Graph Theory Deep Dive: Shortest Path Algorithms

Abby Omer, Adi Sudhakar, Antoinette Tan, Dasha Chadiuk

November 1, 2021

1 Introduction

1.1 Weighted Graphs and Shortest Path

Many problems can be modeled using graphs with weights assigned to their edges. These types of graphs are called weighted graphs. An example of a weighted graph can be seen in Figure 1. This graph represents the time it takes to go between different places at Olin (minutes:seconds), with each edge representing a unique pathway between two places. Note that the numbers in this graph are purely for demonstration purposes, and do not directly correlate with real life. Weighted graphs can be used to determine a path of least length between two vertices in a network. The length of a path in a weighted graph is the sum of the weights of the edges of the path [1]. So, for example, we can use the shortest path in Figure 1 to find what path a student should take if they live in East hall and are about to be late for a class in the MAC. In this Collaborative Deep Dive, we will be exploring shortest path algorithms.

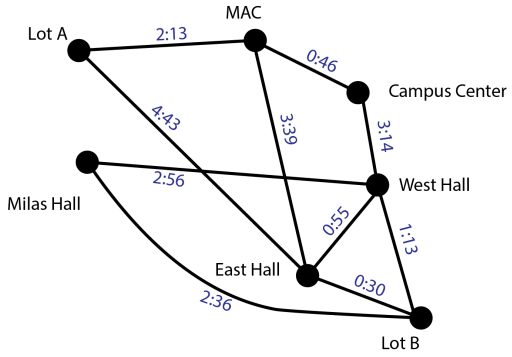

Figure 1: Weighted graph representing the time it takes to go to different places at Olin.

1.2 General Applications and Uses

Shortest path algorithms can be used to solve a number of questions. Some examples of their uses are finding the shortest path in air distance between Boston and Los Angeles, the least expensive set of telephone lines needed to connect computers in San Francisco with those in New York, finding the shortest path to visit 8 different cities exactly once, or even how to solve a Rubik's cube. In the case of the Rubik's cube, for example, the vertices of the graph would be a set of all possible configurations of the cube, and the edges are between those configurations that are one turn away from one another. Then, by finding the shortest path from the node representing the cube's current

state, to the node representing its solved state, one can find the least amount of move's necessary to solve the cube [3].

2 Preliminaries

In this section, we will explicitly define some of the vocabulary needed to talk about shortest path algorithms.

Definition 2.1 *Graphs are discrete structures consisting of vertices and edges that connect these vertices.*

Example 2.2 *The graph seen in Figure 2 represents people by vertices and connects two vertices when the people are friends.*

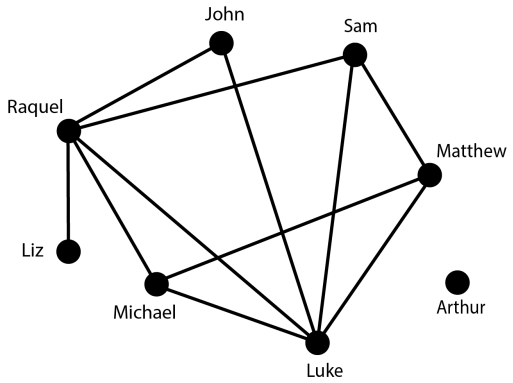

Figure 2: Graph representing friends among a set of people.

Definition 2.3 *A path is a sequence of edges that begins at a vertex of a graph and travels along edges of the graph, always connecting pairs of adjacent vertices.*

Definition 2.4 *An undirected graph is a set of vertices together with a set of undirected edges that connect these vertices.*

Definition 2.5 *A simple graph is an undirected graph with no multiple edges or loops.*

Definition 2.6 *A circuit is a path of length $n \geq 1$ that begins and ends at the same vertex.*

Definition 2.7 *A tree is a connected undirected graph with no simple circuits.*

Example 2.8 *The graph seen in Figure 3 is a tree that represents a family.*

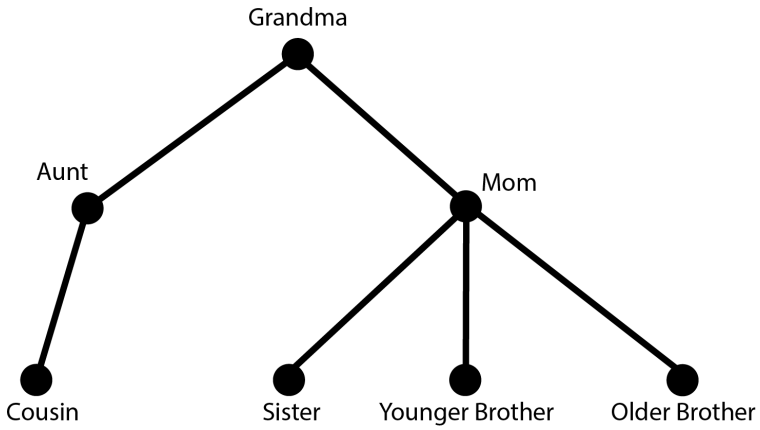

Figure 3: Graph representing a family tree.

Definition 2.9 *Time complexity is the amount of time taken by an algorithm to run, as a function of the length of the input. It measures the time taken to execute each step in an algorithm [2].*

3 Shortest Path Algorithms

3.1 Dijkstra's Algorithm

Dijkstra's algorithm, which was presented in the textbook, was devised by Edsger Dijkstra in 1956. While in its original form can be used to find the shortest path between two vertices of an edge weighted graph, a common variant of the algorithm is finding the shortest path from a source vertex to all other vertices, producing a shortest path tree. In its original form, the algorithm uses a min-priority queue and has a time complexity of $O((|V| + |E|) \log |V|)$, where $|V|$ is the number of vertices in the graph and $|E|$ is the number of edges in the graph. However, by using an array, it can be reduced to $O(|V|^2)$. Variants of the algorithm for specialized cases can be used to lower the time complexity. For example, Dial's algorithm has $O(|E| + |V|C)$ where C is the maximum weight for graphs with positive integer edge weights [5].

3.2 Prim's Minimal Spanning Tree Algorithm

Prim's minimal spanning tree algorithm, also called Jarník's algorithm, is the predecessor of Dijkstra's algorithm. This algorithm forms a minimum spanning tree, a subgraph that connects all vertices of an edge weighted graph without cycles such that the subgraph has the minimum possible total edge weight. Unlike Dijkstra's algorithm which finds the shortest path from a specific vertex to one or all other vertices, Prim's algorithm finds the shortest way to connect all vertices of a graph. This is useful when designing networks such as computer and telecommunications networks when all vertices need to be connected and edges might be something like wires which cost money to lay down [6]. The time complexity of this algorithm depends on the data structures used for the graph. For an adjacency matrix with linear searching, $O(|V|^2)$. For a binary heap and adjacency list, $O((|V| + |E|) \log |V|) = O(|E| \log |V|)$. For a Fibonacci heap and adjacency list, $O(|E| + |V| \log |V|)$ [7].

3.3 Bellman-Ford Algorithm

The Bellman-Ford algorithm, also known as Bellman-Ford-Moore Algorithm, like Dijkstra's algorithm is used to find the shortest path from a source vertex to all other vertices. However, unlike Dijkstra's algorithm, the Bellman-Ford algorithm can be used on graphs with negative weighted edges. This algorithm operates by finding the shortest paths with at most one edge, then two edges, then three and so on, updating the shortest path distance for each vertex as it goes. Without a cycle, the longest possible path can be $|V| - 1$ edges, so the algorithm must scan the graph $|V| - 1$ times to ensure the minimum distance path has been found for each vertex. However, if the minimum distance values of each vertex do not change after a scan, then the algorithm can terminate early, but the worst case condition remains $|V| - 1$. Scanning the graph with an increasing number of edges allows the Bellman-Ford algorithm to handle negative weighted edges at the cost of time, leaving it with a time complexity of $O(|V||E|)$, slower than Dijkstra's algorithm. Because negative cycles, where the distance to a vertex is made less by going around the cycle more times, would produce wrong answers, the Bellman-Ford algorithm terminates when it finds one [8].

3.4 A* Algorithm

The A* search algorithm is an extension of Dijkstra's algorithm published by Peter Hart, Nils Nilsson, and Bertram Raphael. The algorithm is an informed search algorithm that finds the least distance path for a graph with weighted edges. It maintains a path tree, and the algorithm determines which path to extend by an edge based on the current total weight of the path and the estimated weight of the path to its end vertex with a heuristic function. A* selects that path that minimizes $f(n) = g(n) + h(n)$ where n is the next vertex, $g(n)$ is the total weight of the path from the source vertex to n, and $h(n)$ is the heuristic function. Dirkstra's algorithm can be considered a special case of the A* algorithm where $h(n) = 0$ for all vertices n. The time complexity of the A* algorithm depends on the heuristic function with the worst case being $O(b^d)$ where d is the depth of the shortest path and b is the branching factor, the average number of successors per state. The A* algorithm has many variants for different specialized cases [9].

3.5 Floyd-Warshall Algorithm

The Floyd-Warshall algorithm finds the shortest path for a directed edge weighted graph with positive or negative weights but no negative cycles. The algorithm compares all possible paths through a graph, finding the shortest path going through each vertex, leading to a time complexity of $O(|V|^3)$. This is useful with dense graphs when the number of edges is close to the maximum number since the time complexity is only dependent on the number of vertices [10].

3.6 Johnson's Algorithm

Johnson's algorithm is a combination of the Bellman-Ford algorithm and Dijkstra's algorithm that finds shortest paths between all vertices in a directed edge weighted graph with positive or negative weights. It first removes all negative weighted edges by adding a vertex q that is connected to all other vertices by zero weighted edges, and then using the Bellman-Ford algorithm to find the shortest path to all vertices. If the path is negative, then the distance to vertex v, represented by $h(v)$, is negative, and if it is positive, then the shortest path is $h(v) = 0$. The edges are rewritten from its original weight $w(u, v)$ to $w(u, v) + h(u) - h(v)$ where u is the starting vertex and v is the ending vertex. This then allows Dijkstra's algorithm to be used on the graph to find the shortest path, returning a path weight of $d(u, v)$ for the adjusted graph. The total weight of the path on

the unadjusted graph can then be found by $d(u, v) + h(v) - h(u)$. By using Fibonacci heaps in Dijkstra's algorithm, the time complexity of Johnson's algorithm is $O(|V|^2 \log |V| + |V||E|)$. This time complexity means that it is useful for sparse graphs when it is faster than the Floyd-Warshall algorithm [11].

3.7 Implementation of the Bellman-Ford and Dijkstra's Algorithms

We chose to implement two shortest path algorithms: Dijkstra's Algorithm which we learned in class and the Bellman-Ford algorithm, which is similar to Dijkstra's but handles negative edge weights. The code for both algorithms can be found in the appendix. We ran the algorithms on two different graphs. The first had 4 vertices and 6 edges; the second had 7 vertices and 20 edges. For each graph, we ran both algorithms 1000 times and plotted the time it took for each algorithm to run on a histogram.

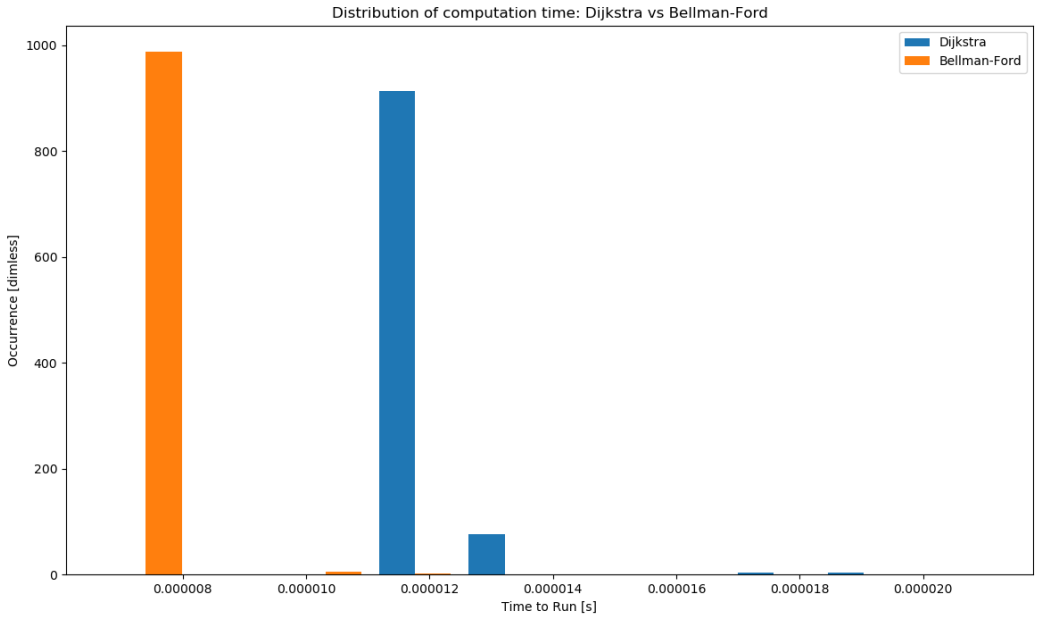

Figure 4: Comparing the distribution of computation time of Dijkstra's and Bellman-Ford algorithms for the first graph (4 vertices, 6 edges).

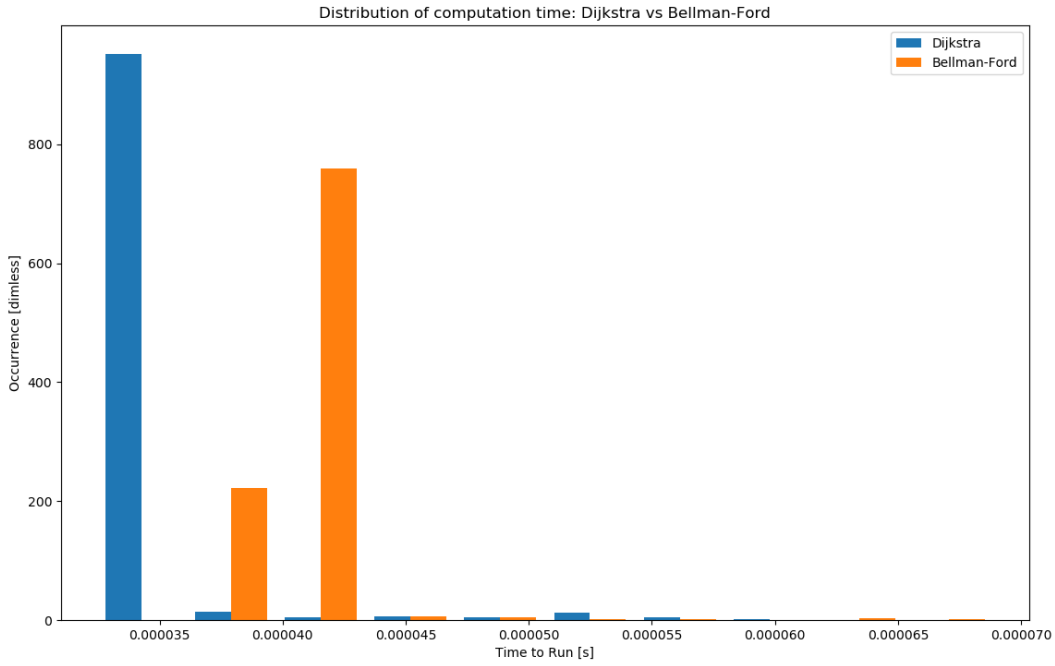

Figure 5: Comparing the distribution of computation time of Dijkstra's and Bellman-Ford algorithms for the second graph (7 vertices, 20 edges).

After running the algorithms with the first graph, it looked like the Bellman-Ford algorithm was faster at finding the optimized paths between vertices. Most of the Bellman-Ford runs took 0.000008 seconds compared to 0.000012 seconds for Dijkstra's, as seen in Figure 4. However, as the graph becomes more complex, Dijkstra's algorithm proves to be more efficient, as seen in Figure 5. While the Bellman-Ford algorithm has certain advantages, such as the ability to account for negative weights, Dijkstra's algorithm can be the better choice for applications that do not require negative weights.

4 Who Did What?

Antoinette: Wrote the introduction, explaining shortest path algorithms.
Abby: Wrote the introduction, explaining shortest path algorithms.
Adi: Implemented the Dijkstra's shortest path algorithm in python.
Dasha: Implemented the Bellman-Ford Shortest Path algorithm in python.

5 Appendix

```
'''
Implements the Bellman Ford and Dijkstra's shortest path algorithms.
Authors: Adi Sudhakar and Dasha Chadiuk
'''
```

```
import time
import matplotlib.pyplot as plt
from sys import maxsize
import numpy as np

#### VARIABLES ####
def redefine_variables():
    max_val = maxsize #placeholder super big number to represent infinity
    visit_dist = [[0, 0]] #[[visited?, distance to],]
    return max_val, visit_dist

'''
#### DEFINE GRAPHS #####
def make_graph():
    # define connections between verticies
    connections = [[0 , 1 , 0 , 1],
                   [1 , 1 , 1 , 0],
                   [0 , 1 , 1 , 1],
                   [1 , 0 , 1 , 0]]

    #define weight of each edge generated
    edge_weight = [[0 , 10 , 0 , 1],
                    [10 , 10 , 1 , 0],
                    [0 , 1 , 10 , 5],
                    [1 , 0 , 5 , 0]]

    total_vertices = len(connections[0])

    return connections, edge_weight, total_vertices
'''

def visit_next():
    #Identify which vertex to visit next to achieve shortest path
    visited = -1

    # Choosing the vertex with the minimum distance
    for vertex in range(total_vertices):
        if visit_dist[vertex][0] == 0 and \
                (visited < 0 or visit_dist[vertex][1] \
                <= visit_dist[visited][1]):
            visited = vertex

    return visited

def run_dijkstra(connections, edge_weight, total_vertices):
    start = time.time()
    #default min length to inf
    # visit_dist = []
    for vertex in range(total_vertices-1):
        visit_dist.append([0, max_val])

    for vertex in range(total_vertices):
    # Finding the next vertex to be visited.
```

```
        to_visit = visit_next()
        for adj_vertex in range(total_vertices):
            # Calculate distance to unvisited adjacent vertices
            if connections[to_visit][adj_vertex] == 1 and \
                    visit_dist[adj_vertex][0] == 0:
                updated_dist = visit_dist[to_visit][1] + \
                        edge_weight[to_visit][adj_vertex]
            # Updating the distance of the adjacent vertex
                if visit_dist[adj_vertex][1] > updated_dist:
                    visit_dist[adj_vertex][1] = updated_dist
        # mark as visited
        visit_dist[to_visit][0] = 1
    loc = 0

    # Printing out the shortest distance from the source to each vertex
    for distance in visit_dist:
        # FOR DEBUG ONLY. COMMENT OUT 'PRINT' LINE FOR GETTING TIME DATA
        # print("The shortest path to vertex [",(ord('a') + loc),"]
        #from vertex [a] is:",distance[1])
        loc = loc + 1
    end = time.time()

    duration = end - start
    return duration


def bellman_ford(graph, total_vertices, total_edges, source):
    '''
    Runs the Bellman Ford algorithm on a given graph, determining the distance
    of all vertices from a source vertex.
    Args:
        graph: a graph in the form of [[u1,v1,w1],...,[un,vn,wn]] where each
        edge is from u to v, and w is the weight of the edge.
        total_vertices:an int representing the total number of vertices in the
        graph
        total_edges: an int representing the total number of edges in the graph
        source: an int representing the source vertex
    '''

    start = time.time()
    #define all distances between verteces as infinite
    dist = [maxsize]*total_vertices
    #make the source vertex distance as 0
    dist[source] = 0

    '''
    Go through each of the edges and check whether the recorded distance
    between the edges is smaller than the path distance being calculate.
    Replace it with the new distance if it is smaller.
    '''
    for vertex in range (total_vertices-1):
        for edge in range (total_edges):
            if dist[graph[edge][0]] + graph[edge][2] < dist[graph[edge][1]]:
                dist[graph[edge][1]] = dist[graph[edge][0]] + graph[edge][2]
```

```
        #check if there is a negative weight
        for edge in range (total_edges):
            start_vertex = graph[edge][0]
            end_vertex = graph[edge][1]

            weight = graph[edge][2]
            if dist[start_vertex] != maxsize and \
                   dist[start_vertex] + weight < dist[end_vertex]:
                      # print("Contains negative weight")
                      pass
    '''
    print("Vertex Distance")

    #print the vertex and the distance to that vertex from the source
    for vertex in range(total_vertices):
        print(str(vertex) + " " + str(dist[vertex]))
    '''

    end = time.time()
    duration = end - start
    return duration

def graph_to_adjacency(graph, total_vertices, total_edges):
    '''
    Converts from a u,v,w graph (where each edge is from u to v, and w is the
    weight between them) into two adjacency matrices connections, and
    edge_weights.
    Args:
        graph: a graph in the form of [[u1,v1,w1],...,[un,vn,wn]] where each
        edge is from u to v, and w is the weight of the edge.
        total_vertices: an int representing the total number of vertices in
        the graph
        total_edges: an int representing the total number of edges in the graph
    '''
    connections = np.zeros((total_vertices, total_vertices),int)
    edge_weights = np.zeros((total_vertices, total_vertices),int)
    for edge in range(total_edges):
        connections[graph[edge][0]][graph[edge][1]] = 1
        connections[graph[edge][1]][graph[edge][0]] = 1
        edge_weights[graph[edge][0]][graph[edge][1]] = graph[edge][2]
        edge_weights[graph[edge][1]][graph[edge][0]] = graph[edge][2]

    return connections, edge_weights


def plot_times_hist(dij_times, bell_times):
    plt.hist([dij_times, bell_times], label=['Dijkstra', 'Bellman-Ford'])
    plt.legend(loc='upper right')
    plt.title('Distribution of computation time: Dijkstra vs Bellman-Ford')
    plt.xlabel('Time to Run [s]')
    plt.ylabel('Occurrence [dimless]')
    plt.show()
```

```
if __name__ == "__main__":

    dij_times, bell_times = [], []
    max_val, visit_dist = redefine_variables()

    total_vertices = 7
    total_edges = 20


    '''
    create a u,v,w graph where u is the starting vertex, v is the ending
    vertex, and w is the weight of the edge between them
    '''

    '''
    graph = [
        [0,1,10],
        [1,1,10],[1,2,1],
        [2,2,10],[2,3,5],
        [3,0,1]
            ]
    '''
    graph = [
        [0,1,10],[0,4,10],[0,5,6],[0,6,20],
        [1,1,10],[1,2,1],[1,4,13],[1,5,20],
        [2,2,10],[2,3,5],[2,5,4],[2,4,8],[2,6,5],
        [3,0,1],[3,6,5],[3,4,5],[3,5,20],
            [4,5,20], [4,6,20],
            [5,6,10]
            ]

    #convert graph into 2 matrices of vertex connections and edge weights:
    connections, edge_weight = \
            graph_to_adjacency(graph, total_vertices, total_edges)

    for j in range(1000):
        max_val, visit_dist = redefine_variables()
        duration = run_dijkstra(connections, edge_weight, total_vertices)
        dij_times.append(duration)

    for j in range (1000):
        #run the bellman ford algorithm and determine the time it takes to run
        duration = bellman_ford(graph, total_edges, total_vertices, 0)
        bell_times.append(duration)

    plot_times_hist(dij_times, bell_times)
```

References

[1] Rosen, Kenneth H. Discrete Mathematics and Its Applications. Fifth ed., McGraw-Hill, 2003.

[2] Great Learning Team. "Why is Time Complexity Essential and What is Time Complexity?" GreatLearning Blog: Free Resources What Matters to Shape Your Career!, 17 Aug. 2021, https://www.mygreatlearning.com/blog/why-is-time-complexity-essential/.

[3] Chen, Isaac. "Common Rubik's Cube Algorithms for Machines." Medium, Nerd For Tech, 19 Feb. 2021, https://medium.com/nerd-for-tech/common-rubiks-cube-algorithms-for-machines-part-1-of-2-in-a-quest-to-understand-the-rubik-s-cube-3cb9b53f94b5.

[4] "Shortest Path Problem." Wikipedia, Wikimedia Foundation, 12 July 2021, https://en.wikipedia.org/wiki/Shortest_path_problem.

[5] "Dijkstra's Algorithm." Wikipedia, Wikimedia Foundation, 18 Oct. 2021, https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.

[6] "Minimum Spanning Tree." Wikipedia, Wikimedia Foundation, 7 Sept. 2021, https://en.wikipedia.org/wiki/Minimum_spanning_tree.

[7] "Prim's Algorithm." Wikipedia, Wikimedia Foundation, 16 Sept. 2021, https://en.wikipedia.org/wiki/Prim%27s_algorithm.

[8] "Bellman–Ford Algorithm." Wikipedia, Wikimedia Foundation, 14 Aug. 2021, https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm.

[9] "A* Search Algorithm." Wikipedia, Wikimedia Foundation, 17 Oct. 2021, https://en.wikipedia.org/wiki/A*_search_algorithm.

[10] "Floyd–Warshall Algorithm." Wikipedia, Wikimedia Foundation, 17 Oct. 2021, https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm.

[11] "Johnson's Algorithm." Wikipedia, Wikimedia Foundation, 21 Aug. 2021, https://en.wikipedia.org/wiki/Johnson%27s_algorithm.