Data Structures & Algorithms

Week 12

Graphs

Lecturer: Dr. Msagha J Mbogholi, PhD

Flashback from Lesson 11

- Elements of indexed linear data structures, such as lists, are ordered—the first element (at index 0), second element (at index 1), and so forth. In contrast, the elements of an associative data structure are unordered, instead accessed by an associated key value. In Python, an associative data structure is provided by the *dictionary type.*
- The operations that can be performed on dictionaries relate to their properties, namely: mutability, associative nature, keys, values, and no duplication.
- A set is a mutable data type with nonduplicate, unordered values, providing the usual mathematical set operations.
- The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set. This is based on a data structure known as a hash table.
- Python provides another built-in type called a **frozenset**, which is in all respects exactly like a set, except that a frozenset is immutable. You can perform non-modifying operations on a frozenset

Content

- Terminology
- Data structures
- Graph traversals
- Shortest paths
- Minimum spanning trees

Part 1

Terminology

1.1 Introduction

- Our exposure to the term graph is limited to the data visualization object that is used to explain data patterns…histograms, line graphs, bar graphs, and so on.
- However, we do have graphs that are used as a data structure.
- The graphs in this domain are used to explain relationships between different objects.
- We apply graphs in many real life situations as we shall see in this lesson; most of the time it is just that we are not aware of it.
- We begin the lesson by explaining some terms related to graphs, then go on to describe the different operations applicable to graphs.

1.2 Terminology

- The terminology related to graphs is found in several literature. We use the definitions shared by (Goodrich et al., 2018):
- Definition: A ***graph*** is a way of representing relationships that exist between pairs of objects. That is, a graph is a set of objects, called vertices, together with a collection of pairwise connections between them, called edges.
- Viewed abstractly, a ***graph*** *G* is simply a set *V* of ***vertices (also known as nodes)*** and a collection *E* of pairs of vertices from *V*, called ***edges (also known as arcs)***. Thus, a graph is a way of representing connections or relationships between pairs of objects from some set *V*.
- Directed edges: An edge (*u*,*v*) is said to be ***directed*** from *u* to *v* if the pair (*u*,*v*) is ordered, with *u* preceding *v*.
- Undirected edges: An edge (*u*,*v*) is said to be ***undirected*** if the pair (*u*,*v*) is not ordered. Undirected edges are sometimes denoted with set notation, as {*u*,*v*}, but for simplicity we use the pair notation (*u*,*v*), noting that in the undirected case (*u*,*v*) is the same as (*v*,*u*).

1.2 Terminology

- Semantics: Graphs are typically visualized by drawing the vertices as ovals or rectangles and the edges as segments or curves connecting pairs of ovals and rectangles.
- Undirected graph: If all the edges in a graph are undirected, then we say the graph is an ***undirected graph***.
- Directed graph: if all the edges in a graph are directed, then we call it a directed graph, or a **digraph.**
- Mixed graph: one that has both directed and undirected edges.
- An undirected or mixed graph can be converted into a directed graph by replacing every undirected edge (u,v) by the pair of directed edges (u,v) and (v,u).
- End vertices (of an edge): The two vertices joined by an edge are called the ***end vertices*** (or ***endpoints***) of the edge. Further, if an edge is directed, its first endpoint is its ***origin*** and the other is the ***destination*** of the edge.

1.2 Terminology

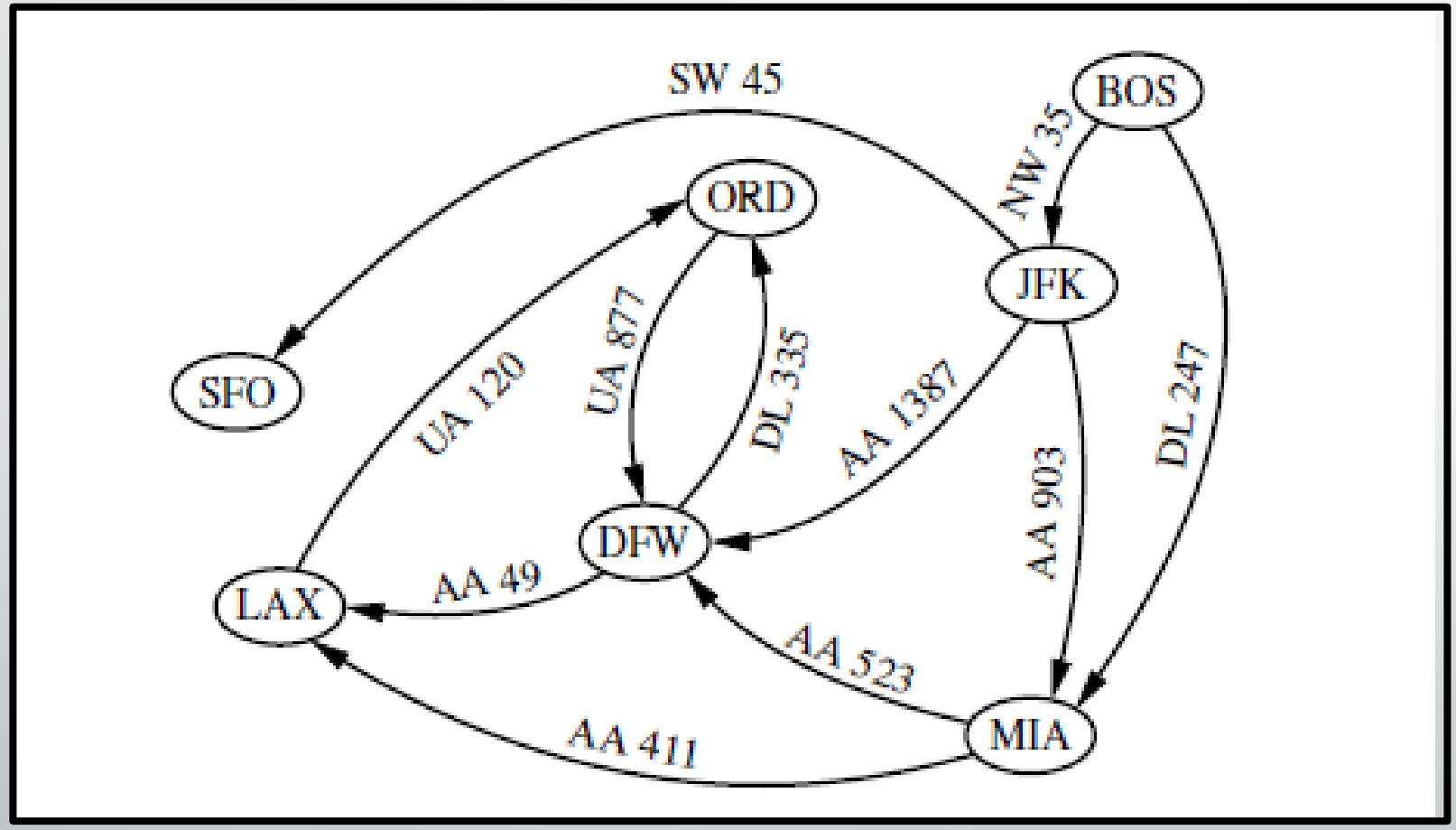
- Adjacency: Two vertices *u* and *v* are said to be ***adjacent*** if there is an edge whose end vertices are *u* and *v*.
- Incidence: An edge is said to be ***incident*** to a vertex if the vertex is one of the edge's endpoints.
- Incoming edge: the ***incoming edges*** of a vertex are the directed edges whose destination is that vertex.
- Outgoing edge: The ***outgoing edges*** of a vertex are the directed edges whose origin is that vertex.
- Degree: The ***degree*** of a vertex *v*, denoted deg(*v*), is the number of incident edges of *v*. Further, The ***in-degree*** and ***out-degree*** of a vertex *v* are the number of the incoming and outgoing edges of *v*, and are denoted indeg(*v*) and outdeg(*v*), respectively.
- Figure 1 shows a graph G whose vertices are associated with airports and edges associated with flights. We note that that the flights are directed showing the direction of the flight from source to destination.

1.2 Terminology

- Goodrich et al.(2018) explain graph G as follows:
- "The endpoints of an edge *e* in *G* correspond respectively to the origin and destination of the flight corresponding to *e*. Two airports are adjacent in *G* if there is a flight that flies between them, and an edge *e* is incident to a vertex *v* in *G* if the flight for *e* flies to or from the airport for *v*. The outgoing edges of a vertex *v* correspond to the outbound flights from *v*'s airport, and the incoming edges correspond to the inbound flights to *v*'s airport. Finally, the in-degree of a vertex *v* of *G* corresponds to the number of inbound flights to *v*'s airport, and the out-degree of a vertex *v* in *G* corresponds to the number of outbound flights."
- Based on the above explanation it can be seen that the endpoints of edge SW45 are SFO and JFK; hence, SFO and JFK are adjacent. The in-degree of ORD is 2, and the out-degree of DFWis 1.
- It can also be observed that BOS has no in-degree, but an out-degree of 2. This means there are no flights going to Boston; rather only outgoing flights.

Figure 1

Graph G of flight network



(From Goodrich et al., 2018)

1.2 Terminology

- Parallel (or multiple) edges: The definition of a graph refers to the group of edges as a ***collection***, not a ***set***, thus allowing two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called ***parallel edges*** or ***multiple edges***. (Goodrich et al., 2018)
- In figure 1 it is possible to have parallel edges; suppose there are two flights per day at different times between JFK and MIA? We could show these using two parallel edges to show demonstrate this (two directed edges from JFK to MIA that is).
- Self loop: an edge (undirected or directed) is a ***self-loop*** if its two endpoints coincide. (happens very rarely).
- Path: A **path** is a sequence of edges that allows one vertex to be reached from another vertex in a graph. Thus, a vertex is **reachable** from another vertex if and only if there is a path between the two. (Lambert, 2014). Alternatively, a ***path*** is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex. (Goodrich et al., 2018)

1.2 Terminology

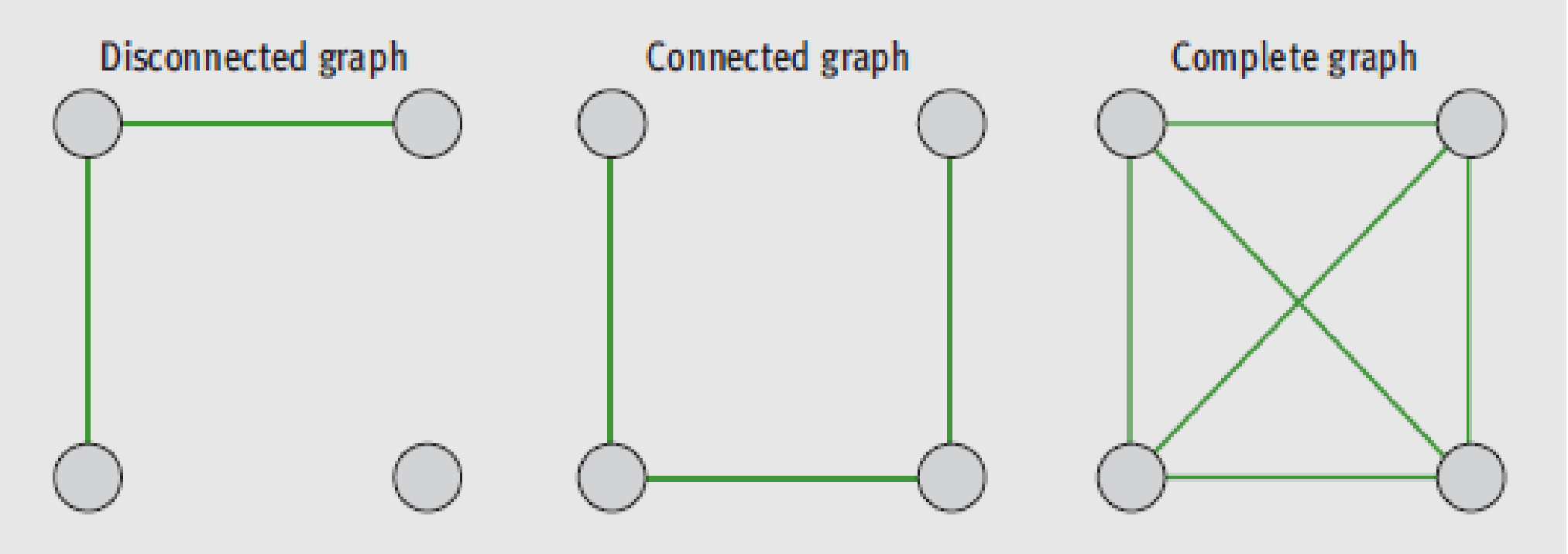
- Some more terminology on path from Lambert (2014) and Goodrich et al. (2018):
- A ***cycle*** is a path that starts and ends at the same vertex, and that includes at least one edge.
- We say that a path is ***simple*** if each vertex in the path is distinct, and we say that a cycle is ***simple*** if each vertex in the cycle is distinct, except for the first and last one.
- The **length of a path** is the number of edges on the path.
- A ***directed path*** is a path such that all edges are directed and are traversed along their direction. A ***directed cycle*** is similarly defined. In figure 1 (BOS, DL247, MIA, AA 411, LAX) is a directed path, while (LAX, UA 120, ORD, UA 877, DFW, AA 49, LAX) is a directed simple cycle; the same with (DFW, DL335, ORD, UA877, DFW).
- A directed graph is ***acyclic*** if it has no directed cycles; it is then referred to as directed acyclic graph (DAG). For example, if we were to remove the edge UA 877 from figure 1 , the remaining graph is acyclic.

1.2 Terminology

- Lambert (2014) also defines the following terminology:-
- Simple path: A **simple path** is a path that does not pass through the same vertex more than once.
- Connected graph: A graph is **connected** if there is a path from each vertex to every other vertex.
- Complete graph: A graph is **complete** if there is an edge from each vertex to every other vertex.
- A graph that has relatively many edges is called a **dense graph**, whereas one that has relatively few edges is called a **sparse graph.**
- The number of edges in a complete directed graph with N vertices is $N * (N - 1)$, and the number of edges in a complete undirected graph is $N * (N - 1) / 2$.
- Figure 2 illustrates the difference between a disconnected, connected, and complete graph.

Figure 2

Graph types



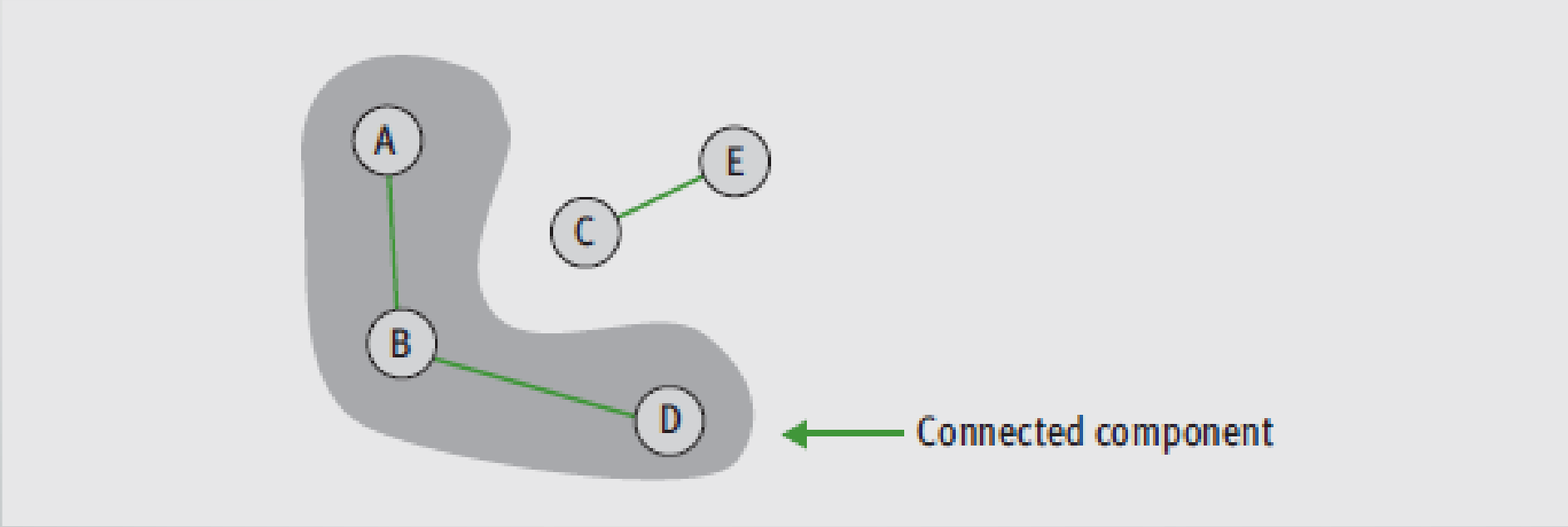
(From Lambert, 2014)

1.2 Terminology

- Subgraph: A **subgraph** of a given graph consists of a subset of that graph's vertices and the edges connecting those vertices. (Lambert, 2014).
- Spanning subgraph: A ***spanning subgraph*** of (a graph) *G* is a subgraph of *G* that contains all the vertices of the graph *G*. If a graph *G* is not connected, its maximal connected subgraphs are called the ***connected components*** of *G*. (Goodrich et al., 2018). Therefore, a **connected component** is a subgraph consisting of the set of vertices that are reachable from a given vertex. (Lambert, 2014)
- Forest: this is a graph without cycles
- Tree: this is a connected forest, meaning a graph without cycles.
- Spanning tree: is a spanning subgraph that is a tree.
- Figure 3 shows a disconnected graph with vertices A, B, C, D, and E and the connected component that contains the vertex B.

Figure 3

Subgraph of connected component



(From Lambert, 2014)

Part 2

Data structures

2.1 Graph ADT

- Though there are different suggestions on the methods for the graph ADT (such as https://runestone.academy/ns/books/published/pythonds/Graphs/TheGraphAbstractDataType.html, and several implementations using either dictionaries or sets, for example, https://www.geeksforgeeks.org/generate-graph-using-dictionary-python/), in this course we use the one provided by Goodrich et al. (2018):"
- We model the abstraction as a combination of three data types: Vertex, Edge, and Graph. A Vertex is a lightweight object that stores an arbitrary element provided by the user (e.g., an airport code); we assume it supports a method, element(), to retrieve the stored element. An Edge also stores an associated object (e.g., a flight number, travel distance, cost), retrieved with the element() method.
- In addition, we assume that an Edge supports the following methods:
- endpoints( ): Return a tuple (u,v) such that vertex u is the origin of the edge and vertex v is the destination; for an undirected graph, the orientation is arbitrary.

2.1 Graph ADT

- opposite(v)**:** Assuming vertex *v* is one endpoint of the edge (either origin or destination), return the other endpoint.
- The primary abstraction for a graph is the Graph ADT. We presume that a graph can be either ***undirected*** or ***directed***, with the designation declared upon construction; recall that a mixed graph can be represented as a directed graph, modeling edge {*u*,*v*} as a pair of directed edges (*u*,*v*) and (*v*,*u*). The Graph ADT includes the following methods:
- vertex count()**:** Return the number of vertices of the graph.
- vertices( )**:** Return an iteration of all the vertices of the graph.
- edge count()**:** Return the number of edges of the graph.
- edges( )**:** Return an iteration of all the edges of the graph.

2.1 Graph ADT

- get edge(u,v)**:** Return the edge from vertex *u* to vertex *v*, if one exists; otherwise return None. For an undirected graph, there is no difference between get edge(u,v) and get edge(v,u).
- degree(v, out=True)**:** For an undirected graph, return the number of edges incident to vertex *v*. For a directed graph, return the number of outgoing (resp. incoming) edges incident to vertex *v*, as designated by the optional parameter.
- incident edges(v, out=True)**:** Return an iteration of all edges incident to vertex *v*. In the case of a directed graph, report outgoing edges by default; report incoming edges if the optional parameter is set to False.
- insert vertex(x=None)**:** Create and return a new Vertex storing element *x*.

2.1 Graph ADT

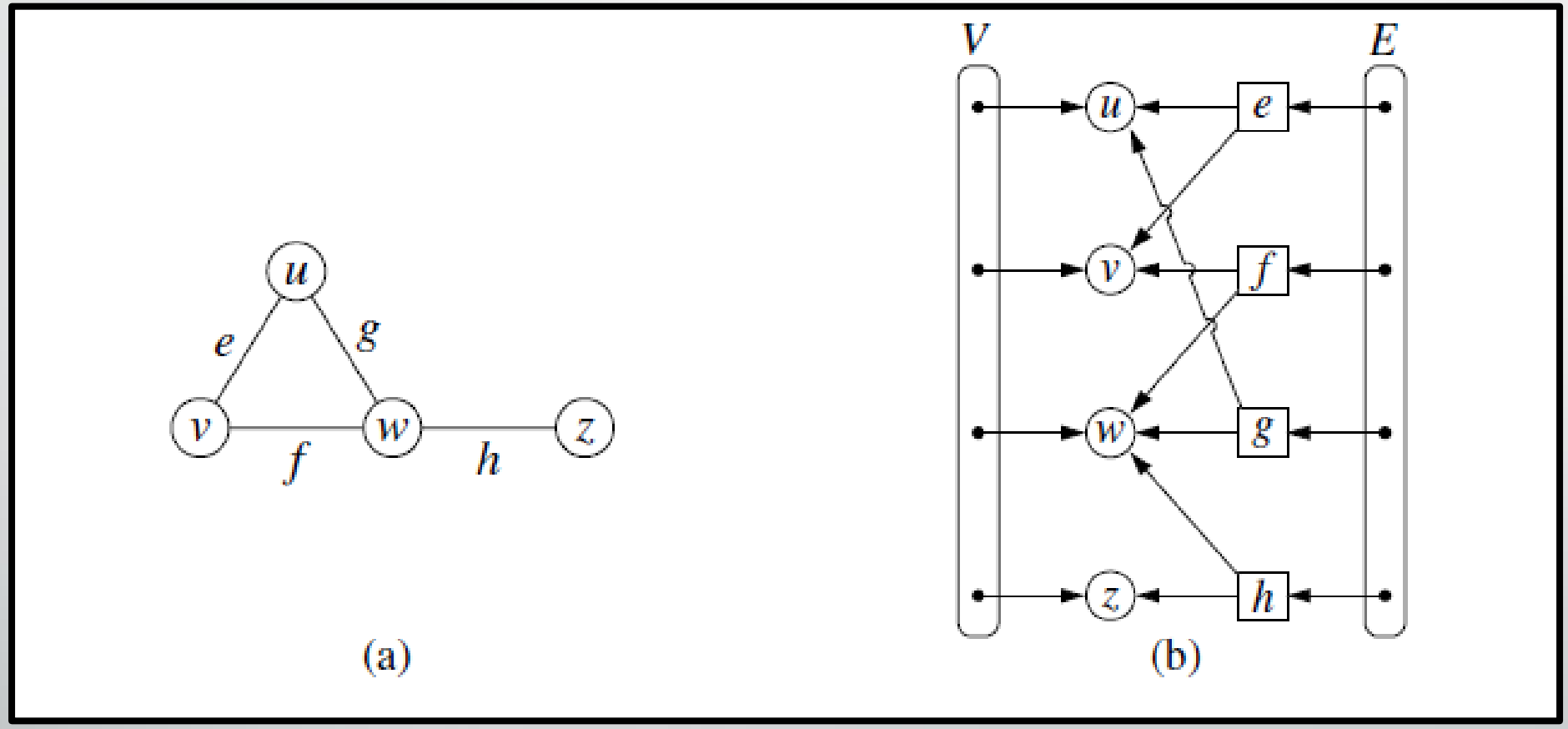
- insert edge(u, v, x=None)**:** Create and return a new Edge from vertex *u* to vertex *v*, storing element *x* (None by default).
- remove vertex(v)**:** Remove vertex *v* and all its incident edges from the graph.
- remove edge(e)**:** Remove edge *e* from the graph."
- We may also add the following methods:
- Graph(): creates a new graph
- **in** returns True for a statement of the form vertex in graph, if the given vertex is in the graph, False otherwise.
- addVertex(v): adds an instance of Vertex to the graph.

2.2 Edge list

- The ***edge list*** structure is possibly the simplest, though not the most efficient, representation of a graph *G*. All vertex objects are stored in an unordered list *V*, and all edge objects are stored in an unordered list *E*. (Goodrich et al., 2018).
- Figure 4 demonstrates this concept. Vertex objects u, v, w, and z are placed in unordered list V while edge objects e, f, g, and h are placed in unordered list E. An edge object refers to vertices (which it is connected to; endpoints that is) and is connected to them, while vertices do not connect to the edges.
- (AlgoDaily, n.d.) describes the edge list using a matrix: "An edge list is a list or array of all the edges in a graph…nthe underlying data structure for keeping track of all the nodes and edges is a single list of pairs. Each pair represents a single edge and is comprised of the two unique IDs of the nodes involved. Each line/edge in the graph gets an entry in the edge list, and that single data structure then encodes all nodes and relationships. "
- Figure 5 demonstrates this representation: there are we have three nodes: 1, 2, and 3. Each edge is given an index and represents a reference from one node to another. There isn't any particular order to the edges as they appear in the edge list, but every edge must be represented.

Figure 4

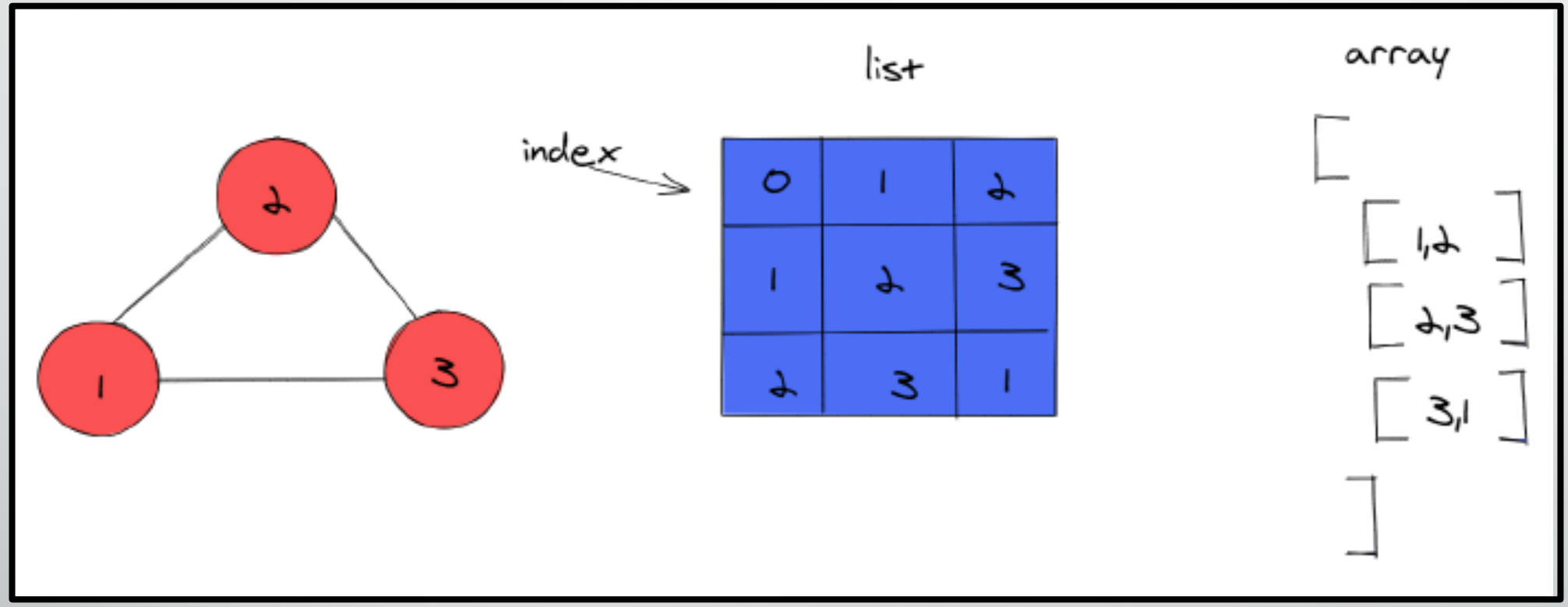
Edge list



(From Goodrich et al., 2018)

Figure 5

Edge list matrix representation



(From AlgoDaily, n.d.)

2.2 Edge list

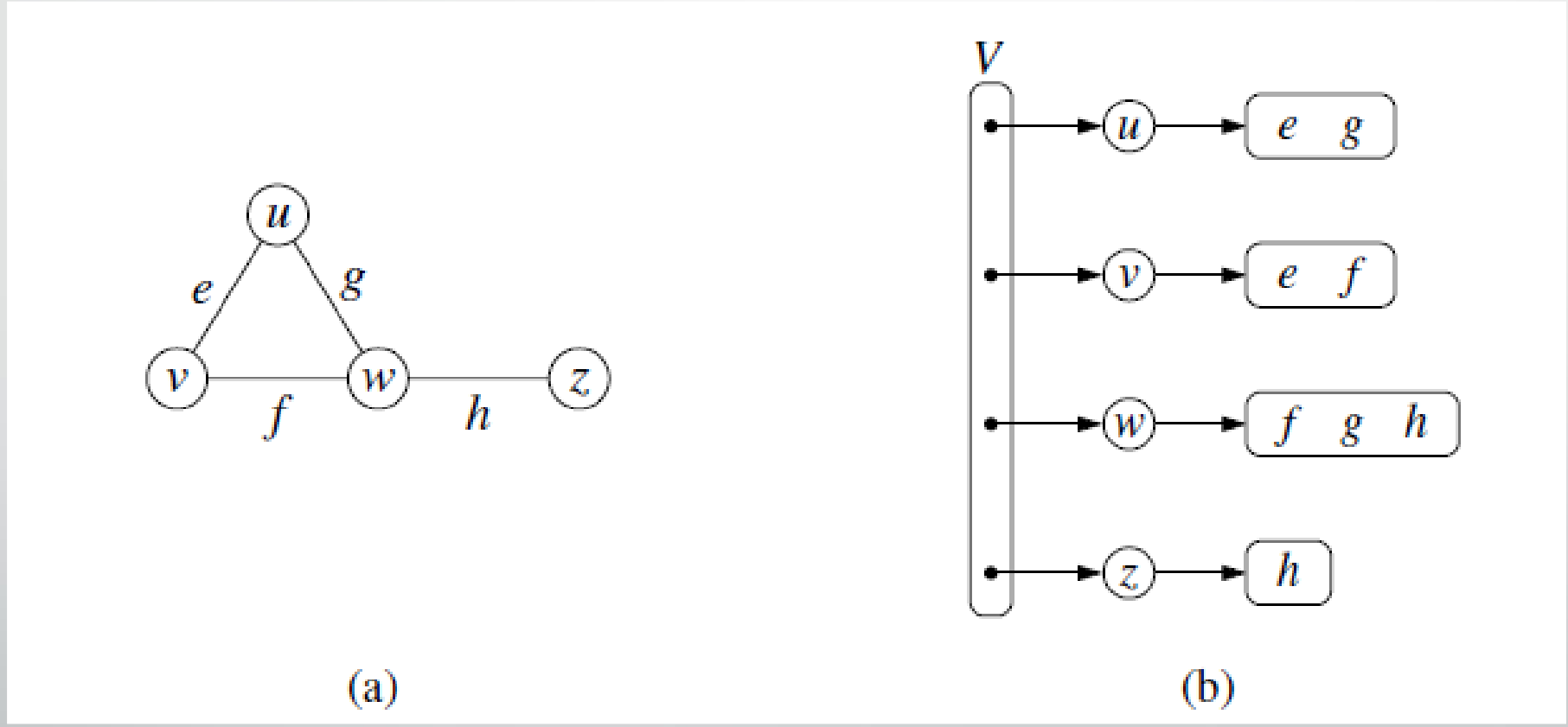
- The Python implementation of the edge list is as follows (from figure 5), courtesy Algodaily(n.d.):
- edge_list = [2 [1,2], 3 [2,3], 4 [3,1] 5]
- The methods to be used will be the same as those defined in part 2.1 for the graph ADT. For example to add edges (after defining the method):
- our_graph.add_edge(v0, v1, 2) #where our_graph is the name of the graph, for example, G.
- our_graph.add_edge(v1, v2, 3)
- our_graph.add_edge(v2, v0, 1)
- our_graph.add_edge(v2, v3, 1)
- our_graph.add_edge(v3, v2, 4)
- Due to time and space limitations it is not possible to examine the full Python implementation from the beginning…this is left as further reading for the learner.

2.3 Adjacency list

- "The ***adjacency list*** structure groups the edges of a graph by storing them in smaller, secondary containers that are associated with each individual vertex. Specifically, for each vertex *v*, we maintain a collection *I*(*v*), called the ***incidence collection*** of *v*, whose entries are edges incident to *v*. (In the case of a directed graph, outgoing and incoming edges can be respectively stored in two separate collections, *I*out(*v*) and *I*in(*v*).) Traditionally, the incidence collection *I*(*v*) for a vertex *v* is a list, which is why we call this way of representing a graph the ***adjacency list*** structure." (Goodrich et al., 2018)
- Figure 6 illustrates this: the edges connected to vertex v are e and f; similarly for vertex u, it is edges e and g; in a similar manner adjacent edges for the remaining vertices are found.
- Algodaily (n.d.) also demonstrates this using the matrix representation. If there is an edge between two nodes it is represented by a 1, while if there is none, it is represented by a 0. This is illustrated in figure 7.

Figure 6

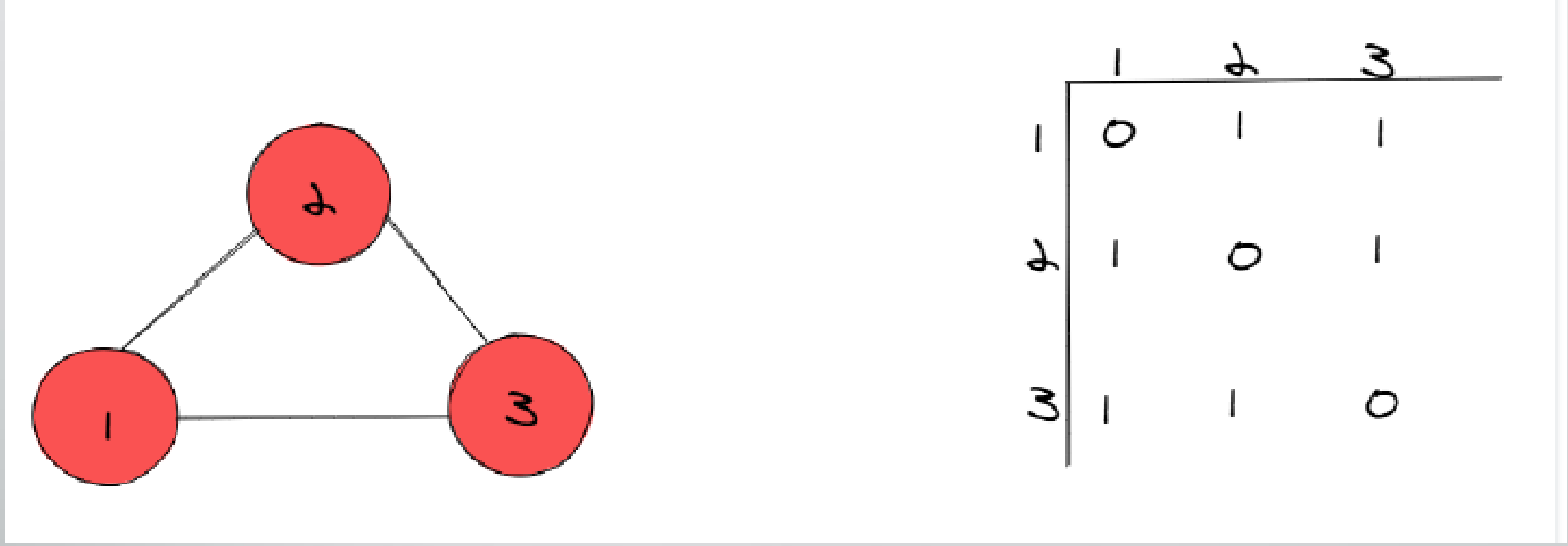
Adjacency list



(From Goodrich et al., 2018)

Figure 7

Adjacency list matrix representation

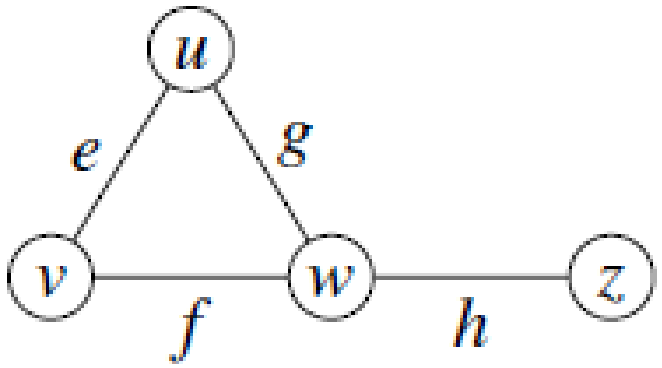


(From Algodaily, n.d.)

- Another way of depicting the adjacency matrix is shown in figure 8. We represent the vertices using index numbers 0…(n-1), where n is the number of vertices in the graph; the matrix is then filled by naming the edges between the vertices. If there is no edge between vertices it is left blank.

Figure 8

Adjacency matrix



		0	1	2	3
u	0		e	g	
v	1	e		f	
w	2	g	f		h
z	3			h	

(From Goodrich et al., 2018)

Part 3

Graph traversal

3.1 Introduction

- To traverse means to travel across or through (Michel traversed the country, for example); it also means to move back and forth or sideways (the sewing needle traversed the fabric).
- This is exactly what we seek to do with our graphs; we would like to traverse them in order to perform different kinds of operations. The key is to try and find a way of reaching the targeted vertices (nodes) in order to get what is stored in them, for example. Goodrich et al. (2018) share some of these reachability problems; let us sample two such problems for directed graphs and two for undirected graphs.
- Computing a directed path from vertex *u* to vertex *v*, or reporting that no such path exists. (directed graphs)
- Finding all the vertices of *G* that are reachable from a given vertex *s*. (directed graphs)
- Computing a path from vertex *u* to vertex *v*, or reporting that no such path exists. (undirected graphs)
- Computing the connected components of *G*. (undirected graphs)
- Two popular traversal algorithms are described that are designed to solve exactly this kind of problem.

3.2 Depth first search (DFS)

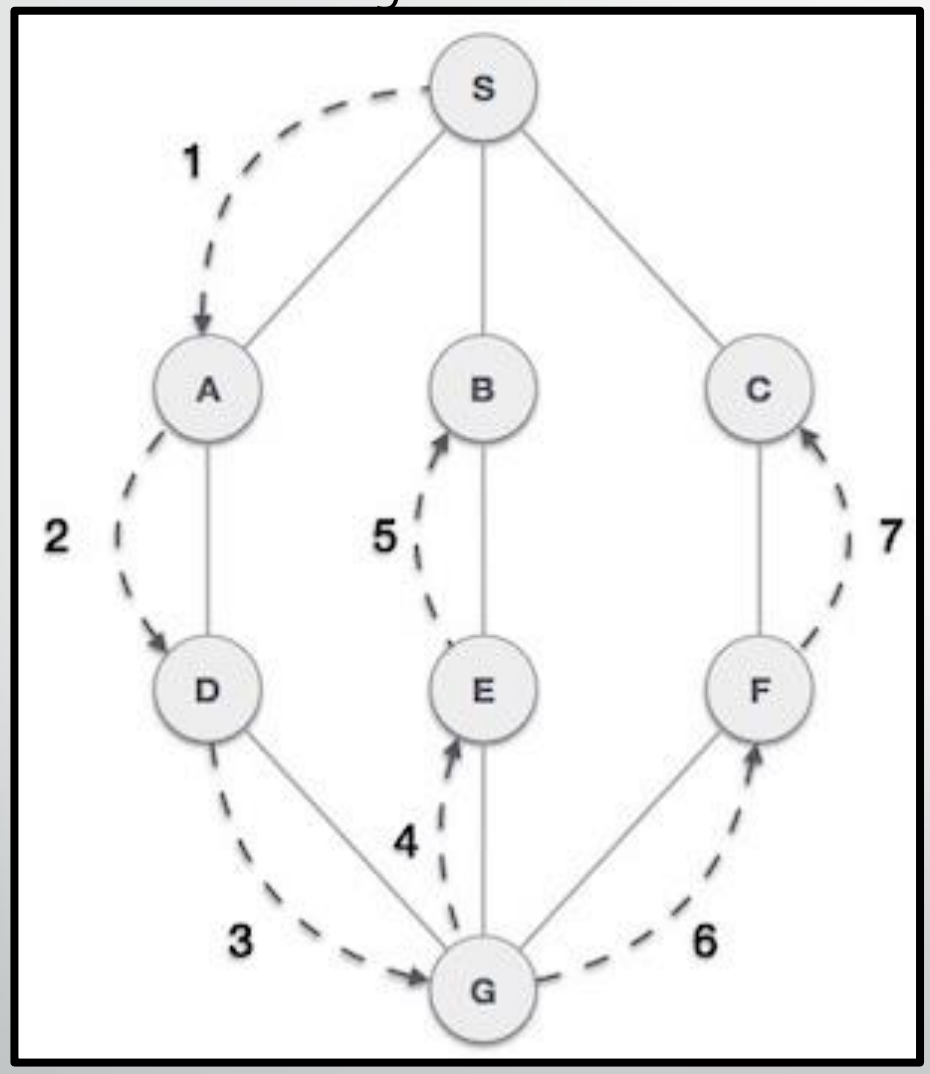
- Recall that traversal algorithms are well, for traversing the object, in this case a graph. Since we are aware of the properties and terminologies associated with graphs, we are aware that the graph may be cyclic (or not), simple, and so on.
- Our main aim is to be able to traverse the graph in the most efficient manner possible.
- The depth first search (DFS) algorithm is a popular one for traversing the graph; just as the name implies we start from the top and traverse by depth (downwards) starting from the left side of the graph (like the tree we discussed in lesson 10); once we get to the endpoint of this path and don't find the vertex (node) we are looking for we repeat the process from the top for every path, until we find the node; we must ensure that we don't traverse a node that we have already visited though.
- The DFS algorithm implements a stack to track the traversal. Let us demonstrate this using a simple example from (*Data Structure - Depth First Traversal - Tutorialspoint*, n.d.).

3.2 Depth first search (DFS)

- "Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.
- Figure 9 demonstrates the algorithm: the DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C; it uses the following rules:
- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.
- Figure 10 describes the steps as they happen in regard to the traversal and the use of the stack in implementation.

Figure 9

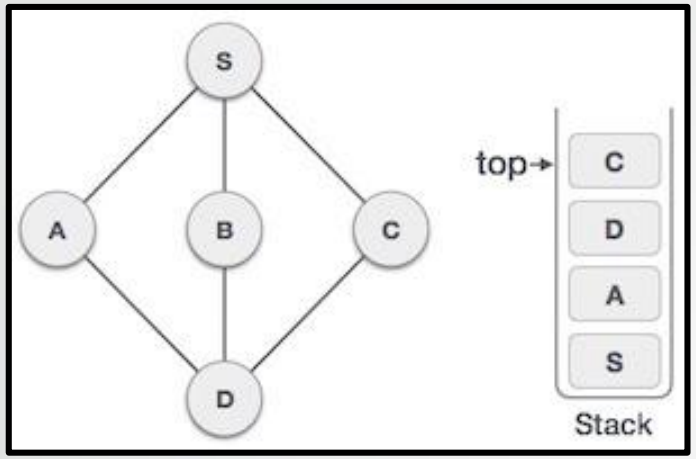
DFS traversal algorithm



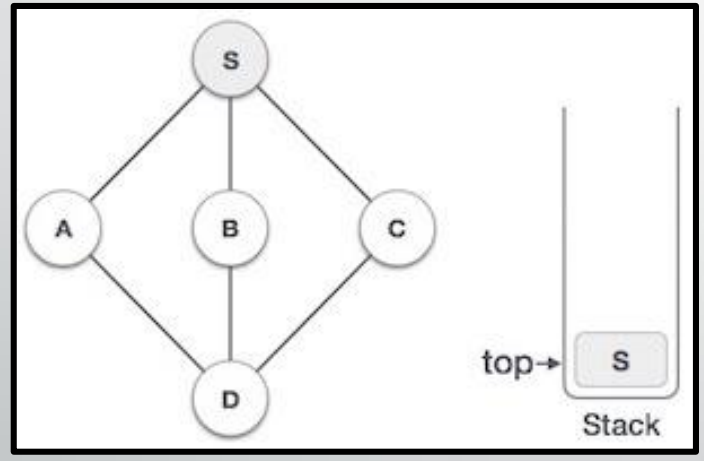
(From *Data Structure - Depth First Traversal - Tutorialspoint*, n.d.).

Figure 10 (a)

DFS traversal using stack



Begin: Step 1
Initialize the stack.

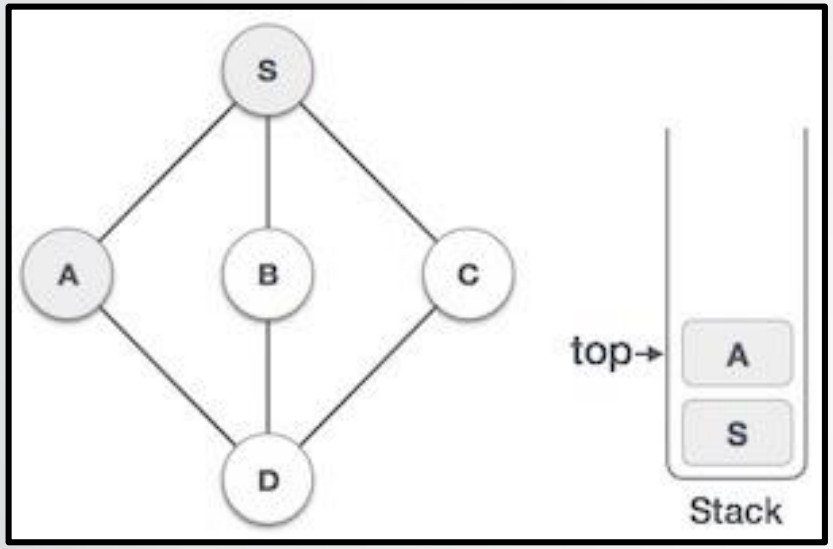


Next: Step 2
Mark **S** as visited and put it onto the stack.
Explore any unvisited adjacent node from **S**.
We have three nodes and we can pick any of them.
For this example, we shall take the node in an alphabetical order.

(From *Data Structure - Depth First Traversal - Tutorialspoint*, n.d.).

Figure 10 (b)

DFS traversal using stack



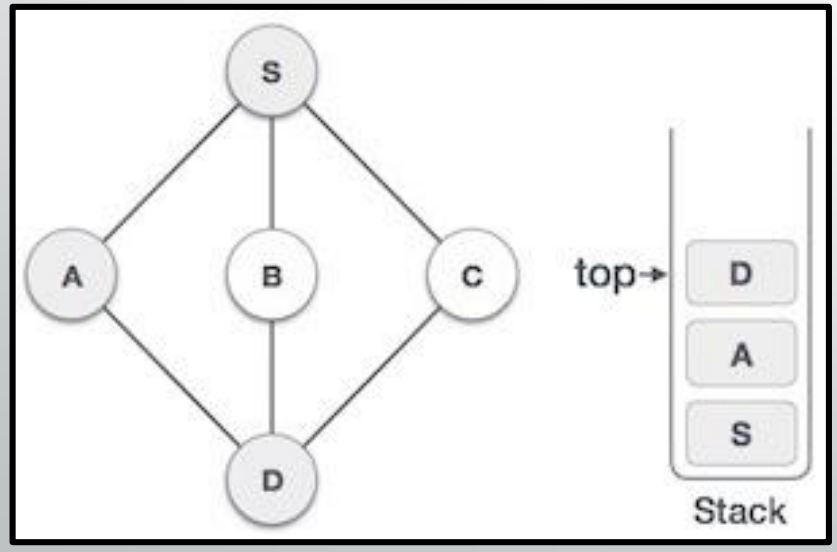
Next: Step 3

Mark **A** as visited and put it onto the stack.

Explore any unvisited adjacent node from A.

Both **S** and **D** are adjacent to **A,**

but we are concerned with unvisited nodes only.



Next: Step 4

Visit **D** and mark it as visited and put onto the stack.

Here, we have **B** and **C** nodes, which are adjacent to **D**

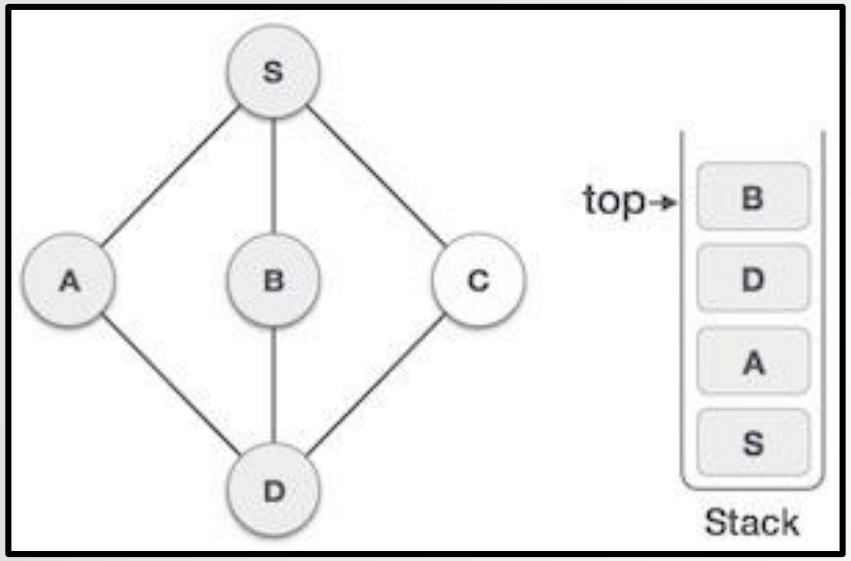
and both are unvisited.

However, we shall again choose in an alphabetical order.

(From *Data Structure - Depth First Traversal - Tutorialspoint*, n.d.).

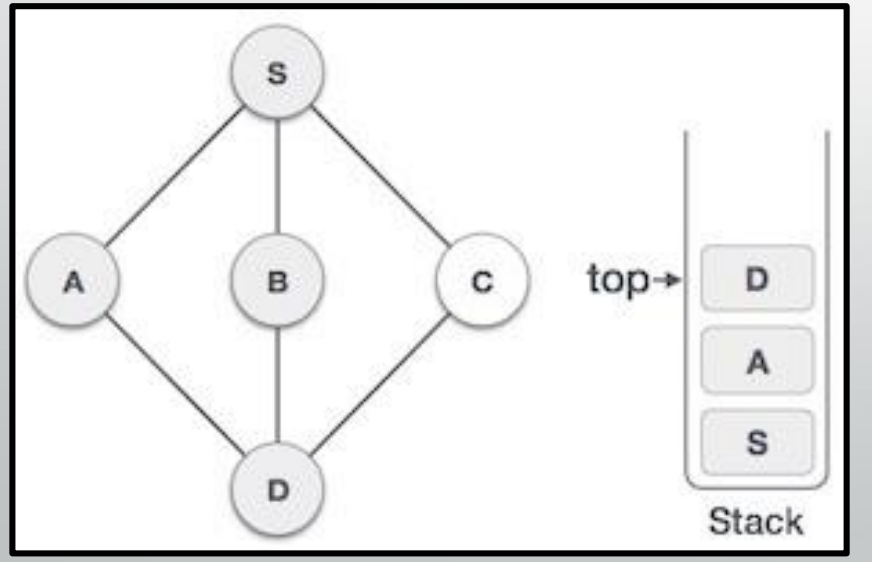
Figure 10 (c)

DFS traversal using stack



Next: Step 5

We choose **B,** mark it as visited and put onto the stack.
Here **B** does not have any unvisited adjacent node.
So, we pop **B** from the stack.

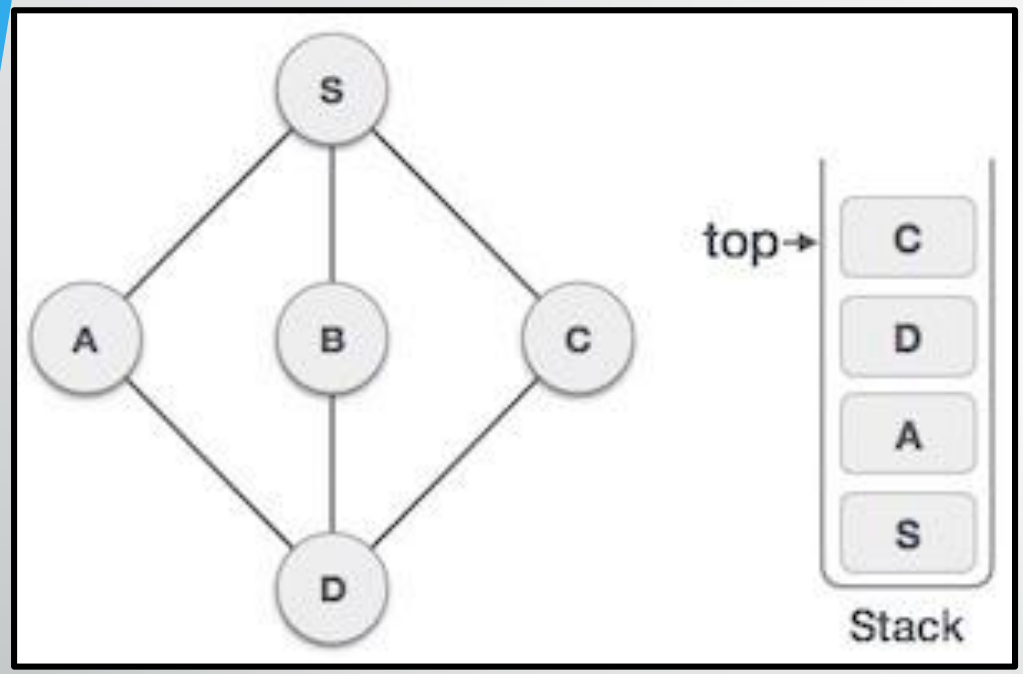


Next: Step 6

We check the stack top for return to the previous node and check if it has any unvisited nodes.
Here, we find **D** to be on the top of the stack.

(From *Data Structure - Depth First Traversal - Tutorialspoint*, n.d.).

Figure 10 (d)
DFS traversal using stack



Next: Step 7
Only unvisited adjacent node is from **D** is **C** now.
So we visit **C,** mark it as visited and put it onto the stack.
As **C** does not have any unvisited adjacent node
so we keep popping the stack until we find a node that has an unvisited adjacent node.
In this case, there's none
and we keep popping until the stack is empty.

(From *Data Structure - Depth First Traversal - Tutorialspoint*, n.d.).

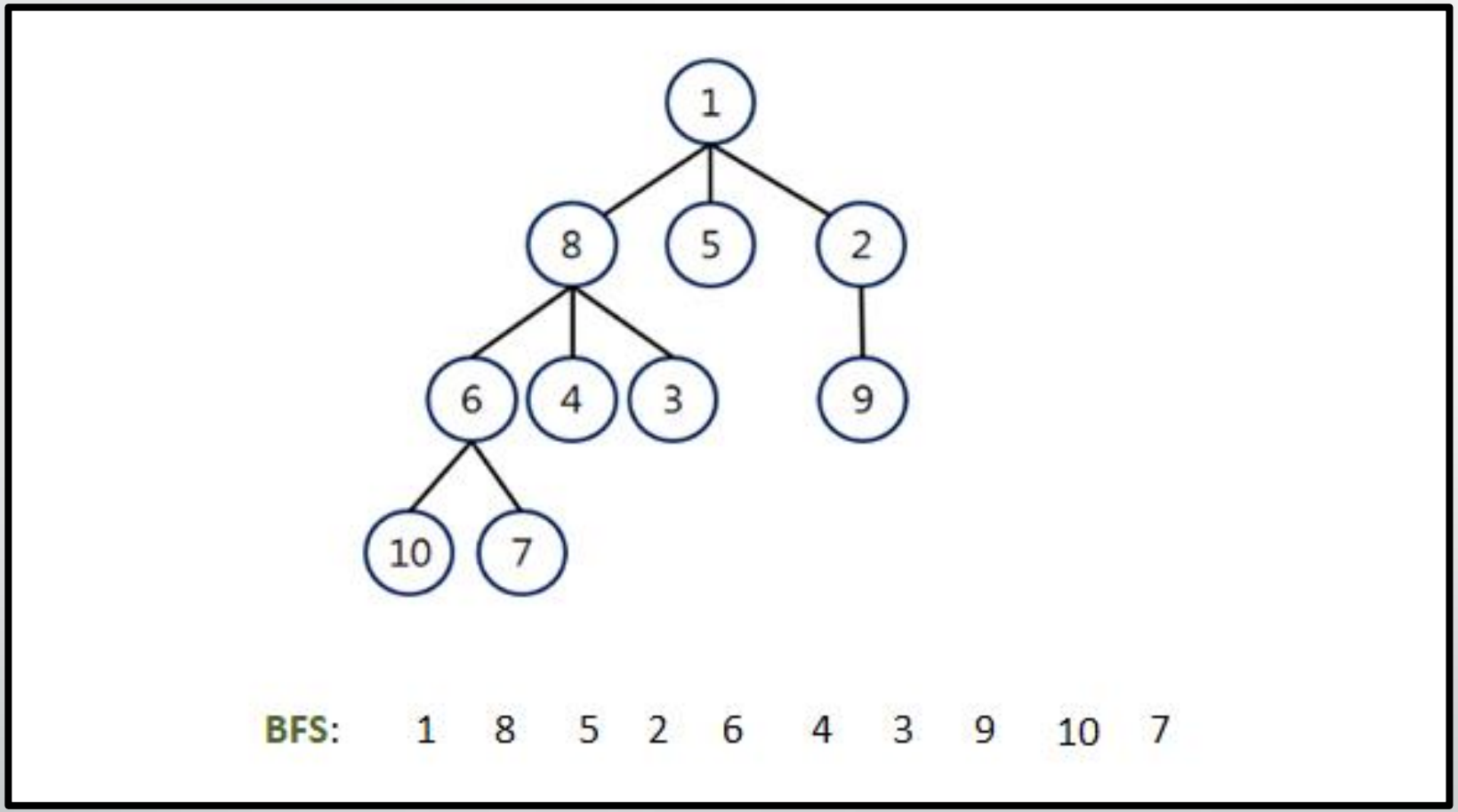
3.3 Breadth first search (BFS)

- In the DFS algorithm we 'dived' into the depth of the graph in performing our search; after finishing a path we moved to the adjacent top and repeated the process, till we got what we were searching for.
- The BFS traversal algorithm searches across the breadth of a level below moving to the level(s) below it; the process continues until the vertex is found, or the last vertex in the graph is visited (whichever comes first).
- Again let us an example from (*Data Structure - Breadth First Traversal - Tutorialspoint*, n.d.) to demonstrate the working of the algorithm.
- Unlike the DFS algorithm, this one uses the queue for its implementation.
- Figure 11 shows the order/path of traversal using a simple graph
- Figure 12 demonstrates the traversal process: we traverse from S, to A to B to C; then move to the next level and traverse from D to E then F; and lastly to G.

Figure 13 shows the step by step approach in implementing this with a queue.

Figure 11

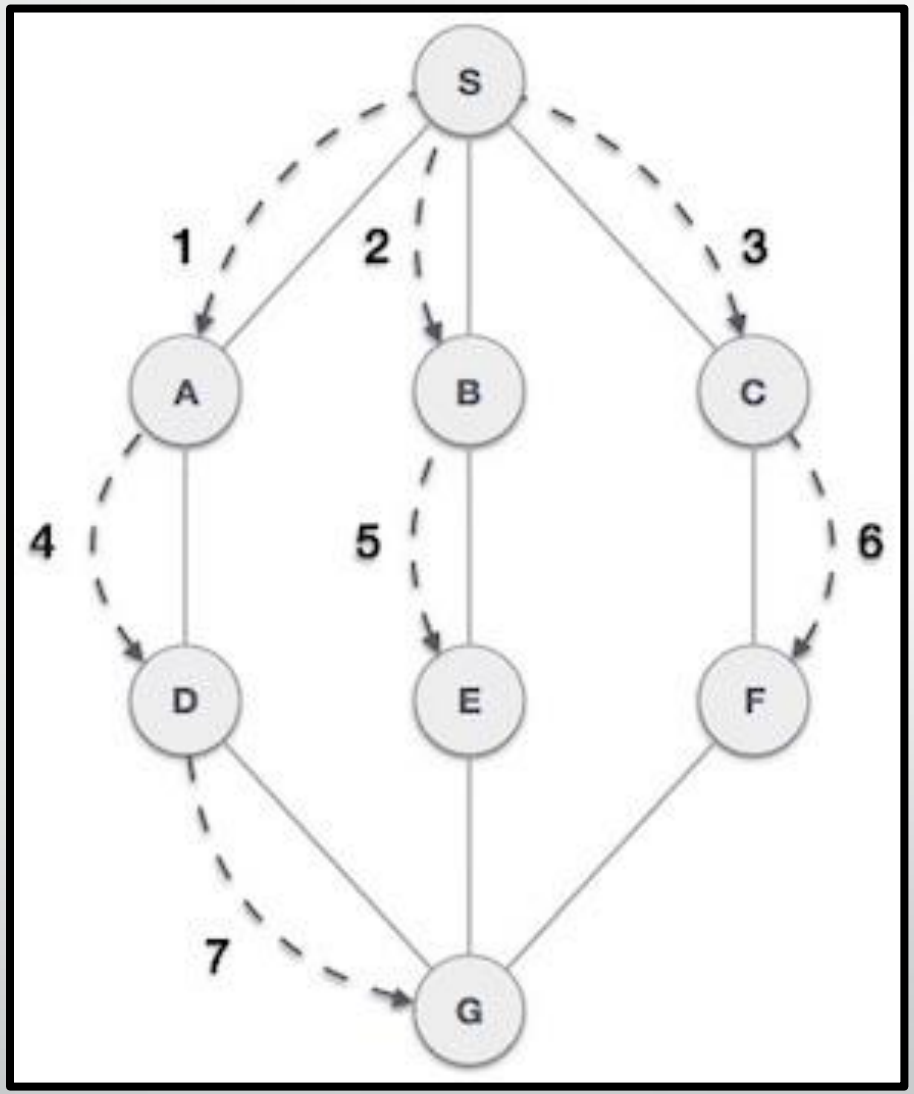
BFS traversal algorithm and path



(From Ahmed , 2020)

Figure 12

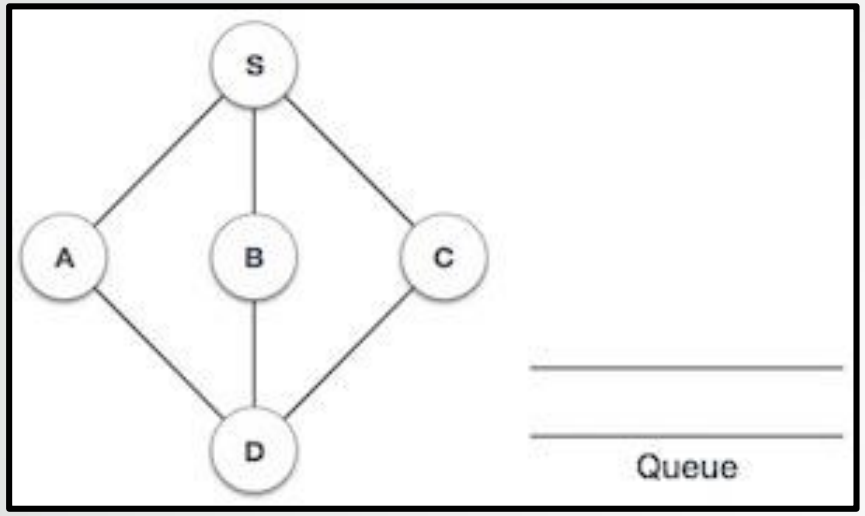
BFS traversal algorithm



(From *Data Structure - Breadth First Traversal - Tutorialspoint*, n.d.)

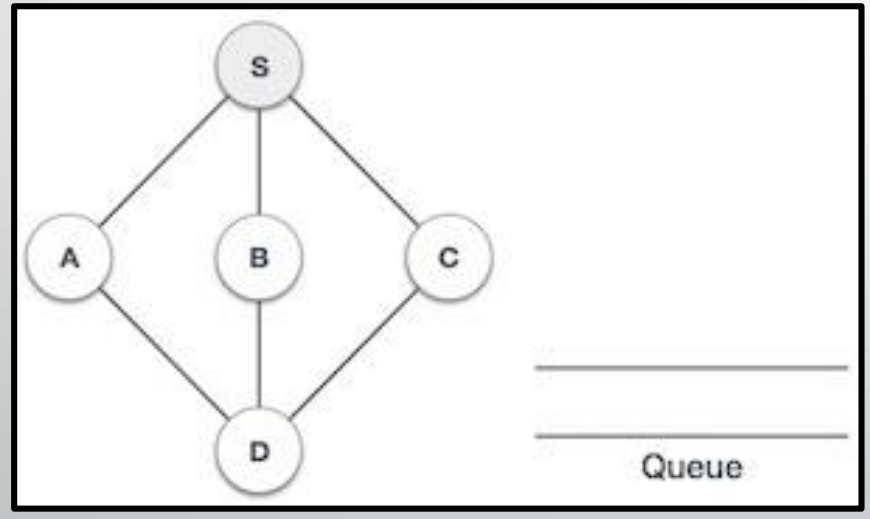
Figure 12 (a)

BFS traversal algorithm



Step 1: Begin

Initialize the queue



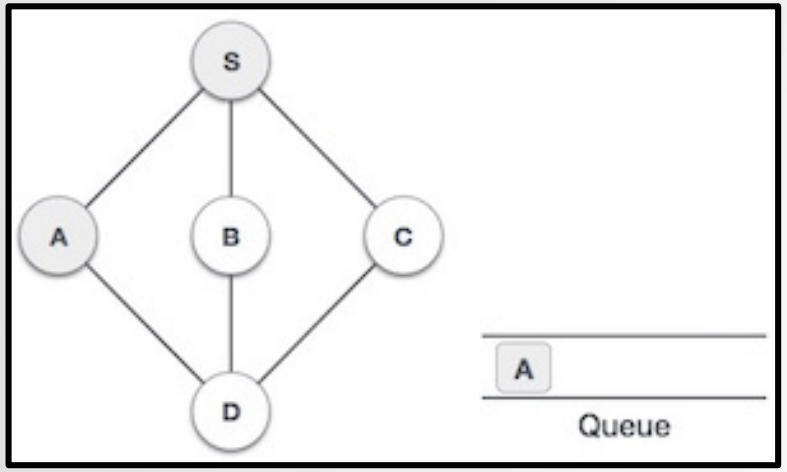
Next: Step 2

We start from visiting **S** (starting node), and mark it as visited.

(From *Data Structure - Breadth First Traversal - Tutorialspoint*, n.d.)

Figure 12 (b)

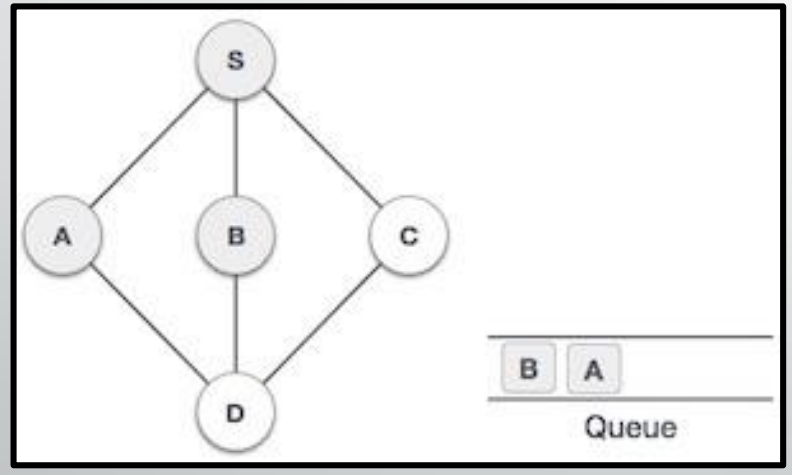
BFS traversal algorithm



Next: Step 3

We then see an unvisited adjacent node from **S**.

In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.



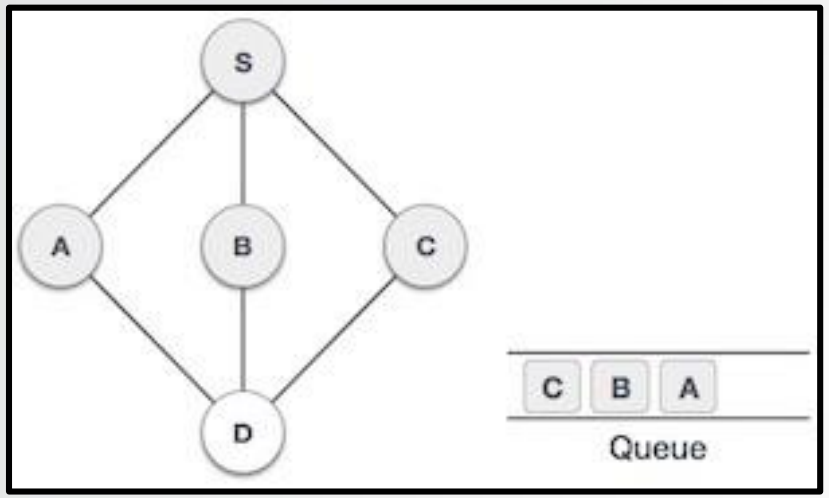
Next: Step 4

Next, the unvisited adjacent node from **S** is **B**.

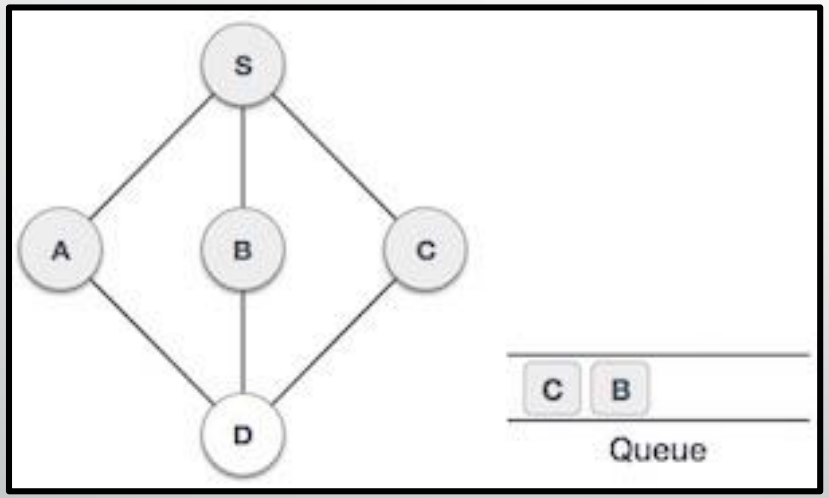
We mark it as visited and enqueue it.

(From *Data Structure - Breadth First Traversal - Tutorialspoint*, n.d.)

Figure 12 (c)
BFS traversal algorithm



Next: Step 5
Next, the unvisited adjacent node from **S** is **C**.
We mark it as visited and enqueue it.

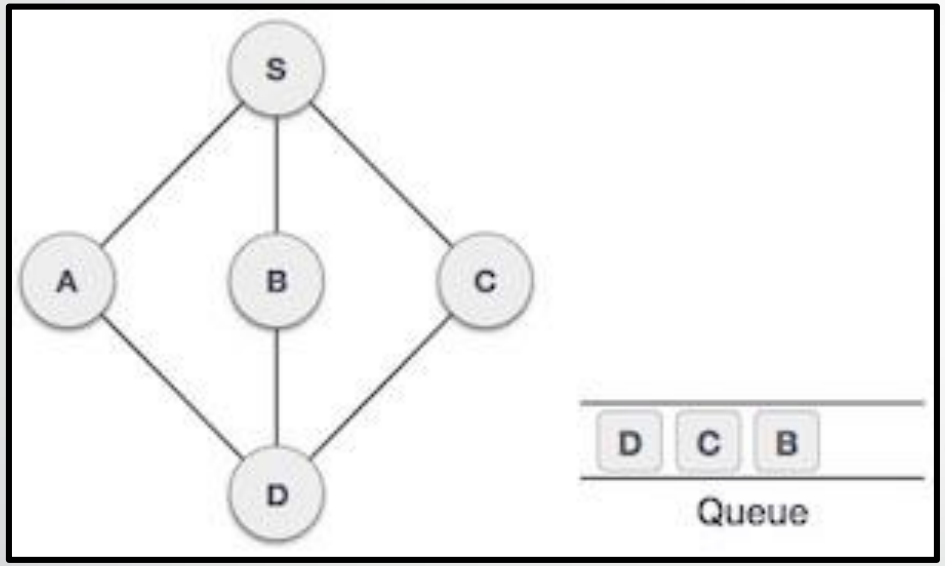


Next: Step 6
Now, **S** is left with no unvisited adjacent nodes.
So, we dequeue and find **A**.

(From *Data Structure - Breadth First Traversal - Tutorialspoint*, n.d.)

Figure 12 (d)

BFS traversal algorithm



Next: Step 7

From **A** we have **D** as unvisited adjacent node.

We mark it as visited and enqueue it.

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

(From *Data Structure - Breadth First Traversal - Tutorialspoint*, n.d.)

3.3 Breadth first search (BFS)

- The rules for the BFS algorithm are as follows (*Data Structure - Breadth First Traversal - Tutorialspoint*, n.d.):
- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.
- In application terms BFS is used in the following areas (Miladev95, 2023):
- Web Crawling: BFS is used by web crawlers to explore websites level by level, ensuring that all reachable pages are visited.
- Social Network Analysis: BFS can be used to analyze social networks, identifying relationships and connections between individuals.

Part 4

Shortest paths

4.1 Introduction

- In our previous parts we have used traversal algorithms to find the path to a vertex that is reachable. Both algorithms aim to find a short path to the target in ideal situations, though BFS is said to be more effective in achieving this.
- These situations are called ideal since they don't take into account some factors that may make a path appear to be short but in essence isn't. Let me explain by example.
- In networking we might consider the path(s) to a certain node say A. One path is physically shorter than the other; ideally this would be the path to use. However, there are still other factors to put in consideration; for example, the path may be physically shorter but it has less bandwidth available, which ultimately affects the throughput. Another path might be physically longer but the bandwidth allocation is higher meaning better throughput.

4.1 Introduction

- Consider the routes available to travel from point A to point B by road; the shortest distance isn't necessarily the ideal distance. It could be a distance of 50 kms, but half of it is full of potholes, unnecessary bumps, and poorly maintained off road sections. The longer route is 85 kms but it is well tarmacked, no potholes, and bumps in just one section; which one is the better route to use in terms of saving time and fuel cost? Obviously the longer route.
- This tells us that we need to use some parameter to determine which path to use to reach a certain point; the parameter we use is called weight.
- The weight assigned to an edges in a graph are based on some factor applied equally to all the edges; it could be distance between the edges, bandwidth, or some other factor. The term cost is also used in place of weight and serves the same purpose.
- Goodrich et al. (2018) describe a weighted graph as follows:"(it) is a graph that has a numeric (for example, integer) label $w(e)$ associated with each edge e, called the ***weight*** of edge e. For $e = (u,v)$, we let notation $w(u,v) = w(e)$."

4.1 Introduction

- Further, if we let G be a weighted graph, the ***length*** (or weight) of a path is the sum of the weights of the edges of P. That is, if $P = ((v0,v1), (v1,v2), \ldots, (vk-1,vk))$, then the length of P, denoted $w(P)$, is defined as

$$w(P) = \sum_{i=0}^{k-1} w(v_i, v_{i+1}).$$

- The ***distance*** from a vertex u to a vertex v in G, denoted $d(u,v)$, is the length of a minimum-length path (also called ***shortest path***) from u to v, if such a path exists. People often use the convention that $d(u,v) = \infty$ if there is no path at all from u to v in G.
- We describe an algorithm that is used to find the shortest path between vertices u and v. Due to time and space constraints the learner is encouraged to read on the Floyd Warshall Algorithm for shortest distances between nodes.

4.2 Dijkstra's Algorithm

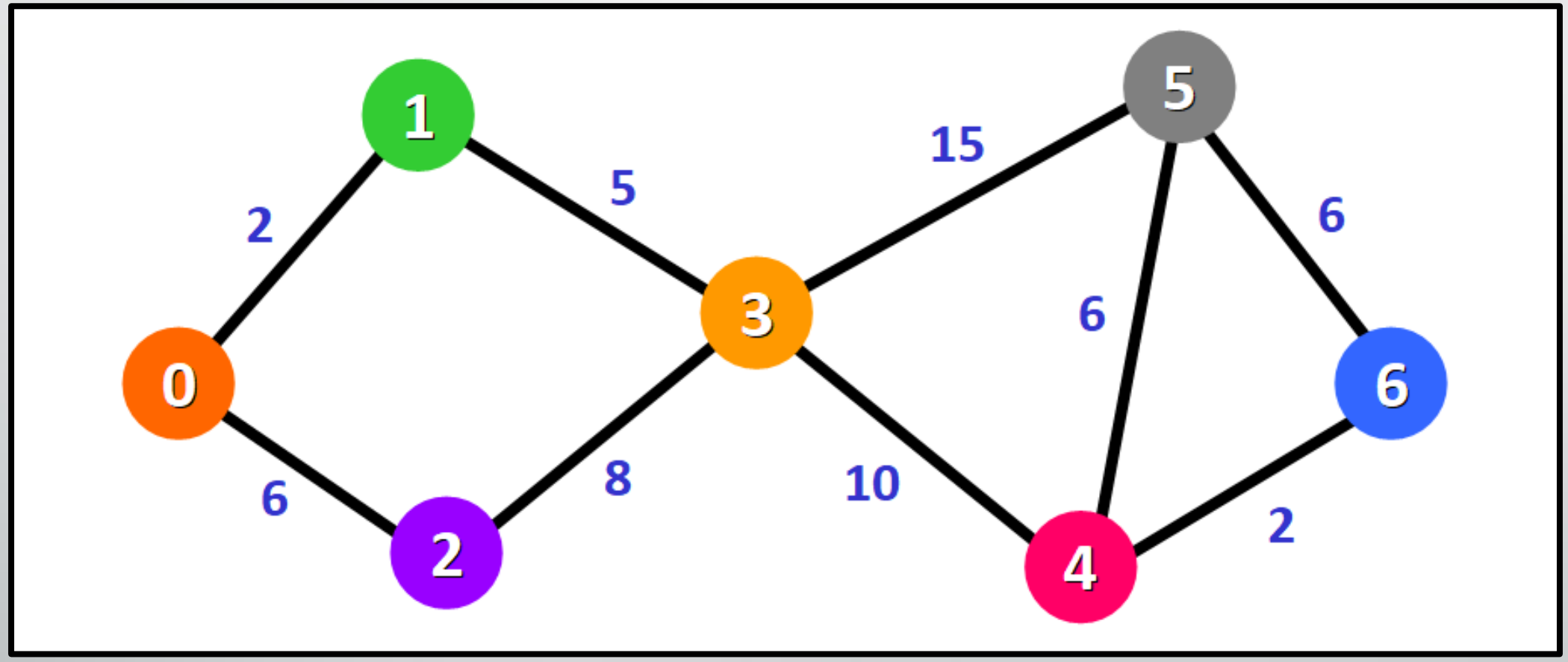
- With Dijkstra's Algorithm, you can find the shortest path between nodes in a graph. Particularly, you can **find the shortest path from a node (called the "source node") to all other nodes in the graph**, producing a shortest-path tree. (Navone, 2020)
- This algorithm is a famous one and is used in many areas like GPS of car systems to find shortest distance from one point to another. Its working (steps) can be found in many literature; however, let us use the description provided by Navone (2020): "
- Dijkstra's Algorithm basically starts at the node that you choose (the source node) and it analyzes the graph to find the shortest path between that node and all the other nodes in the graph.
- The algorithm keeps track of the currently known shortest distance from each node to the source node and it updates these values if it finds a shorter path.

4.2 Dijkstra's Algorithm

- Once the algorithm has found the shortest path between the source node and another node, that node is marked as "visited" and added to the path.
- The process continues until all the nodes in the graph have been added to the path. This way, we have a path that connects the source node to all other nodes following the shortest path possible to reach each node.
- The algorithm can only be used with graphs where the edges have positive weights; obviously, since we need to calculate a cumulative distance between vertices (nodes).”
- Let us demonstrate it’s working via an example.

Figure 13

Weighted graph



(From Navone, 2020)

4.2 Dijkstra's Algorithm

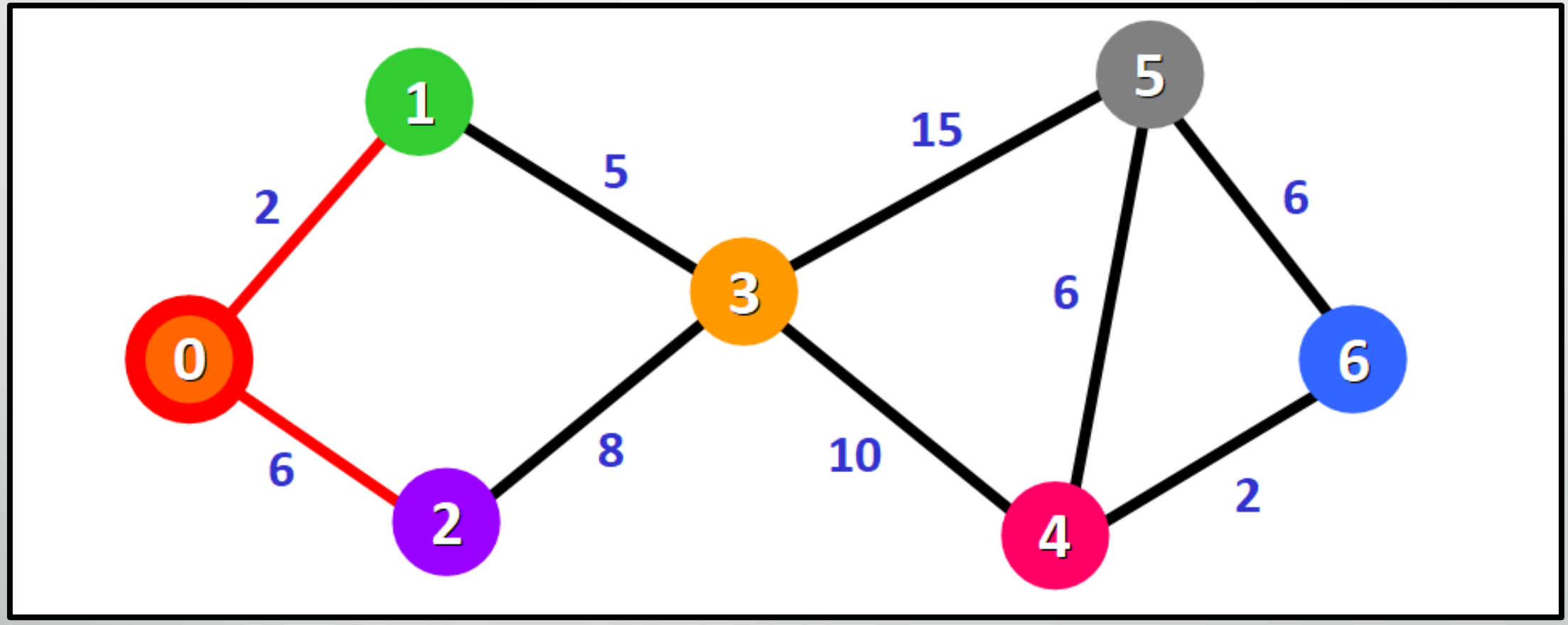
- Figure 13 illustrates a weighted graph. We wish to use Dijkstra's algorithm to find the shortest path from node 0 to all other nodes. The working out is provided by Navone (2020):
- We will have the shortest path from node 0 to node 1, from node 0 to node 2, from node 0 to node 3, and so on for every node in the graph.
- We denote unknown (uncomputed) distances between nodes as ∞, meaning we are yet to find the shortest path to the said nodes. We begin our calculation from node 0 itself as follows:
- Distance: 0: 0, 1: ∞, 2: ∞, 3: ∞, 4: ∞, 5: ∞, 6: ∞, since we know the distance from a node to itself is 0.
- Therefore we need to visit each node to calculate the shortest distance between it and node 0. We keep track of the unvisited nodes by placing them in a set:
- {0, 1, 2, 3, 4, 5, 6} …we note that the algorithm is complete only after we have visited all the nodes.

4.2 Dijkstra's Algorithm

- We examine the adjacent nodes to 0 (1 and 2) and add the weight of the edges to them. This modifies our visited nodes as follows:
- Distance: 0: 0, 1: ~~∞~~ 2, 2: ~~∞~~ 6, 3: ∞, 4: ∞, 5: ∞, 6: ∞, resulting in :
- Distance: 0: 0, 1: 2, 2: 6, 3: ∞, 4: ∞, 5: ∞, 6: ∞
- The unvisited node distances remain unknown at this point.
- The next step is to:
- Select the node that is closest to the source node based on the current known distances.
- Mark it as visited.
- Add it to the path
- If we check the list of distances, we can see that node 1 has the shortest distance to the source node (a distance of 2), so we add it to the path. We represent this with a red edge as shown in figure 14.

Figure 14

Marking distances using Dijkstra's algorithm



(From Navone, 2020)

4.2 Dijkstra's Algorithm

- We mark it with a red square in the list to represent that it has been "visited" and that we have found the shortest path to this node. (also figure 14).
- Thus 1 is officially visited in the list of unvisited nodes.
- Next we need to analyze the new adjacent nodes to find the shortest path to reach them. We will only analyze the nodes that are adjacent to the nodes that are already part of the shortest path (the path marked with red edges in figure 14).
- Node 3 and node 2 are both adjacent to nodes that are already in the path because they are directly connected to node 1 and node 0, respectively, as seen in figure 14. These are the nodes that we will analyse in the next step.
- Since we already have the distance from the source node to node 2 written down in our list, we don't need to update the distance this time. We only need to update the distance from the source node to the new adjacent node (node 3):
- Distance: 0: 0, 1: 2, 2: 6, 3: ~~∞~~ 7, 4: ∞, 5: ∞, 6: ∞ => 0, 1: 2, 2: 6, 3: 7, 4: ∞, 5: ∞, 6: ∞

4.2 Dijkstra's Algorithm

- We may be wondering how we came up with 7?
- For node 3: the total distance is 7 because we add the weights of the edges that form the path 0 -> 1 -> 3 (2 for the edge 0 -> 1 and 5 for the edge 1 -> 3).
- Using this same logic we proceed with the remaining nodes:
- We continue using a red circle to show a visited node; however, the most obvious path isn't necessarily the shortest; the further we are from the source node, the more the need to examine all possible routes and their weights (remember we are looking for the shortest distance/weight/cost from source node to the destination).
- Let us consider the path to node 6: there are quite a few alternatives:

4.2 Dijkstra's Algorithm

- 0 →1, 1 →3, 3 →5, 5 →6 ……weight (2+5+15+6) = 28
- 0 →1, 1 →3, 3 →4, 4 →6 ……weight (2+5+10+2) = 19
- 0 →1, 1 →3, 3 →4, 4 →5, 5 →6 ……weight (2+5+10+6+6) = 29
- 0 →2, 2 →3, 3 →5, 5 →6 ……weight (6+8+15+6) = 35
- 0 →2, 2 →3, 3 →4, 4 →5, 5 → 6 ……weight (6+8+10+6+6) = 36
- 0 →2, 2 →3, 3 →4, 4 →6 ……weight (6+8+10+2) = 26
- As can be seen all weights and possible routes have to be examined to determine the shortest path; in our case the shortest path to 6 from node 0 is :
- 0 →1, 1 →3, 3 →4, 4 →6 ……weight (2+5+10+2) = 19
- Compute the remaining distances as an exercise. It should not be difficult.

Part 5

Minimum spanning trees

5.1 Introduction

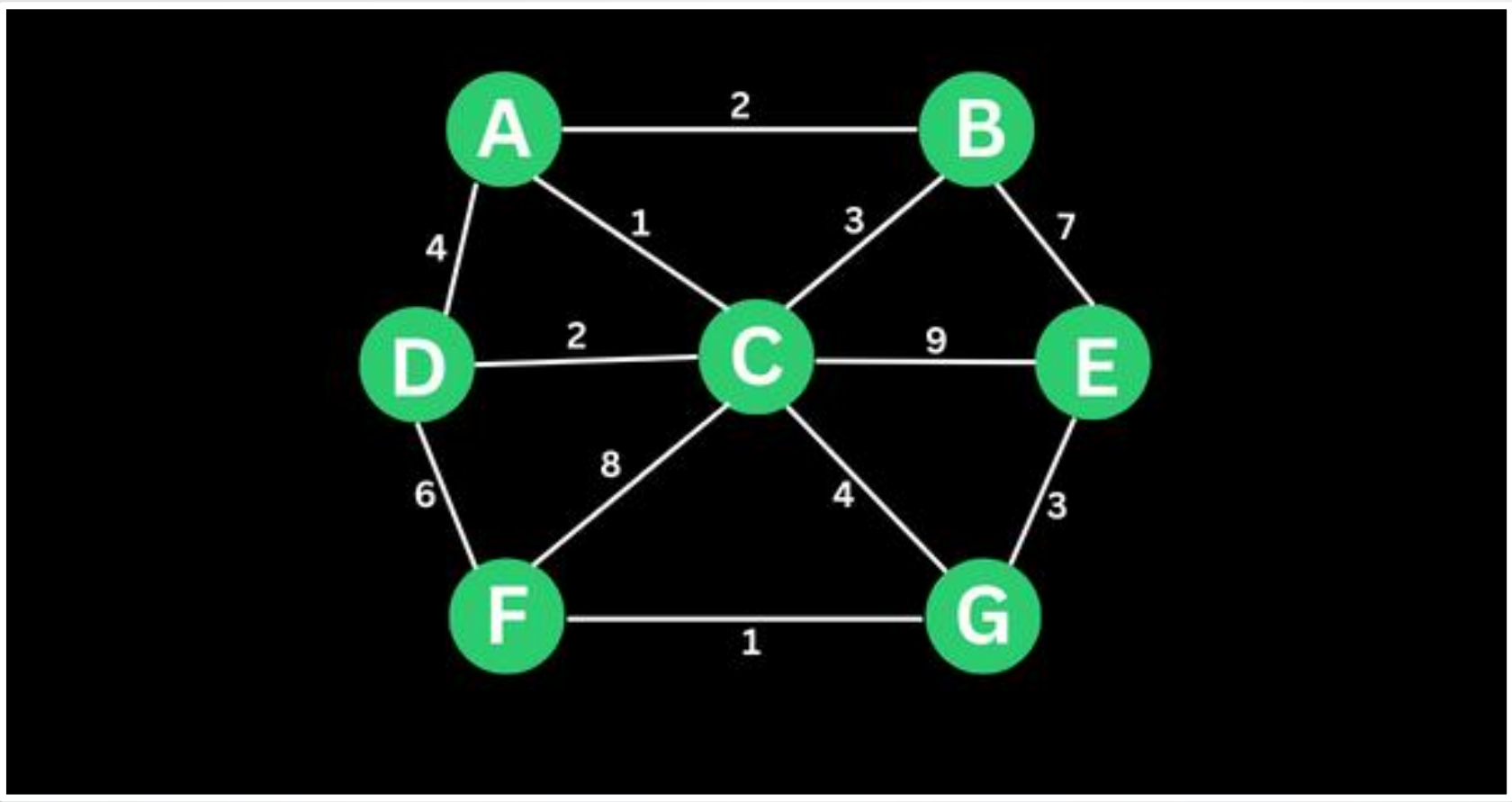
- In the last part we discussed algorithms that look for the shortest distance from a source node u to all other nodes in the graph.
- There is another group of algorithms similar to these; the spanning trees. A spanning tree is defined as (*Design and Analysis - Spanning Tree*, n.d.) "a subset of an undirected Graph that has all the vertices connected by minimum number of edges. If all the vertices are connected in a graph, then there exists at least one spanning tree. In a graph, there may exist more than one spanning tree. The properties of a spanning tree are:
- A spanning tree does not have any cycle.
- Any vertex can be reached from any other vertex.
- A **Minimum Spanning Tree (MST)** is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight (and without forming a cycle at any point)."
- MST algorithms are greedy algorithms, meaning they pick the best option available at the time (like what greedy people do). In this part we briefly describe Prim's algorithm to show how to create an MST; there are other MST algorithms and their goal is the same, though the approach may be slightly different.

5.2 Prim's algorithm

- By the properties of an MST, "if you're finding the MST of a graph with Prim's algorithm, there must be no cycle. That is, if A links to B and B links to C, C cannot link to A again because that would make a cycle." (Kolade, 2023).
- The steps to follow in implementing Prim's algorithm are as follows (Kolade, 2023):"
- all the vertices of the graph must be included
- the vertex with the minimum weight must be selected first. You'll also hear some people refer to that weight as distance, but let's keep calling it weight.
- all the vertices must be connected
- there must be no cycle"
- Let us find the MST for the graph in figure 15 as an example.

Figure 15

Graph



(From Kolade, 2023)

5.2 Prim's algorithm

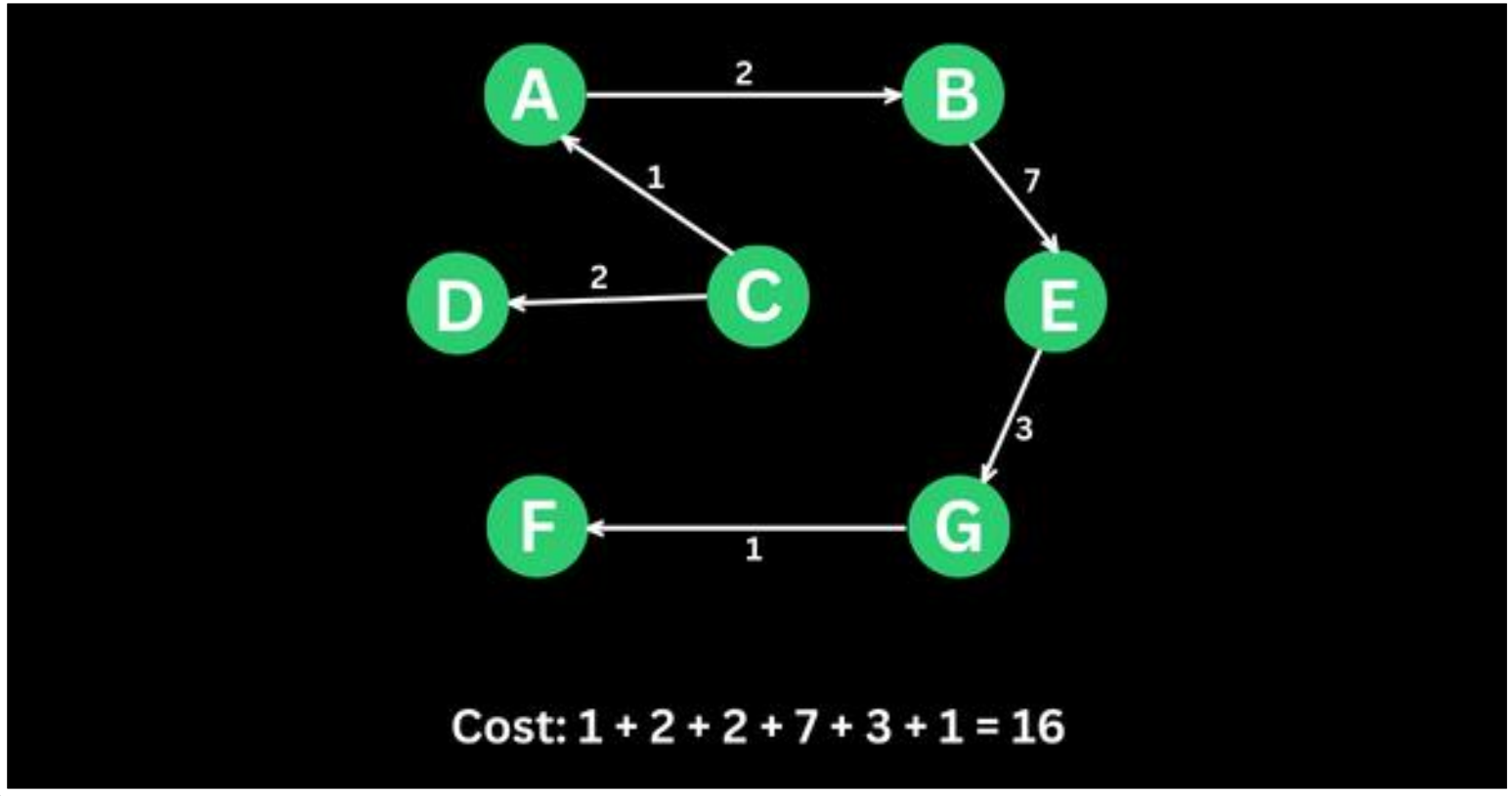
- In determining the MST, we first determine a starting point. For this demonstration we start from D. Kolade (2023) explains how to come up with the MST:"
- The next minimum weight connected to D is 2 – the line between D and C. So, we chose it.
- Looking at vertex C, the next minimum weight to it is 1 – the line between C and A. So, I chose it as the next one.
- Looking at A, lines 2 and 4 are connected to it. We cannot choose 4 because it's bigger than 2 and it'll lead us back to the starting point D. So, we have to choose 2 – the line connecting vertices A and B.
- Looking at B, line 3 connects it to C and line 7 connects it to E. We cannot choose line 3 because that will form a cycle between C, A, and B. We also should think twice before choosing line 7 because it's a big number. There's a line 4 connecting C to G, so, I chose it
- On the vertex G, there's a connection to F with line 1 and line 3 to E, so I'll choose the minimum weight which is 1

5.2 Prim's algorithm

- (Finally)
- At this point, E is the only vertex not connected yet. It's possible to connect it because it won't form a cycle at any point. So, I connected it."
- Based on this we can now show the MST for this graph in figure 16.
- Creating the MST using Prim's algorithm is about ensuring that each vertex is reached using the shortest path, and without using a cycle anywhere. Some uses of Prim's algorithm are (Kolade, 2023):
- designing transportation networks
- building phylogenetic trees in bioinformatics
- segmenting images based on color and pixel intensity
- grouping similar objects together in clustering algorithms

Figure 16

MST for figure 15



(From Kolade, 2023)

Summary

- A ***graph*** is a way of representing relationships that exist between pairs of objects. That is, a graph is a set of objects, called vertices (or nodes), together with a collection of pairwise connections between them, called edges.
- The ***edge list*** structure is possibly the simplest, though not the most efficient, representation of a graph *G*. All vertex objects are stored in an unordered list *V*, and all edge objects are stored in an unordered list *E.*
- The ***adjacency list*** structure groups the edges of a graph by storing them in smaller, secondary containers that are associated with each individual vertex.
- Two popular traversal algorithms are depth first search (DFS) and breadth first search (BFS).
- To find shortest distance to nodes from a source node we use algorithms with the same name; the most popular one is Dijkstra's algorithm. Floyd Warshall Algorithm is another example.
- Minimum spanning tree (MST) is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight (and without forming a cycle at any point). An MST algorithm such as Prim's algorithm creates the MST. Other algorithms that do this include Kruskal's Minimum Spanning Tree (MST) Algorithm.

References

- Ahmed , A. (2020, March 26). *Breadth-First Search - A BFS Graph Traversal Guide with 3 Leetcode Examples*. FreeCodeCamp.org. https://www.freecodecamp.org/news/breadth-first-search-a-bfs-graph-traversal-guide-with-3-leetcodeexamples/
- AlgoDaily. (n.d.). *AlgoDaily - Daily coding interview questions. Full programming interview prep course and software career coaching.* Algodaily.com. Retrieved November 12, 2023, from https://algodaily.com/lessons/implementing-graphs-edge-list-adjacency-list-adjacency-matrix/python
- *Data Structure - Breadth First Traversal - Tutorialspoint*. (n.d.). Www.tutorialspoint.com. Retrieved November 12, 2023, from https://www.tutorialspoint.com/data_structures_algorithms/breadth_first_traversal.htm
- *Data Structure - Depth First Traversal - Tutorialspoint*. (n.d.). Www.tutorialspoint.com. https://www.tutorialspoint.com/data_structures_algorithms/depth_first_traversal.htm

References

- *Design and Analysis - Spanning Tree*. (n.d.). Www.tutorialspoint.com. Retrieved November 13, 2023, from https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_spanning_tree.htm#:~:text=A%20Minimum%20Spanning%20Tree%20(MST
- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2018). *Data structures and algorithms in Python*. Wiley.
- Kolade, C. (2023, February 14). *Prim's Algorithm – Explained with a Pseudocode Example*. FreeCodeCamp.org. https://www.freecodecamp.org/news/prims-algorithm-explained-with-pseudocode/
- Lambert, K. (2014). *Fundamentals of Python*. Cengage Learning Ptr.

References

- Miladev95. (2023, August 12). *Breadth-First Search (BFS) Graph search algorithm in PHP*. Medium. https://medium.com/@miladev95/breadth-first-search-bfs-graph-search-algorithm-in-php-dbb5e368e3b6
- Navone, E. C. (2020, September 28). *Dijkstra's Shortest Path Algorithm - A Detailed and Visual Introduction*. FreeCodeCamp.org. https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/#:~:text=Dijkstra