Own Topic - Graph Theory Algorithms

Isak Oswald
s225375329

April 12, 2025

1 Module learning goals

1. Explore and implement different methods of graph representation
2. Apply graph traversal algorithms like DFS and BFS to explore graphs.
3. Apply Dijkstra's and Bellman ford algorithms to computed weighted graphs with positive and negative weights
4. Apply the knowledge to practise problems

Requirements

1.1 Request for feedback

1.2 Response to feedback

N/A

2 Graph Representation

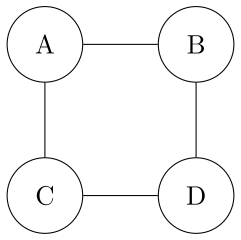
Introduction

Graph theory is a fundamental topic used to model relationships between different objects. These objects could represent many things everywhere if you look out for them, areas like networking, transportation maps, algorithms, network flow, and even your friends list on Facebook are all graph theory-related. To work with these graphs effectively, we must choose an appropriate method to represent them in our computer memory, two common ways of doing this are through the adjacency list and adjacency matrix.

Adjacency list

An adjacency list representation for a graph links each vertex with its finite neighboring vertices or edges. The main goal of the adjacency list data structure is to report a list of neighbors given vertices v. There are many variations of this basic idea, one being **Guido van Rossum**'s idea where he uses a has table to associate each vertex in a graph with an **array** of adjacent vertices. **Cormen et al** suggests that all of the vertices are represented by an index number, this means that we use an array-indexed approach with the array index being the vertex number [1]. Now, if we break this down further, for each array entry (so at each index) there is a pointer to a *singly linked list* which contains all of the neighboring vertices of that vertex. There is one final approach that I would like to cover which is the object-oriented approach suggested by **Goodrich and Tamassia** which is a little more memory than other methods where we list them directly. This method has an object of the vertex class that points to a collection object that holds the neighboring edges objects. As we said before, this uses more memory, however, is very flexible in terms of adding new information about edges as we can simply add and edge the object [1]. All of these methods take a slightly different approach to how exactly we represent the vertices and edges, however, all achieve the same goal which is storing them somewhere in memory. For simplicity and demonstration purposes, I will look at a method that uses a more direct approach to store the vertices and edges. Let's have a look at a quick example of how these are represented so you can get a visual idea that will assist in reading and creating them yourself.

Consider the following graph:



Now as we have studied graph theory previously, we can quickly identify a few things about this graph, specifically concerning each vertex and its neighboring vertexes. We know that node A is connected to both node B and C with an edge, therefor they are adjacent (or neighbors). This means that node B's neighbors are node A and D, we can then do the same for vertices D and C and you will now have a list of neighbors based on each vertex V. Now, let's put what we just learned about this representation method into something more readable.

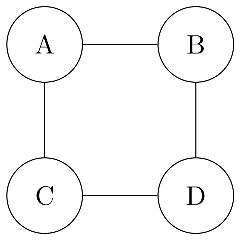
Vertex	Neighbours
A	$B,\ C$
B	$A,\ D$
C	$A,\ D$
D	$B,\ C$

As you can see, as we just identified, vertex v_0 now has two neighbors, v_1 and v_2. This however is still not directly in the computer memory, to convert this into something that can be stored we can use a hash map (or a dictionary in Python) where the key is v_0 and the pair is v_1 and v_2. Now, we have a way to access these relationships, if we want to find out what adjacent vertex's node x has, we just search for the key in the hash-map and read its values, which in this case will be two other vertices.

Adjacency matrix

In graph theory, an adjacency matrix is a square matrix used to represent whether pairs of vertices and adjacent within a graph. This is very similar to the timetable magnets you might have had when you were first learning multiplication, however instead of finding the intersection of your two numbers and then getting the answer, there will either be a 1 or a 0 to represent if vertex v_0 is adjacent to vertex v_1. For a **simple** graph with the vertex set $U = \{u_1, u_2, u_3, \ldots, u_n\}$, the adjacency matrix is represented with a square n by n matrix denoted by A such that element $A_{i,j}$ holds a value of one if there is a edge connecting vertex u_1 and u_j. As you probably guessed, if there is no edge connecting these vertices $A_{i,j}$ will hold a value of 0 [2].

In my opinion, this is the most intuitive representation, let's provide a sample with the same graph as we did previously to see how they differ.

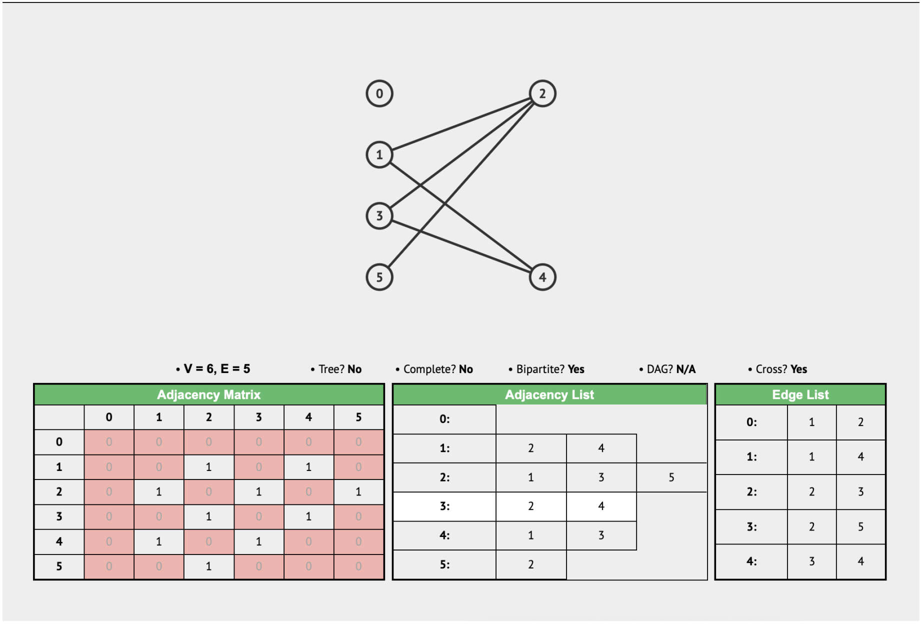


The Adjacency matrix representation of this would be:

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

Now how do you read this? Well, remember the definition we constructed before, let's see if nodes A and B are connected. This means we are looking for the intersection in the matrix, specifically A_{A_B} as our i and j are the intersection we are looking for. It seems that there is a value of one here, great! We know that they are connected by an edge.

Have a look below where you can compare these methods.



• **V = 6, E = 5** • Tree? **No** • Complete? **No** • Bipartite? **Yes** • DAG? **N/A** • Cross? **Yes**

Adjacency Matrix

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	1	0	1	0
2	0	1	0	1	0	1
3	0	0	1	0	1	0
4	0	1	0	1	0	0
5	0	0	1	0	0	0

Adjacency List

0:			
1:	2	4	
2:	1	3	5
3:	2	4	
4:	1	3	
5:	2		

Edge List

0:	1	2
1:	1	4
2:	2	3
3:	2	5
4:	3	4

Figure 1: Comparison between Adj list and matrix [3]

Advantages and disadvantages of the representations

One thing that I wondered when researching these methods was "Does it matter which one I use?" and the answer is yes. Lucky for me, I have been studying programming a did an advanced module in Big O notation which is very important, especially when the input size is large. The adjacency matrix is very fast to look up if an edge exists, specifically, it is $O(1)$ as there is no need to iterate, we just need to know i and j and search. The agency matrix is also very good for graphs that are dense (graphs with a lot of edges) as we can easily look up the existence of an edge. However, there are some issues with this method, the main one being it often wastes a lot of memory, especially if the graph is sparse as most of the entries will be zero. On the other, the adjacency list does what the matrix can't, its main strength is flexibility (it's very easy to append new info like edge weights). Considering this, let's go back to the question "Which should I use?", well, you should use the adjacency matrix if your graph is dense, and you often check if there is an edge between A and B. However, you should consider an adjacency list if your graph is sparse and you are often checking what neighbors node A has. Below is a table that I have created with some additional information.

Comparison of Graph Representations

Aspect	Adjacency List	Adjacency Matrix
Space Complexity	Efficient for sparse graphs: $O(V+E)$	Requires $O(V^2)$ space, even if the graph is sparse
Edge Lookup Time	Slower: $O(k)$, where k is the number of adjacent vertices	Fast: $O(1)$ time to check if an edge exists
Iteration over neighbours	Fast: only adjacent vertices are stored	Slower: must scan entire row to find neighbours
Ease of implementation	Slightly more complex (linked lists or vectors)	Simpler (2D array)
Best use case	Sparse graphs with fewer edges	Dense graphs or when frequent edge lookups are required
Dynamic Graph Changes	Easier to add/remove edges and vertices	Harder to resize (fixed size 2D array)

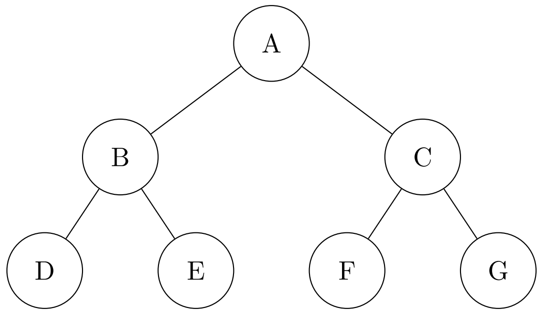
Table 1: Comparison of Adjacency List and Adjacency Matrix

Some beginner algorithms

Now that we know how to create and extract different representations of graphs from the image itself, as well as how to implement them in code we need to find a way to explore these graphs. It would be tedious and computationally expensive to draw and do this manually on something like pen and paper however lucky for us, there have been some algorithms created where we can use our newfound ways of representing graphs and then traversing them. Some of the algorithms we will initially look at are Breadth First Search (BFS) and Depth First Search (DFS). Do not mistake DFS and BFS for being the same, as they are both methods to traverse/visit the nodes in a tree *or* graph (they have the same goal) the difference is in the way and the order they explore each node.

3 Example graph

As DFS and BFS are similar algorithms (they both explore graphs) I will be referring to a single graph when talking about both concepts. This will also assist in my LO 4. One thing to note is that DFS and DFS does **not** have to be exclusively used on a tree, it can also be performed on a cyclic graph. However, this is where the importance comes down to updating the nodes after you visit them. The graph can be found below.



4 DFS

Think of a graph as a pool and you aim to find an object at the bottom, DFS is like scanning from the shallow end to the deep end (without searching all of the shallow end first) and then going back to the shallow end a repeating the process. DFS starts from the first node, often the root of a tree, and goes as deep as it can until it traverses to the leaf node before retracing its steps and then traversing down another path, this is very similar to scanning a family tree vertically. Let's formalise this high-level idea into a proof.

Let $G = (V, E)$ be a graph with vertices V and edges E, and let $s \in V$ be the starting vertex. Let V_{visited} represent the set of vertices that have been visited during the DFS traversal.

Now, we will assume that DFS will eventually visit all vertices in the graph that **reachable** from the starting vertex s. This essentially means that if vertex u is reachable from s, it will eventually be visited by DFS. This can be easily done by hand if the graph is small enough.

Now for our base case, since the DFS algorithm starts at vertex s, that means that, $s \in V_{\text{visited}}$. If there were no vertices in the graph of-course it would terminate as there is nothing to start searching from, or nothing to search to.

At each step/iteration of the algorithm, DFS explores the adjacent vertices of the current vertex v:

- For each adjacent vertex u such that $u \notin V_{\text{visited}}$, DFS recursively visits u.

- This process continues until there are no more unvisited adjacent vertices, at which point DFS backtracks to the most recently visited vertex.

Since the DFS algorithm terminates when there are no more unvisited vertices in the graph. This happens because:

- Every vertex $v \in V$ that is reachable from s will eventually be encountered during the traversal as a result of visiting its adjacent vertices.
- If any vertex is left unvisited, it must be **unreachable** from s, and therefore the DFS will not visit it.

How to apply the algorithm to a problem

Now we know how the BFS algorithm is working, let's apply it by hand to achieve LO 2. If we consider the graph above, we will start at node A (which is our s). We then check this off at add it to our set of $V_{Visited}$ nodes, this means that so far node A is our only element of this set. The algorithm now for each adjacent vertex of A we follow down the path, this means that we will reach B as we travel along the AB edge. We now add this B to our $V_{Visited}$ set. We then continue following the path along the AB edge and add D to our $V_{Visited}$ set. Since D has a degree of one, there are no further "deeper" vertices which means the only way to go is to reverse our steps, we follow the BD edge back up to node B. Now we have two options, follow the BD edge or the BE edge, since vertex $D \in V_{Visited}$ we will follow the BE edge. We reach vertex E as add that to our $V_{Visited}$ set, since E has a degree of one, the only option we have is to follow the edged back up to node A.

Now we are in the same position we were when beginning the algorithm and have two choices, follow the AB edge or the AC edge, since $B \in V_{Visited}$ it must also mean that its children are $\in V_{Visited}$. Now we just follow the same process for the right side of the graph starting by following the edge AC, adding it to the $V_{Visited}$ set, and following deeper to the F node. Again, since F has a degree of one we must retrace our steps, which results us in finally adding G to our set. Now since both C and $B \in V_{Visited}$, it means we have successfully traversed the whole graph (since A has a degree of two). Therefore, the order of traversal is $A \to B \to D \to E \to C \to F \to G$

This explanation is very, very descriptive highlighting every step that does not need to explicitly be done when conducting practice problems, I have done this to explicitly demonstrate progress towards LO 1. The reason we need to understand this process is that it's not us who's going to be conducting it, the computer is.

Pseudo-code

Now as we have explored how to use it and how it works at a lower level, the computer does almost the same thing, have a look below.

```
Algorithm DFS(Graph G, Vertex s):
    Input: Graph G = (V, E), starting vertex s
    Output: Set of visited vertices V_visited

    1. Initialise empty set V_visited
    2. Define recursive function DFS-Visit(vertex v):
        a. Add v to V_visited
        b. For each neighbour u of v in G:
            i. If u not in V_visited:
                - Call DFS-Visit(u)

    3. Call DFS-Visit(s)
    4. Return V_visited
```

Now after learning this, you should be comfortable with applying these algorithms when you are given a graph. I will be doing this in my self-assessment.

5 BFS

Initially, when I was learning, I thought that BFS and DFS were two completely unrelated topics however, it turns out that they are very closely related, and *almost* function in the same way. BFS similarly starts from a selected node (usually called the root) however explores the graph in a layer-by-layer approach. This means that BFS visits all of the immediate neighbors before exploring the children of the node of the root. If we look back at our pool analogy, this is like starting at the shallow end, searching that part, then going a little deeper and restarting the process. One thing that I would like to address is that there are multiple answers when asked to "find the reversal" as you could either explore the nodes in different orders, these ways of traversal are called pre-order, in-order, and post-order. Since my LOs are concerned with how to apply the algorithms not how we visit the nodes and in what order I will not go in-depth on this.

Lets again let $G = (V, E)$ which represents a graph with V vertices and E edges, we can also say that $s \in V$ is the starting vertex. We will let $V_{Visited}$ denote the set of vertices that have been visited during the traversal of the algorithm. BFS commonly uses a queue data structure (FIFO) to keep track of the order of the exploration.

Now, we will assume that the BFS algorithm will eventually visit all of the vertices in the graph that are **reachable** from our starting vertex s. We know that this will happen because BFS adds every unvisited adjacent node to the queue.

Since the BFS starts at a vertex s we know that $s \in V_{Visited}$ as if the graph was empty (V is a null set) there would be no nodes to explore.

After each step of the algorithm, we dequeue a vertex v from the front of the queue and explore all of its children (if it's a tree). We then make u as visited and remove u from the queue. Because the algorithm sifts through the nodes in the order they were discovered, all nodes at distance k are visited before any node at distance $k+1$ which ensures that when a node is visited we have taken the shortest path to reach it in terms of edges.

How to apply the algorithm to a problem

Let's look at the same example we did for BFS. We have our starting node s which is vertex A. We know that A has two children B and C so these are enqueued. We dequeue B and then process its children which results in our new queue C, D, E. Now we dequeue C and process its children, therefore we have the queue D, E, F, G. We dequeue D and check its children, since it has none we continue. Since F, G and E also have no children we are finished, the queue is empty. This means that one solution to this problem would be $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow G$.

Pseudo-code

Now that we understand how BFS works on a conceptual level, let's see how this would look in terms of computer operations. The computer processes the queue and visits the nodes in the same systematic way as we discussed earlier.

```
Algorithm BFS(Graph G, Vertex s):
    Input: Graph G = (V, E), starting vertex s
    Output: Set of visited vertices V_visited

    1. Initialise empty set V_visited
    2. Initialise empty queue Q
    3. Add s to V_visited
    4. Enqueue s into Q

    5. While Q is not empty:
        a. Dequeue vertex v from Q
        b. For each neighbour u of v in G:
            i. If u not in V_visited:
                - Add u to V_visited
                - Enqueue u into Q
```

```
6. Return V_visited
```

As you can see we are using a queue, the role of the queue is to hold the vertices at that current layer. When we initialize the queue Q to hold the root node v we append all of the neighbors of v and then remove it from the queue, this is exactly how it searches layer by layer.

6 How do we know what algorithm to use?

There is no definitive answer to this question and it depends upon the problem itself. Both BFS and DPS are powerful graph traversal algorithms, however, they both serve slightly different purposes concerning the nature of the problem and the structure of the graph.

We would use BFS we we need to find the shortest path in terms of edges in an unweighted graph, this is because since we are searching layer by layer, we will always find vertex v's path with the least amount of edges from the starting vertex s. Another case we may use BFS is when we are dealing with a very wide and shallow graph and want to explore all of the options before searching for options further away. When considering graph problems we have to think of what the vertexes and edges could represent, if vertex s was our house and the edges represented roads, it would make sense to use BFS to find out which shopping mall you would go to (considering you didn't have a favorite and they all stocked the same items) as it makes logical sense to go to the one the closest to your house before even considering ones further away. BFS guarantees the shortest path in an unweighted graph G however, it can consume a lot of memory if the graph is wide because of storing the unsorted vertices in a queue. Imagine a graph was only two levels and the root node had 1000000 children, that's a loss of entries into the queue.

This doesn't mean DFS is useless, however, as it is useful when we want to explore all possible paths to traverse the entire graph. This makes DFS great for problems like cycle detection, topological sorting, or even solving puzzles and mazes. I'm sure you are familiar with the game Sudoku, imagine you have never seen it and played it for the first time, one logical approach would be to tap on a square and then enter numbers one through nine and continue this process for all available squares and if you run into a conflict (you most likely will) and then backtrack and re-try a new number. This is essentially "committing" to one path and the backtracking if it does not work which is exactly what BFS is about. However, as you know if you have played the game before, this is not very efficient and can get you stuck going into a deeper path than you need to, even when a closer solution exists (maybe the solution to that square was 3 but you had to try all values). DFS does not guarantee the shortest path as you are simply "brute-forcing" the graph and just observing if a path exists.

7 Dijkstra and Bellman Ford contributions

Reflecting on our knowledge so far, we have only used an unweighted graph, meaning that the edge connecting vertices v_1 and v_2 doesn't represent anything except for a connection. Let's go back to our demonstration using the shopping malls, you could use DFS to find the shopping mall the closest, however, roads have traffic, so choosing a route based only on the number of edges might not be the fastest anymore, you may take two backroads instead of the super densely populated highway which is faster. Suddenly, DFS and BFS are out the window and we need to find a new way to not only traverse the graph but also record the weight (in this case the time to travel) of each edge to find the true shortest path of a graph - or the quickest way to the shopping mall, avoiding edges with large weights. This is where by far the most interesting algorithm comes into play, Dijkstra's algorithm, and soon later Bellman Fords's algorithm which will address a key flaw in Dijkstra's algorithm.

Dijkstra

As we discussed, Dijkstra's algorithm is designed to compute the **shortest path** from a single source vertex s to all other vertices in a weighted graph with **no negative edge weights**. The algorithm works by systematically exploring the graph and choosing the vertex with the *currently* known smallest distance from the source node. This means that the shortest distance can update while the algorithm is running, it's like you found a new shortest path to the store and you just scratch your previous route to the store because what's the point in keeping the longer one?

Dijkstra's algorithm works by picking a source/root node to begin its traversal from, this vertex is denoted as s and is set with a distance of 0 (as we are already there). We don't currently know how 'heavy' the path will be to the other vertices yet to be explored, so they are assigned a value of ∞. We need a way to keep track of which vertices have been explored as well as a way of storing the graph which can be done through an adjacency list **or** adjacency matrix where each edge also stores its weight. [10]

As we previously explored in DFS, we used a queue to keep track of the nodes we needed to search, similarly, here we use a priority queue which is fundamentally the same but has a priority assigned to each entry. With this data structure, we can retrieve the vertex with the smallest tentative distance. Since we have only explored the source node s, it is our only current entry into the queue.

Now the algorithm will extract the vertex u with the smallest predominate distance from the queue, for each neighbor v of u we calculate the alternative path distance and if and only if this distance is shorter than what is currently recorded, we update v's distance.

Once all nodes have been visited by the algorithm, you should have the shortest path from any v to s that is of course reachable from s. Based upon my research we are *usually* not looking for every single vertex v shortest path from s however just the single shortest path from s to anyone v [11]. This can be achieved with an additional check to see if the algorithm has found the shortest path from s to v and then terminates early. One thing about Dijkstra's algorithm is that it actually **not** computes the shortest path in the sense you may think, it only gives us the shortest edge weight to get to node v. This is because the algorithm follows a two-step process, updating all neighboring nodes with our new best estimate, and then traveling to the node with the shortest path, this means our final destination will be the sum of all of the previous edge weights of nodes on the way to destination v. To solve this, we can add another step, this step would be included after updating the node's shortest path without our best estimate, we then record the location that it travels from to the new node ()I have shown this in my pseudo-code.

Below, you can see some pseudo-code to get an idea of how it would work.

```
function Dijkstra(Graph, source):
    // Initialize distances and previous nodes
    for each vertex v in Graph:
        dist[v] = infinity
        prev[v] = undefined
    dist[source] = 0

    // Create a priority queue Q and add all vertices
    Q = priority queue containing all vertices in Graph, ordered by dist[]

    while Q is not empty:
        u = vertex in Q with smallest dist[u]
        remove u from Q

        for each neighbor v of u:
            alt = dist[u] + weight(u, v)
            if alt < dist[v]:
                dist[v] = alt
                prev[v] = u
                decrease priority of v in Q to dist[v]

    return dist[], prev[]
```

One thing that I love about coding is that there are always many solutions to a single problem, my pseudo code is not a defined way you have to structure your code, it is only one way (storing data in a priority queue).

7.1 The problem with negative edges

As we explored DFS, BFS, and Dikstra's algorithm we know how to apply algorithms to non-weighted graphs and also non-negative weighted graphs. However, there is an issue with Dijkstra's algorithm, what happens if an edge weight is negative? Dijkstra assumes that when you find the shortest path to a place, you *never* need to check it again, if the graph has a negative weighted edge this is suddenly no longer true and will lead to incorrect results because the algorithm *thinks* you are done when you are not. To effectively handle negative edge weights, we need a different algorithm which is exactly where Bellman-ford comes in.

8 Bellman-Ford

As I previously discussed, Bellman-Ford's algorithm, just like Dijkstra's, is designed to computer the shortest path from s - the starting vertex, to all other vertices in a weighted graph. This is one major difference between these two algorithms, and that is Bellman-Ford's algorithm can traverse graphs with negative edge weights. While Dijkstra grabs the closet-looking node and moves outwards into the graph greedily - meaning that it makes the choice that looks the best *right now* without worrying about future consequences. This means that Dijkstra's algorithm is very quick at the sacrifice of checking every single path, while the Bellman-Ford algorithm is more patient as it repetitively relaxes all of the edges and checks them if a new shorter path is formed and updates its current distance if it is. Think of this as you are constantly checking all of your routes every single day for $V - 1$ days just to be sure there is no shortcut, and if there is, update the shortest path [13]

8.1 How to apply the algorithm

Just like Dijkstra's algorithm we pick a source/root node s and set its distance to 0 because we are already at the vertex. All of the other nodes are set to ∞ as we have yet to explore them and don't know any paths between them. To keep track of the graph, we still have to use either an adjacency list or a matrix that contains not only the edges between two nodes but the weight of traversing between them.

, Unlike Dijkstra, we don't use anything like a priority queue or a min-heap data structure, we simply iterate through all edges of the graph and check if the distance to any node v can be improved upon based on our *current* best estimate. If the algorithm successfully finds a shorter path, we simply update our current best estimate. We need to repeat this process $V - 1$ times as in the worst-case scenario, a graph with no cycles involves $V - 1$ total edges. This means that we can find the shortest paths in a graph, even ones with negative edge weights because we are not "locking in" the shortest path when we get to it as Dijkstra's algorithm does, we take it more slowly and make sure every path is explored.

Have a look at the pseudo-code below to get an idea of how this would look in code [12].

```
function BellmanFord(Graph, source):
    // Initialize distances
    for each vertex v in Graph:
        dist[v] = infinity
        prev[v] = undefined
    dist[source] = 0

    // Relax edges repeatedly
    for i from 1 to size(Graph.V) - 1:
        for each edge (u, v) with weight w in Graph:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                prev[v] = u

    // Check for negative-weight cycles
    for each edge (u, v) with weight w in Graph:
        if dist[u] + w < dist[v]:
            error "Graph contains a negative-weight cycle"

    return dist[], prev[]
```

Other

One thing to note here is that not every reference was used directly, most were additional research and videos to strengthen my understanding for a personal purpose. All of the resources have been read and watched by me, however, only a few were used directly in the submission.

References

[1] Wikipedia contributors. (2025, March 28). *Adjacency list*. Wikipedia.

[2] Wikipedia contributors. (2025, March 28). *Adjacency matrix*. Wikipedia.

[3] Graph Data Structures *(Adjacency Matrix, adjacency list, edge List)* - VisuAlgo. (n.d.).

[4] Revathy Govindarasu. (2021, February 27). *A Beginners guid to BFS and DFS*. LeetCode.com

[5] Reducible. (2020, September 26). *Breadth First Search (BFS): visualized and explained*. YouTube. https://www.youtube.com/watch?v=xlVX7dXLS64

[6] Reducible. (2020, July 5). *Depth First Search (DFS) explained: algorithm, examples, and code*. https://www.youtube.com/watch?v=PMMc4VsIacU

[7] Pranshu Ranjan Singh. (2018, March 25). *Sudoku Solver (Depth First Search)* [Video]. YouTube. https://www.youtube.com/watch?v=hZWSqg–01k

[8] Joshi, V. (2018, June 21). *Deep Dive Through A Graph: DFS Traversal*. Medium. https://medium.com/

[9] *BFS Graph Algorithm(With code in C, C++, Java and Python)*. (n.d.). https://www.programiz.com/dsa/graph-bfs

[10] Wikipedia contributors. (2025, April 8). *Dijkstra's algorithm*. Wikipedia.

[11] W3Schools.com. (n.d.). *Dijkstras algorithm* W3Schools

[12] W3Schools.com. (n.d.). *Bellman-Ford algorithm* W3Schools

[13] Wikipedia contributors. (2024, November 27). *Bellman–Ford algorithm*. Wikipedia.