** to the list of processed nodes.



The search then backtracks to vertex **A** and explores the branch $\boldsymbol{A \rightarrow D \rightarrow C \rightarrow E}$, adding vertices **D, C**, and **E** to the list of processed nodes.



The search then backtracks to vertex **A** and explores the final branch ***A → B***, which adds vertex **B** to the list of processed nodes.



V.5.2. Breadth First Search (BFS)

The breadth-first search (BFS) algorithm works similar to level order traversal of the trees, but this algorithm also uses queues. In fact, level order traversal got inspired from BFS. BFS works level by level. The breadth-first search (BFS) algorithm is another graph traversal algorithm that can be used to explore the vertices and edges of a graph that are reachable from a given source vertex.

In a breadth-first search, the algorithm starts at a source vertex and explores vertices of the graph in a breadth-ward fashion, in order of increasing edge distance from the source node.

The implementation of a breadth-first search relies on the **queue<>** data structure. An outline of the algorithm is as follows:

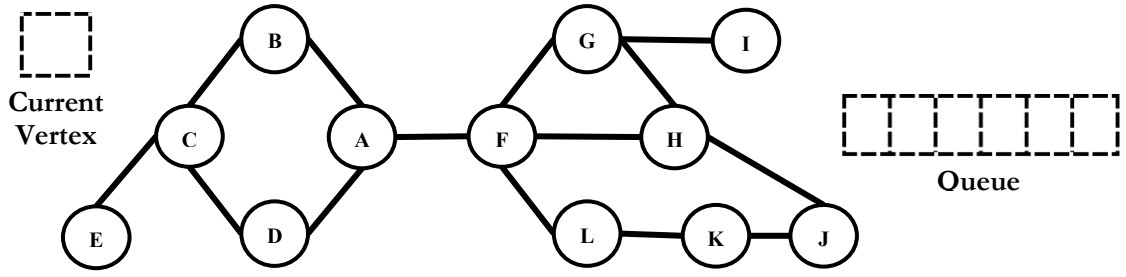
1) Mark the source vertex as visited, and then push its neighbors (i.e., adjacent vertices) into a **queue**.
2) Pop the vertex at the front of the **queue** and set it as the current vertex.
3) Push the unvisited neighbors of the current vertex into the **queue** and mark them as visited.
4) Repeat **steps 2** and **3** until the **queue** is empty.

Example : (Algorithm Breadth First Search (BFS))

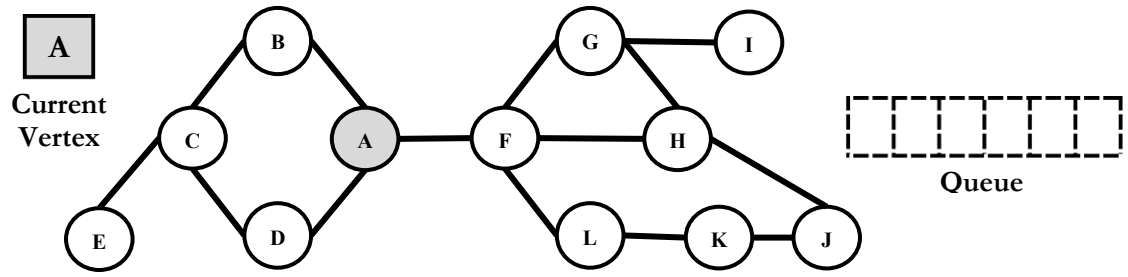
As an example, consider the following graph, which we will traverse using a breadth-first search starting at vertex **A**.

- Similar to the depth-first search example, **discovered** vertices will be given a **light gray** shading,
- while **processed** vertices will be given a **dark gray** shading.

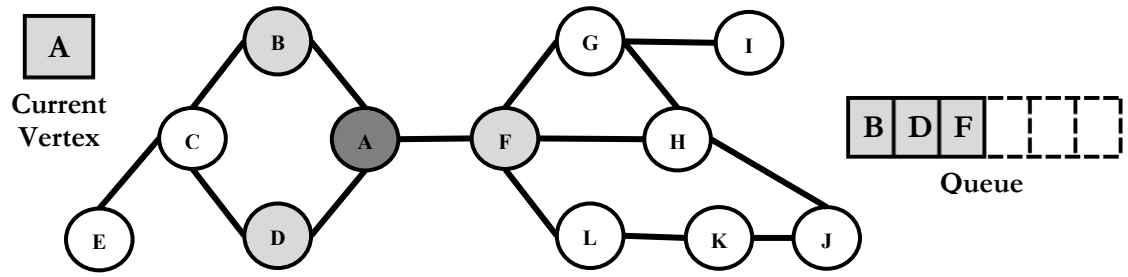
We start the breadth-first search process by creating a **queue<>** that will be used to keep track of the vertices that we still need to explore.



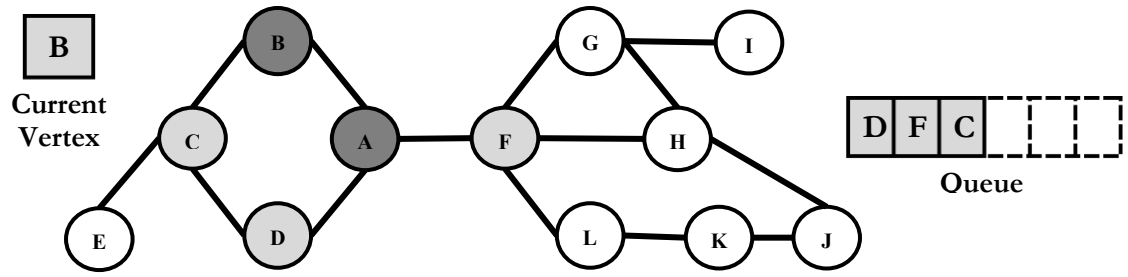
The first step is to mark the starting vertex as visited. In this case, our starting vertex is vertex **A**, so we will mark **A** as visited.



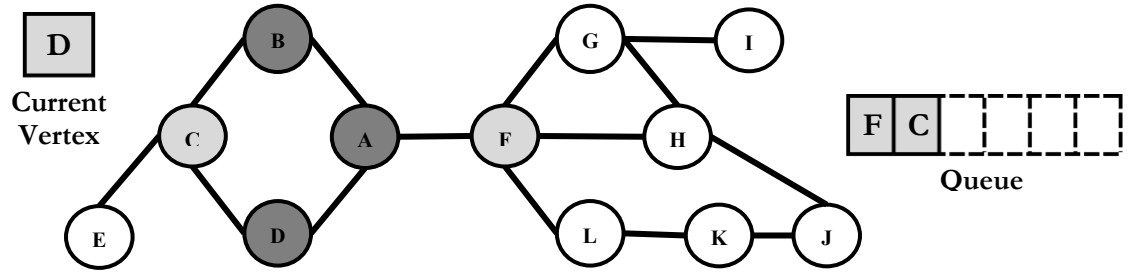
Next, we look at the vertices that vertex **A** is connected to, mark them as visited, and add them to the **queue**. The order we add these vertices does not matter, but for our example, we will add them in alphabetical order (i.e., **B** is inserted first, then **D**, then **F**).



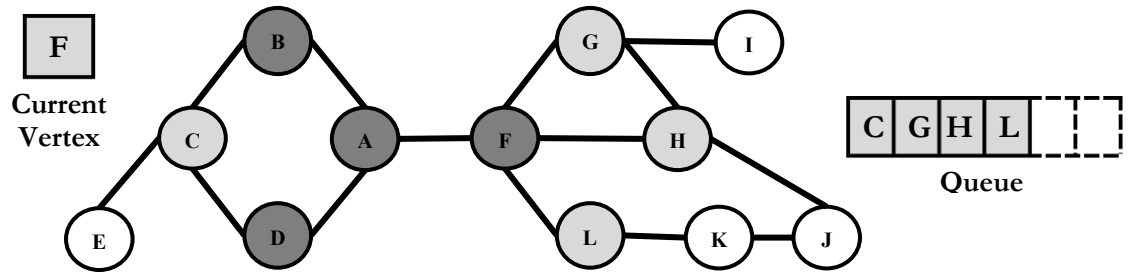
We then take out the vertex at the front of the **queue**, vertex **B**, and set it as our current vertex. Since vertex **B** is our current vertex, we push all of its unvisited neighbors into the **queue**. Vertex **B** is connected to vertices **A** and **C**, but only vertex **C** is marked as unvisited, so vertex **C** gets pushed into the **queue** and marked as visited.



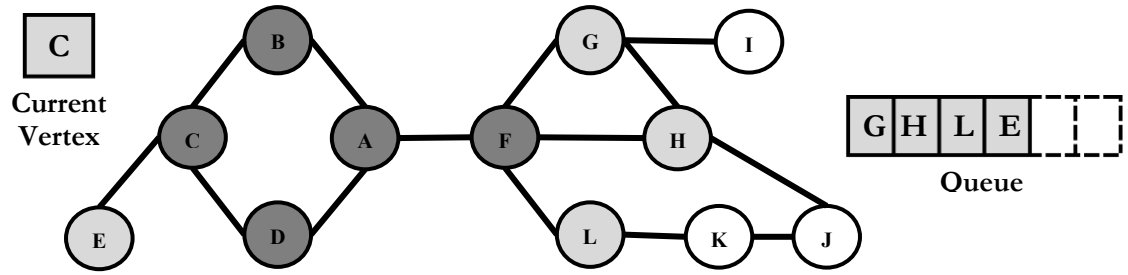
Vertex **D** is now at the front of the **queue**, so we pop it off and set it as our current vertex. We then push all of vertex **D**'s unvisited neighbors into the **queue**. Vertex **D** is connected to vertices **A** and **C**, but both **A** and **C** have been marked as visited. Thus, nothing is pushed into the **queue** at this step.



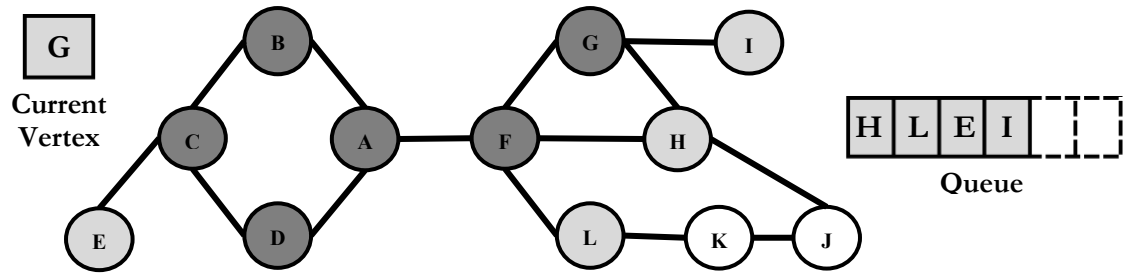
Vertex **F** is now at the front of the **queue**, so we pop it off and set it as our current vertex. We then push all of vertex **F** 's unvisited neighbors into the **queue**. Vertex **F** is connected to vertices **A, G, H**, and **L**, but only vertex **A** has been marked as visited. Thus, vertices **G, H,** and **L** are pushed into the **queue** and marked as visited.



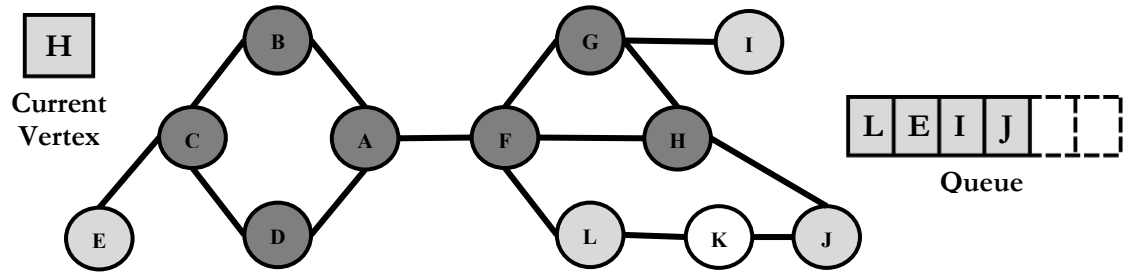
Vertex **C** is now at the front of the **queue**, so we pop it off and set it as our current vertex. We then push all of vertex **C**'s unvisited neighbors into the **queue**. Vertex **C** is connected to vertices **B, D,** and **E**, but only vertex **E** has not yet been marked as visited. Thus, vertex **E** is pushed into the **queue** and marked as visited.



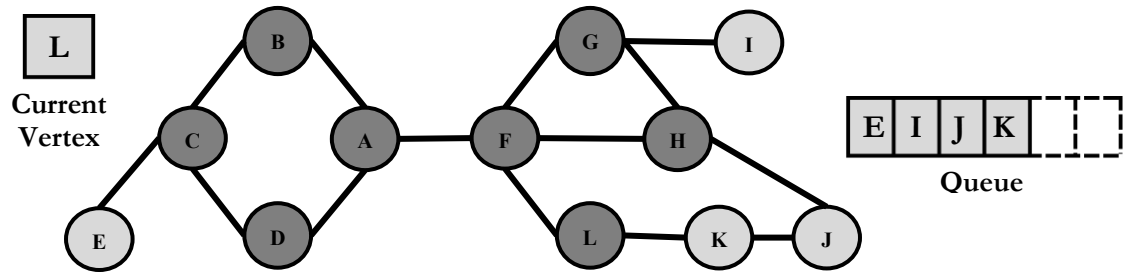
Vertex **G** is now at the front of the **queue**, so we pop it off and set it as our current vertex. We then push all of vertex **G**'s unvisited neighbors into the **queue**. Vertex **G** is connected to vertices **F, H**, and **I**, but only vertex **I** has not yet been marked as visited. Thus, vertex **I** is pushed into the **queue** and marked as visited.



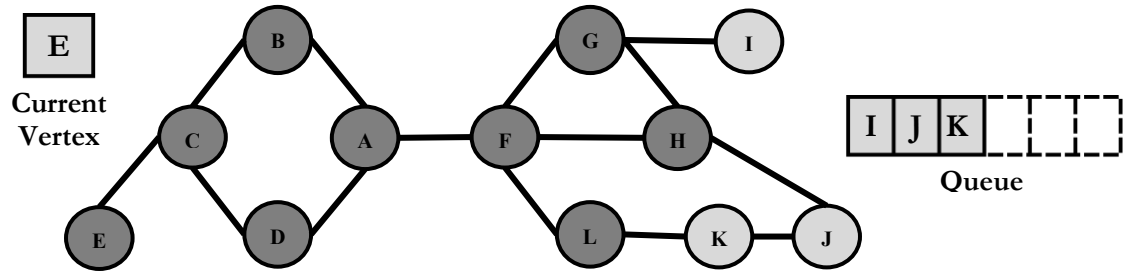
Vertex **H** is now at the front of the **queue**, so we pop it off and set it as our current vertex. We then push all of vertex **H**'s unvisited neighbors into the **queue**. Vertex **H** is connected to vertices **F, G**, and **J**, but only vertex **J** has not yet been marked as visited. Thus, vertex **J** is pushed into the **queue** and marked as visited.



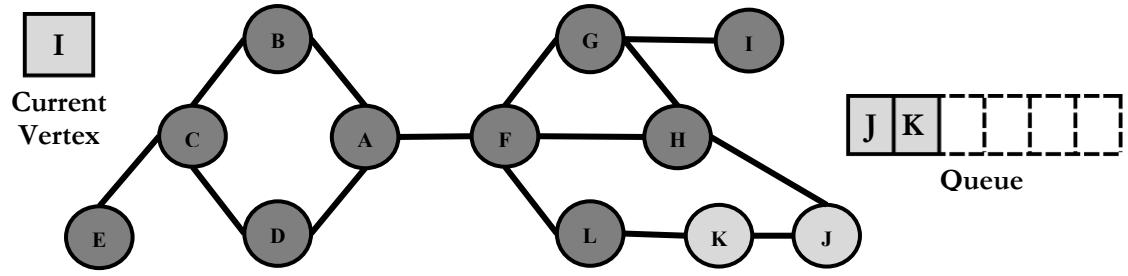
Vertex **L** is now at the front of the **queue**, so we pop it off and set it as our current vertex. We then push all of vertex **L**'s unvisited neighbors into the **queue**. Vertex **L** is connected to vertices **F** and **K**, but only vertex **K** has not yet been marked as visited. Thus, vertex **K** is pushed into the **queue** and marked as visited.



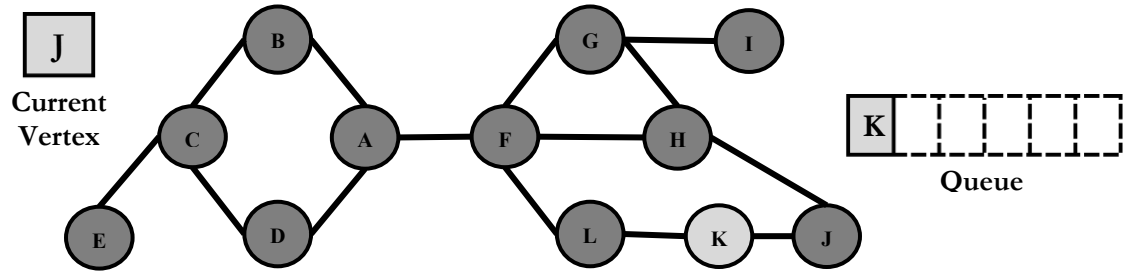
Vertex **E** is now at the front of the **queue**, so we pop it off and set it as our current vertex. We then push all of vertex **E**'s unvisited neighbors into the **queue**. Vertex **E** is connected to vertex **C**, which has already been marked as visited. Thus, nothing is pushed into the **queue** at this step.



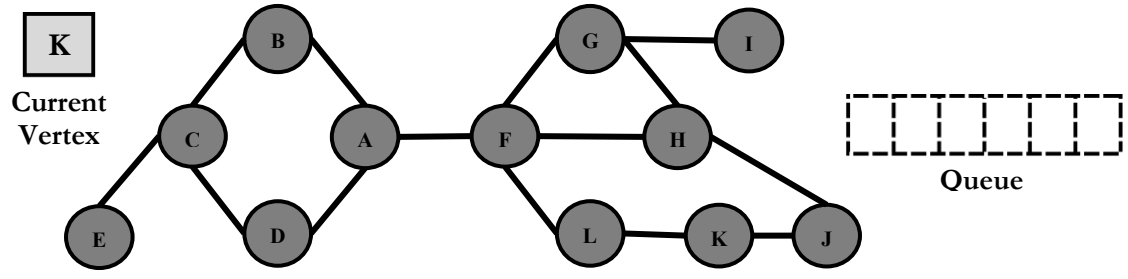
Vertex **I** is now at the front of the **queue**, so we pop it off and set it as our current vertex. We then push all of vertex **I**'s unvisited neighbors into the **queue**. Vertex **I** is connected to vertex **G**, which has already been marked as visited. Thus, nothing is pushed into the **queue** at this step.



Vertex **J** is now at the front of the **queue**, so we pop it off and set it as our current vertex. We then push all of vertex **J**'s unvisited neighbors into the **queue**. Vertex **J** is connected to vertices **H** and **K**, but both have already been visited. Thus, nothing is pushed into the **queue** at this step.



Vertex **K** is now at the front of the **queue**, so we pop it off and set it as our current vertex. We then push all of vertex **K**'s unvisited neighbors into the **queue**. Vertex **K** is connected to vertices **J** and **L**, but both have already been visited. Thus, nothing is pushed into the **queue** at this step.

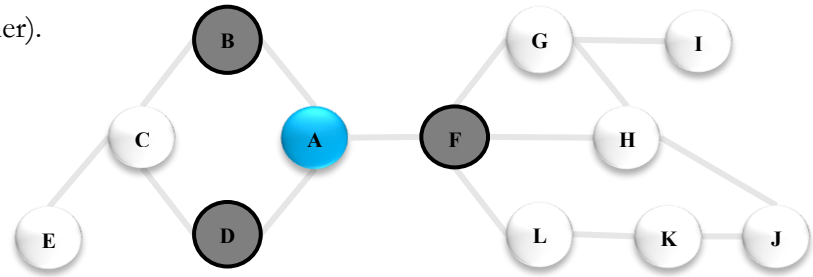


The **queue** is now empty, so our breadth-first search is complete, and we have successfully processed every vertex in the graph. **The BSF traversal order of the vertices is as follows:**

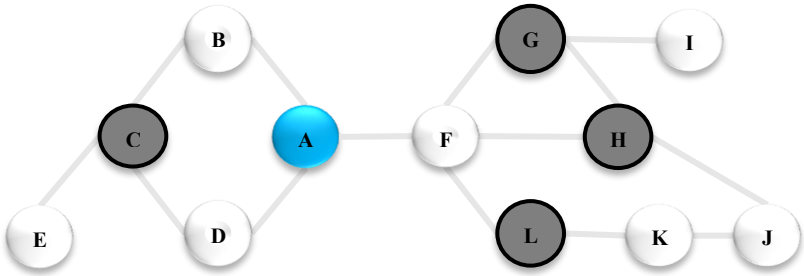
A → B → D → F → C → G → H → L → E → I → J → K

Because a breadth-first search relies on a **queue** to process nodes in a graph, the algorithm processes vertices in the graph in **FIFO** order. As a result, a breadth-first search ends up visiting vertices in order of increasing edge distance from the source vertex: vertices that are closer to the source vertex are pushed into the **queue before** vertices that are farther away, and are thus taken out of the queue and processed earlier as well!

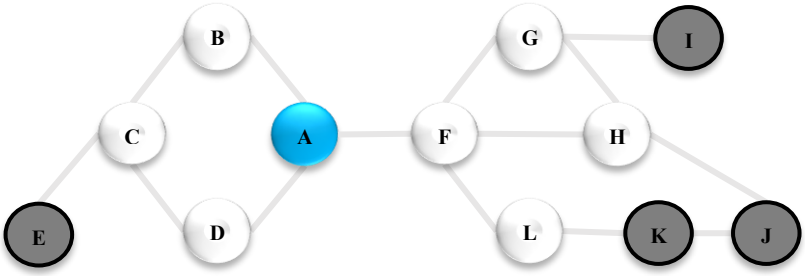
In our example, the breadth-first search first processed vertices that were one edge away from the source vertex **A**: vertices **B, D,** and **F** (in this order, since we pushed them into the queue in alphabetical order).



Then, the breadth-first search processed vertices that were two edges away from the source vertex **A**:



Then, the breadth-first search processed vertices that were three edges away from the source vertex **A**:



Had the graph been larger, the breadth-first search would then have processed vertices four edges away, then five edges away, and so on. Because breadth-first searches exhibit this behavior, they can always be used to find the **shortest path between any two vertices in an unweighted graph, or a graph where all edge weights are the same.**

As a result, breadth-first searches are often useful for solving shortest path graph problems, particularly for unweighted graphs.

V.5.3. DFS and BFS Time Complexity

Graph Representation	Adjacency List	Adjacency Matrix
DFS Time Complexity	$\mathcal{O}(\|V\| + \|E\|)$	$\mathcal{O}(\|V\|^2)$
BFS Time Complexity	$\mathcal{O}(\|V\| + \|E\|)$	$\mathcal{O}(\|V\|^2)$