Project 3

CIIC4070 – Computer Networks

Prof. Kejie Lu

Carlos A. Rodriguez

802-21-6906

CIIC

Table of Contents

Section 1: Introduction

This report will consist of:

1) Learning the basics of shortest path routing problems.
2) How to formulate a shortest path routing problem.
3) Learn about algorithms utilized in shortest path routing problems.
4) Implement an algorithm to find the shortest path between a source and destination node in a graph.
5) YouTube Video: https://www.youtube.com/watch?v=hZf6o2uPsR4 [15]

Section 2: Basics of the shortest path routing problem

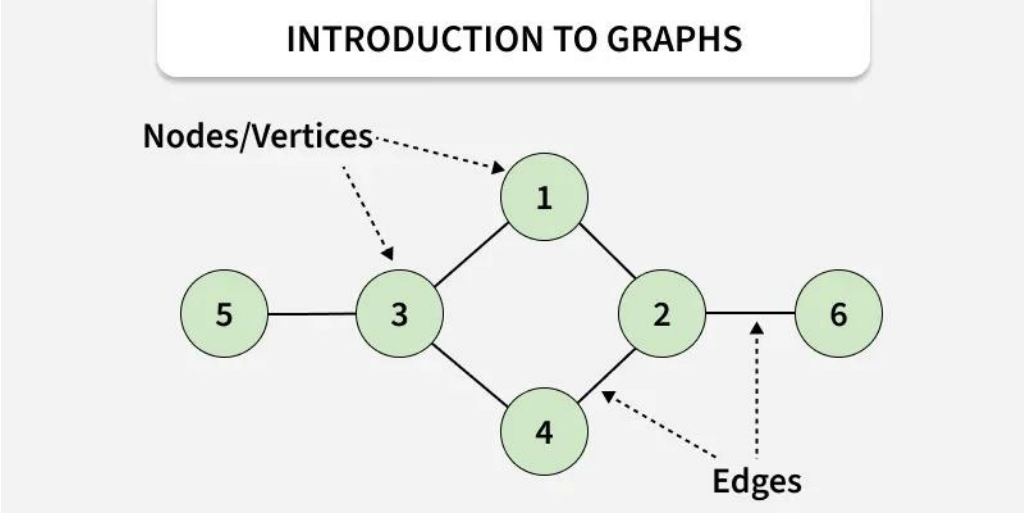
1. What is a graph?
    a. According to [1], a **graph data structure** is a collection of nodes connected by edges. It's utilized to represent relationships between different entities. **Graph Algorithms** are developed for and utilized in graphs, and as stated by [2], graphs do not follow a sequential order like arrays or linked lists do.
    i. 

Figure 1: Example of Graph from [2].
2. How do we represent a network using a graph?
    a. As stated in [3], Networks are real-world applications of graphs. They take the abstract concept of a graph and apply it to practical scenarios. An example in [3] is transportation networks, where vertices represent different locations and edges represent interactions between them. Sometimes these graphs are **weighted** to better determine the information regarding the relationship between two vertices, such as how a bigger weight in the transportation network can mean a longer distance.
3. What is the shortest path routing problem?
    a. According to [4], the problem is defined as finding a path between two nodes in a graph such that the sum of the weights of its constituent edges is minimized. In **unweighted graphs**, the shortest path is the one with the least number of edges, while in **weighted graphs**, the shortest path is the one with the lowest total sum of edge weights.

Section 3: Formulation of the shortest path routing problem

1. Define all necessary variables [5][6]
    a. Weighted Graph $G = (V, E, f)$
        i. V *is the set of vertices* $(nodes)$
        ii. E *is the set of edges* $(links)$
        iii. $f: E \rightarrow \mathbb{R}$ which assigns a real-value weight to each edge
    b. Source node $s \in V$
    c. Destination node $d \in V$
2. Define an objective function [5][6]
    a. We can use QUBO Encoding utilizing $|V| + |E| - 2$ binary variables $\{0,1\}$
    b. Valid Path Definition
        i. Starts from specified source node s
        ii. Ends at specified destination node d
        iii. No broken links in path between $s \rightarrow d$
    c. First $|V| - 2$ variables correspond to nodes in the graph excluding s and t
        i. Denoted as x_i where $i \in V$ corresponds to a node
    d. Remaining $|E|$ variables represent the Edges E
        i. Denoted as x_{ij} where $(i, j) \in E$ represents an edge present in G between nodes i and j
    e. Define state x as $x = (x_1, \dots, x_{|V|-2}, \dots, x_{ij}, \dots.)$. This encoding implicitly specifies the starting and ending nodes, since they are always in the path.
    f. After defining all of this, the QUBO cost function can be written as
        i. $C(x) = C_w(x) + C_P(x)$, where:
            1. $C_w(\mathbf{x})$ contains linear terms that encode the node and edge weights w_i and w_{ij} respectively:
            $$C_w(\mathbf{x}) = \sum_{i \in V} w_i x_i + \sum_{(i,j) \in E} w_{ij} x_{ij},$$
            2. $$C_P(\mathbf{x}) = C_s(\mathbf{x}) + C_d(\mathbf{x}) + C_{path}(\mathbf{x})$$
    g. C_P is the function that penalizes states with bad paths, which essentially explain the constraints in our problem. Penalization is needed to ensure paths accurately represent how optimal they are going from source to destination.
3. Define all constraints [6]
    a. Going off this formula:
        i. $$C_P(\mathbf{x}) = C_s(\mathbf{x}) + C_d(\mathbf{x}) + C_{path}(\mathbf{x})$$

b. We have three different types of constraints to consider regarding our shortest path routing problem:
    i. Source Constraint: Penalizes paths that do not have exactly one edge connected to the source node.
        1. $$C_s(\mathbf{x}) = -x_s^2 + (x_s - \sum_j x_{sj})^2,$$
        2. Where $x_s = 1$, and the sum is over all edges that are connected to node s. This term has a minimum value of -1, which occurs for states where there is only one edge connected to the source node.
    ii. Destination Constraint: Penalizes paths that do not have exactly one edge connected to the destination node.
        1. $$C_d(\mathbf{x}) = -x_d^2 + (x_d - \sum_j x_{dj})^2,$$
        2. Where $x_d = 1$, and the sum is over all edges that are connected to node d. This term has a minimum value of -1, which occurs for states where there is only one edge connected to the destination node.
    iii. Path Constraint: Penalizes paths that do not have exactly 2 edges connected to intermediate nodes.
        1. $$C_{path}(\mathbf{x}) = \sum_{i \in V} C_i(\mathbf{x})$$
        2. Such that for every node $i \in V$
        3. $$C_i(\mathbf{x}) = (2x_i - \sum_j x_{ij})^2,$$
        4. Where the sum is over all edges that are connected to node i. This has a minimum value of 0, which occurs for states where all nodes selected have exactly 2 edges connected to it.

Section 4: Shortest Path Algorithms

1. What is Dijkstra's Algorithm?
    a. According to [7], Dijkstra's algorithm is an algorithm utilized mostly in **weighted**, **undirected** graphs to try and find the shortest path between any two nodes. It always picks the node with the least distance first, ensures each node is processed only once, and all its neighbors are explored immediately with the shortest distance.
2. What is the Floyd-Warshall Algorithm?
    a. According to [8], the Floyd-Warshall algorithm can be utilized in both **undirected weighted** and **directed** graphs to try and find the shortest path between any two nodes. It works by maintaining a two-dimensional array that represents distances between nodes. Initially, this array is filled using only the direct edges between nodes, and the algorithm gradually updates these distances by checking if shorter paths exist through intermediate nodes. It does not work for graphs with negative cycles.
3. What is the Bellman-Ford algorithm?
    a. According to [9], the Bellman-Ford algorithm is best suited to finding the shortest paths in a **directed** graph, with one or more negative edge weights, from the source vertex to all other vertices. It repeatedly checks all edges in the graph for shorter paths, as many times as there are vertices in the graph – 1. It can also be used for positively edged graphs, but Dijkstra's is better in that domain.
4. What is the A* algorithm?
    a. According to [10], the A* algorithm is an informed search algorithm, meaning it leverages a heuristic function to guide its search towards the goal. This heuristic function estimates the cost of reaching the goal from a given node, allowing the algorithm to prioritize promising paths and avoid exploring unnecessary ones. It is used a lot in game development, robotics, and machine learning. It cannot be used for graphs with negative cycles.
5. What is the algorithm you want to implement in this project?
    a. I have a keen interest in trying to implement the Dijkstra algorithm.
6. Why did you choose said algorithm?
    a. I chose to implement Dijkstra's algorithm due to its extensive documentation online, as well as all the practical applications this algorithm can be used for. In [13], we can see it can be used for: Navigation Systems, Network Routing, Transportation, Communications, Social Networks, and many more.

Section 5: Implementation of a shortest path algorithm

Graph Class + Vertex and Adjacency Lists

```python
# Graph Class
class Graph():
    def __init__(self, vert_list, adj_list):
        self.vert_list = vert_list
        self.adj_list = adj_list

# Manually Constructed Vertex List
num_vertices = 14
vert_list = []
for i in range(1,num_vertices+1):
    vert_list.append(i)

# Manually Constructed Adjacency List
# adj_list[vertex] = [(neighbor,weight), (neighbor,weight)...]
adj_list = {}
adj_list[1] = [(2,1050), (3,1500), (8,2400)]
adj_list[2] = [(1,1050), (3,600), (4,750)]
adj_list[3] = [(2,600), (1,1500), (6,1800)]
adj_list[4] = [(2,750), (11,1950), (5,600)]
adj_list[5] = [(4,600), (7,600), (6,1200)]
adj_list[6] = [(3,1800), (5,1200), (14,1800), (10,1050)]
adj_list[7] = [(5,600), (8,750), (10,1350)]
adj_list[8] = [(7,750), (1,2400), (9,750)]
adj_list[9] = [(8,750), (12,300), (13,150), (10,750)]
adj_list[10] = [(9,750), (7,1350), (6,1050)]
adj_list[11] = [(4,1950), (13,750), (12,600)]
adj_list[12] = [(11,600), (9,300), (14,300)]
adj_list[13] = [(9,300), (11,750), (14,150)]
adj_list[14] = [(12,300), (13,150), (6,1800)]
```

Priority Queue Class

```python
# Priority Queue Class
# Obtained from reference 12
class PQ():
    def __init__(self):
        self.q = []

    def insert(self, d):
        self.q.append(d)

    def delete(self):
        try:
            m = 0
            for i in range(len(self.q)):
                # if self.q[i] < self.q[m]:
                if self.q[i][1] < self.q[m][1]:
                    m = i
            item = self.q[m]
            del self.q[m]
            return item
        except IndexError:
            print("Queue empty.")
            exit()

    def is_empty(self):
        return len(self.q) == 0
```

Dijkstra Algorithm Implementation + Tests + Complexity Analysis

```python
# Develop Dijkstra's Algorithm
# We use Distance Array, Previous Node Array and Priority Queue
# Developed using reference 7
def dijkstra(Graph, source, destination):
    dist = [] # Distance Array
    pq = PQ() # Priority Queue
    prev = [] # Previous Node Array

    # Initialize distance array to infinity
    for vertex in range(1, num_vertices+1):
        if vertex == source:
            dist.append(0)
        else:
            dist.append(float('inf'))
        prev.append(None)
    print(f"Distances before = {dist}")
    pq.insert((source, 0))

    # Continue till PQ is empty
    while not pq.is_empty():
        (u, distance) = pq.delete()
        if distance > dist[u-1]:
            continue

        vert_index = 0
        weight_index = 1
        for v in Graph.adj_list[u]:
            if dist[v[vert_index]-1] > distance + v[weight_index]:
                dist[v[vert_index]-1] = distance + v[weight_index]
                prev[v[vert_index]-1] = u
                pq.insert((v[vert_index], dist[v[vert_index]-1]))
            # print(v)
        # print("---")

    path = []
    current = destination
    while current is not None:
        path.append(current)
        current = prev[current-1]
    print(f"Distances after = {dist}")
    print(f"The shortest distance from {source} to {destination} is {dist[destination-1]}")
    print(f"The path is {path}")

# Please take into account this setup only works properly on the graph provided in Project 3.
# To work with other undirected positive graphs, the vertex and adjacency list need to be changed
accordingly.
# The code of the algorithm itself + the PQ implementation should work on any undirected positive
graphs.

network = Graph(vert_list, adj_list)
print("TEST 1")
test_1 = dijkstra(network, 12, 3) # New York to California 2 (Answer: 3900)
print("---")
print("TEST 2")
test_2 = dijkstra(network, 1, 10) # Washington to Georgia (Answer: 3900)
print("---")
print("TEST 3")
test_3 = dijkstra(network, 2, 13) # California 1 to New Jersey (Answer: 3450)

# Complexity Analysis
# Initializing distance and previous node arrays -> O(V) (1)
# Scanning for vertex neighbour elements in the graph -> O(E) (2)
# In the while loop, PQ delete scans the whole PQ -> O(|PQ|) (3)
# Deletes happen minimum as many times as inserts do -> O(|PQ|) (4)
# Since both deleting and inserting to the PQ happen inside the code that checks neighboring elements,
# these can occurr as many times as there are neighbors of a node. Therefore, we can reduce (3) and (4)
-> O(E^2)
# Final Tally: O(V + E + E^2) -> O(E^2)
# In this case, we have 44 edges to explore, meaning 44^2 = 1936 operations
```

Outputs of Dijkstra Algorithm

```
TEST 1
Distances before = [inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, 0, inf, inf]
Distances after = [3450, 3300, 3900, 2550, 2400, 2100, 1800, 1050, 300, 1050, 600, 0, 450, 300]
The shortest distance from 12 to 3 is 3900
The path is [3, 6, 14, 12]
---
TEST 2
Distances before = [0, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]
Distances after = [0, 1050, 1500, 1800, 2400, 3300, 3000, 2400, 3150, 3900, 3750, 3450, 3300, 3450]
The shortest distance from 1 to 10 is 3900
The path is [10, 9, 8, 1]
---
TEST 3
Distances before = [inf, 0, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf]
Distances after = [1050, 0, 600, 750, 1350, 2400, 1950, 2700, 3450, 3300, 2700, 3300, 3450, 3600]
The shortest distance from 2 to 13 is 3450
The path is [13, 11, 4, 2]
```

Formal Analysis of Time Complexity

The algorithm uses an adjacency list representation of the graph and an unsorted-list priority queue. This implementation is considered greedy according to [7].

Initialization

Initializing the distance and predecessor arrays requires one pass over all vertices: $O(V); V = Vertices$

Relaxing edges

During the algorithm, each adjacency list is scanned once when its source vertex is extracted from the priority queue.
Total neighbor scanning over the whole run is $O(E); E = Edges$

Priority Queue Operations

The priority queue is implemented as an unsorted Python list.

Insert: $O(1)$

Delete-min: $O(|\ PQ\ |)$since it scans the entire list to find the smallest element.

In the worst case, every successful relaxation inserts a new element, so the queue may grow to $O(E)$ elements. There are also $O(E)$ delete operations.

Thus, total time spent in delete-min is:

$$O(E) \times O(E) = O(E^2)\backslash$$

Total Time Complexity

Summing up all contributions:

$$O(V) + O(E) + O(E^2) = \boxed{O(E^2)}$$

According to [14], the usual time complexity of Dijkstra's algorithm is $O((V + E)\log V)$, which is better than the complexity in our implementation but not by an incredible margin.

Evaluation Results and Discussion

To evaluate the correctness and behavior of our implemented Dijkstra algorithm, three test cases were executed on the weighted, undirected graph provided in Project 3. The graph contains 14 nodes, all with positive edge weights, which makes it suitable for Dijkstra's greedy selection strategy.

The algorithm was tested with the following source–destination pairs:

1. New York (12) → California 2 (3)
2. Washington (1) → Georgia (10)
3. California 1 (2) → New Jersey (13)

The results of these tests matched the expected values given in the project description. For the first test, the shortest distance from node 12 to node 3 was correctly computed as 3900, and the algorithm also returned the complete path by using the prev array to reconstruct the route. Similar accuracy was observed for the other two tests. These consistent outputs confirm that the algorithm successfully relaxes edges, updates tentative distances, and correctly prioritizes the smallest-distance node during each iteration.

Additionally, the algorithm's behavior aligns with the theoretical properties of Dijkstra's method. The use of a "visited-by-optimal-distance" mechanism, implemented through the check if distance > dist[u-1]: continue, ensures that outdated queue entries are ignored and that each vertex is finalized exactly once at its shortest possible distance. Even though the priority queue is implemented as an unsorted list (which increases running time), this does not affect correctness; it only affects efficiency.

The adjacency list representation proved effective for this specific graph. Since each vertex only stores its set of neighbors and associated weights, scanning edges were straightforward and efficient for the graph's relatively small size. All results were obtained quickly, demonstrating that even an $O(E^2)$ implementation is practical for modest network sizes. As stated in [14], a better time complexity can be achieved if the algorithm stopped once it found the destination node, and if it were to use a Fibonacci heap as well.

Section 6: Conclusions

Section 2 — Basics of the Shortest Path Routing Problem

Section 2 established the foundational concepts necessary to understand shortest path routing. This included defining what graphs are, how networks can be modeled using vertices and edges, and how weights represent real-world constraints such as distance or cost. By grounding the problem in these basic graph concepts, the section clarified how routing challenges naturally map onto graph structures, setting the stage for more advanced formulations.

Section 3 — Formulation of the Routing Problem

Section 3 formalized the shortest path routing problem using mathematical notation, including the weighted graph $G = (V, E, f)$, source and destination nodes, and a structured objective function. The section also introduced constraints through the QUBO encoding approach, emphasizing how penalties enforce valid paths. This formulation provided a precise, mathematical interpretation of the routing problem, highlighting how constraints guide the search toward optimal solutions.

Section 4 — Shortest Path Algorithms

Section 4 surveyed several classical shortest path algorithms—Dijkstra, Floyd-Warshall, Bellman-Ford, and A*. This comparison clarified the strengths, limitations, and use-case differences among them. It demonstrated why Dijkstra's algorithm is appropriate for graphs with non-negative weights and why it is widely used in networking contexts. This background informed the choice to implement Dijkstra in the project.

Section 5 — Implementation of Dijkstra's Algorithm

Section 5 presented the implementation of Dijkstra's algorithm using a manually constructed graph, adjacency lists, and a custom priority queue. The algorithm was tested on several source–destination pairs, producing correct shortest path distances and reconstructed paths. The complexity analysis showed that the unsorted-list priority queue leads to an $O(E^2)$runtime, which, while not optimal, is acceptable for the graph size in this assignment. The implementation demonstrated a working, fully functional shortest path solver.

References

[1] GeeksforGeeks, "Graph Data Structure," *GeeksforGeeks*. https://www.geeksforgeeks.org/dsa/graph-data-structure/

[2] GeeksforGeeks, "Introduction to Graphs – Data Structure and Algorithm Tutorials," *GeeksforGeeks*. https://www.geeksforgeeks.org/dsa/introduction-to-graphs-data-structure-and-algorithm-tutorials/

[3] Hypermode, "Graphs and Networks: A Gentle Introduction," *Hypermode Blog*, 2023. https://hypermode.com/blog/graphs-and-networks

[4] "Shortest Path Problem – an Overview," *ScienceDirect*. https://www.sciencedirect.com/topics/computer-science/shortest-path-problem

[5] F. B. Mourão, J. V. de Alencar, L. R. Pacífico, and M. A. F. Schmoeller, *The Shortest Path Problem*, Universidade da Região de Joinville, 2024.

[6] Entropica Labs, "Shortest Path Problem," *OpenQAOA*, 2024. https://openqaoa.entropicalabs.com/problems/shortest-path-problem/

[7] GeeksforGeeks, "Dijkstra's Shortest Path Algorithm," *GeeksforGeeks*. https://www.geeksforgeeks.org/dsa/dijkstras-shortest-path-algorithm-greedy-algo-7/

[8] GeeksforGeeks, "Floyd–Warshall Algorithm," *GeeksforGeeks*. https://www.geeksforgeeks.org/dsa/floyd-warshall-algorithm-dp-16/

[9] W3Schools, "Bellman-Ford Algorithm in Data Structures," *W3Schools*. https://www.w3schools.com/dsa/dsa_algo_graphs_bellmanford.php

[10] DataCamp, "A* Algorithm: Python Tutorial," *DataCamp*. https://www.datacamp.com/tutorial/a-star-algorithm

[11] GeeksforGeeks, "Priority Queue in Python," *GeeksforGeeks*. https://www.geeksforgeeks.org/python/priority-queue-in-python/

[12] Analytics Steps, "How Dijkstra's Algorithm is Used in Real Life," *Analytics Steps*. https://www.analyticssteps.com/blogs/how-dijkstras-algorithm-used-real-world

[13] GeeksforGeeks, "Time and Space Complexity of Dijkstra's Algorithm," *GeeksforGeeks*. https://www.geeksforgeeks.org/dsa/time-and-space-complexity-of-dijkstras-algorithm/

[14] Stack Overflow, "The Big-O of the Dijkstra Fibonacci Heap Solution," *StackOverflow*, 2014. https://stackoverflow.com/questions/21065855/the-big-o-on-the-dijkstra-fibonacci-heap-solution

[15] Carlos Rodriguez, “Project 3 Video | Networks,” *YouTube*, 2025. https://youtu.be/hZf6o2uPsR4