Lab 2 Report: Shortest path under constraints

Full Name: Nguyễn Lâm Minh Hòa

Student ID: 10422030

Class: CSE2023 - Group 2

Course: Algorithms and Data Structure

Instructor: Dr. Bùi Văn Thạch

Binh Duong, June 2025

No.	**Percentage understood**	**Content understood**	**Percentage Referenced**	**Content Referenced**	**Reference Source**
1.Input Parsing	20%	The structure of the input file, the layout of vertices, edges and the number of graphs.	80%	Basic syntax of “ifstream”, “getline”, and “stringstream ”.	https://cplusplus.com/doc/tutorial/files/ https://cplusplus.com/reference/sstream/stringstream/ https://cplusplus.com/reference/map/map/ Chatgpt.com Google.com
2. Dijkstra’s Algorithm with Traffic Light Constraints	40%	How Dijkstra's Algorithm Works. Dijkstra's Algorithm Code in C++	60%	Implementing Dijkstra’s algorithm and adapting it with waiting time due to traffic light cycles. Structure of the code.	https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/ https://cp-algorithms.com/graph/dijkstra.html

					Lecture 9: Graphs (in class) Chatgpt.com
3. Output Generation	30%	How to print the final result as required by the question. Writing results to both terminal and file, formatting path and travel time.	70%	File output with “ofstream”. How to connect code and output easily.	https://cplusplus.com/reference/fstream/ofstream/ https://stackoverflow.com/questions/10750057/how-do-i-print-out-the-contents-of-a-vector Chatgpt.com https://gemini.google.com/

1. Objective

The objective of this lab is to implement Dijkstra's algorithm to find the shortest path in undirected graphs with an additional constraint: each node has a traffic light with a specific cycle time. The algorithm must compute the shortest path from vertex **A** to vertex **G** while waiting at intersections if the light is red.

2. Problem Description

- An **undirected graph** is provided, where:
    - Each **vertex** has a name and a **cycle time** (representing the green light interval).
    - Each **edge** connects two vertices and has a travel time.
- The graph is represented in a file format, where the **first line** contains the number of graphs.
- Each graph includes:
    - A line indicating the **number of vertices**.
    - The next n lines describe the **vertices** and their cycle times in the format: VertexName, CycleTime.
    - The following lines define **edges** in the format: Vertex1, Vertex2, TravelTime.
- The program must read all graphs from the file, apply the shortest path algorithm with traffic light constraints, and write the output to a file.

3. Implementation Overview

The program is divided into three main stages:

Stage 1: Input Parsing

In this stage, we implemented the function read_graphs_from_file() to read multiple graphs from a text file.

- **Vertex Mapping**: Each vertex name is mapped to a unique index using a map<string, int>. This allows efficient array access.

- **Cycle Times**: Each vertex has a cycle_time, representing its traffic light cycle. These are stored in a vector<int>.

- **Adjacency List**: The graph is represented using an adjacency list of Edge structures.

```
//Read number of vertices for this graph
while (getline(file, line) && line.empty());
if (line.empty()) break;
graph.num_vertices = stoi(line);

//Read vertex names and their cycle times
for (int i = 0; i < graph.num_vertices; ++i) {
    getline(file, line);
    stringstream ss(line);
    string vertex;
    int cycle;
    getline(ss, vertex, ',');      //Vertex name before comma
    ss >> cycle;                   // cycle time after comma
    vertex.erase(remove(vertex.begin(), vertex.end(), ' '), vertex.end());

    //Store mappings and cycle time
    graph.vertex_to_index[vertex] = i;
    graph.index_to_vertex.push_back(vertex);
    graph.cycle_times.push_back(cycle);
}

graph.adjacency_list.resize(graph.num_vertices);  //initialize adjacency list
```

- **Multi-Graph Handling**: The code uses tellg() and seekg() to detect the beginning of the next graph.

```
//check if line is start of next graph (a number)
if (isdigit(token[0])) {
    file.seekg(last_pos);      // rewind to previous position
    break;                     // exit edge reading loop
}
```

Stage 2: Dijkstra's Algorithm with Traffic Light Constraints

This stage implements a modified Dijkstra's algorithm that considers wait times at traffic lights.

- **Distance Array**: dist[] keeps track of the shortest time to each node from A.

- **Wait Time Calculation**: At each node (except the starting node), if the current time doesn't align with the green light, we calculate the time to wait.

```
int v = e.to;
int travel_time = e.weight;
int wait_time = 0;

int current_time = dist[u];  // Current arrival time at u

// Skip traffic light if u is the starting vertex
if (u != start) {
    int cycle_u = g.cycle_times[u];
    if (current_time % cycle_u != 0) {
        wait_time = cycle_u - (current_time % cycle_u);  //
            Wait until next green
    }
}

int arrival_time = current_time + wait_time + travel_time;
```

- **Edge Relaxation**: The algorithm updates the distance and parent only if a faster arrival time is found.

```
// Relaxation step: if shorter path is found, update
if (arrival_time < dist[v]) {
    dist[v] = arrival_time;
    parent[v] = u;
}
```

- **find_min_distance_vertex()** is a helper function used to select the next unvisited vertex with the minimum distance.

```
// Find the vertex with the minimum distance value that has not been
    visited
int find_min_distance_vertex(const vector<int>& distances, const vector
    <bool>& visited) {
    int min_dist = 1e9;  // Start with large value
    int min_index = -1;
    for (int i = 0; i < distances.size(); ++i) {
        if (!visited[i] && distances[i] < min_dist) {
            min_dist = distances[i];
            min_index = i;
        }
    }
    return min_index;  // Return index of closest unvisited vertex
}
```

Stage 3: Output Generation

- **No Path Case**: If the distance to G remains infinite, output "No path from A to G".

- **Path Reconstruction**: Using the parent[] array, the shortest path from A to G is reconstructed in reverse and then reversed back.

```
// Reconstruct shortest path using parent array
    vector<string> path;
    for (int v = goal; v != -1; v = parent[v]) {
        path.push_back(g.index_to_vertex[v]);
    }
    reverse(path.begin(), path.end());
```

- **Output**: Both terminal and output file will contain:
    - The total travel time
    - The full path from A to G

```
// Output result to terminal
    cout << dist[goal] << "\n";
    for (int i = 0; i < path.size(); ++i) {
        cout << path[i];
        if (i < path.size() - 1) cout << " ";
    }
    cout << "\n";

    // Output result to file
    outFile << dist[goal] << "\n";
    for (int i = 0; i < path.size(); ++i) {
        outFile << path[i];
        if (i < path.size() - 1) outFile << " ";
    }
    outFile << "\n";
```

4. Testing and Evaluation

4.1. Sample Test Case Provided by Instructor

To verify the correctness of our implementation, we first tested the program using a sample input file provided by the instructor. The input file includes a graph with defined vertices, traffic light cycles, and bidirectional edges with specific travel times.

The test input file was named **tests.txt**, and the output was directed to **output_10422030.txt**. The expected output includes the total minimum time required to travel from vertex A to vertex G, along with the sequence of vertices on the shortest path.

Sample Input File (tests.txt)

~/Desktop/10422030/tests.txt - Mousepad

File Edit Search View Document Help

```
2
7
A, 2
B, 3
C, 5
D, 4
E, 6
F, 2
G, 1
A, B, 4
A, C, 2
B, D, 5
C, D, 8
C,E, 10
D, F, 6
E, F, 3
F, G, 1
7
A, 2
B, 3
C, 4
D, 2
E, 5
F, 3
G, 1
A, B, 2
A, C, 4
B, D, 3
C, E, 5
D, E, 1
D, F, 2
E, G, 3
F, G, 4
```

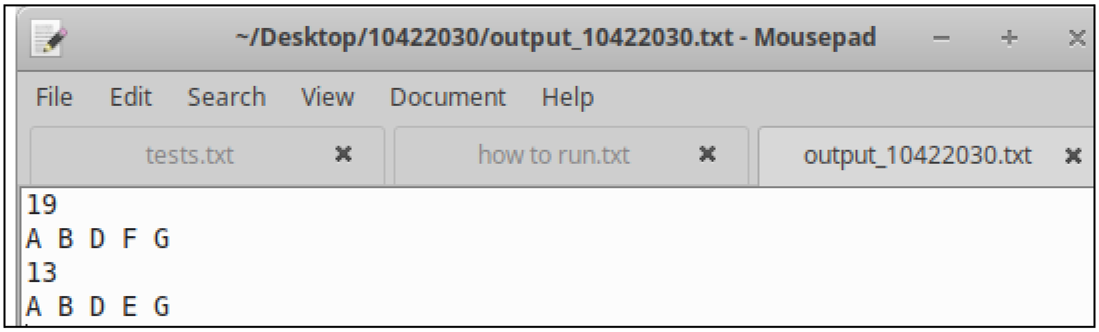
Terminal Output

Terminal - minhhoa@Compaq-510: ~/Desktop/10422030

File Edit View Terminal Tabs Help

```
minhhoa@Compaq-510:~/Desktop/10422030$ g++ 10422030.cpp -o 10422030
./10422030 tests.txt output_10422030.txt
19
A B D F G
13
A B D E G
minhhoa@Compaq-510:~/Desktop/10422030$
```

Generated Output File (output_10422030.txt)



This test case confirmed that the program correctly parses the input, applies traffic light constraints during pathfinding, and outputs the expected result.

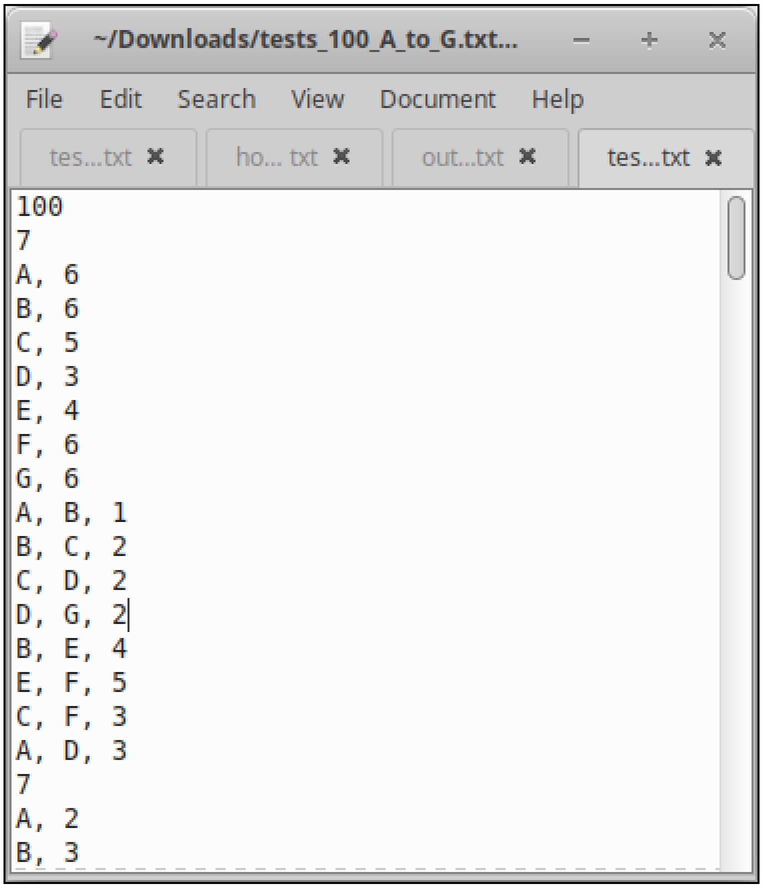
4.2. Stress Test with 100 Graphs

To evaluate the scalability and performance of the implementation, we conducted a stress test using a large input file containing **100 graphs**, each with various numbers of vertices, edges, and traffic light cycles.

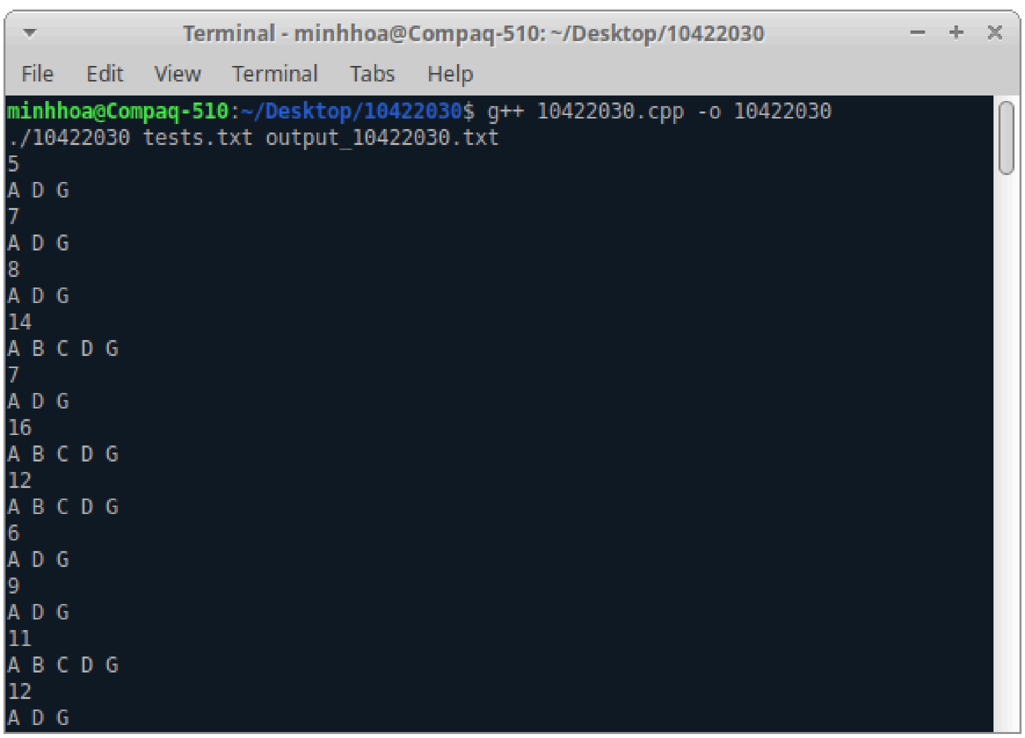
The goal was to ensure that:

- The program efficiently handles multiple consecutive graphs.
- The parsing logic accurately separates and processes each graph.
- The algorithm produces correct output without crashing or excessive delay.

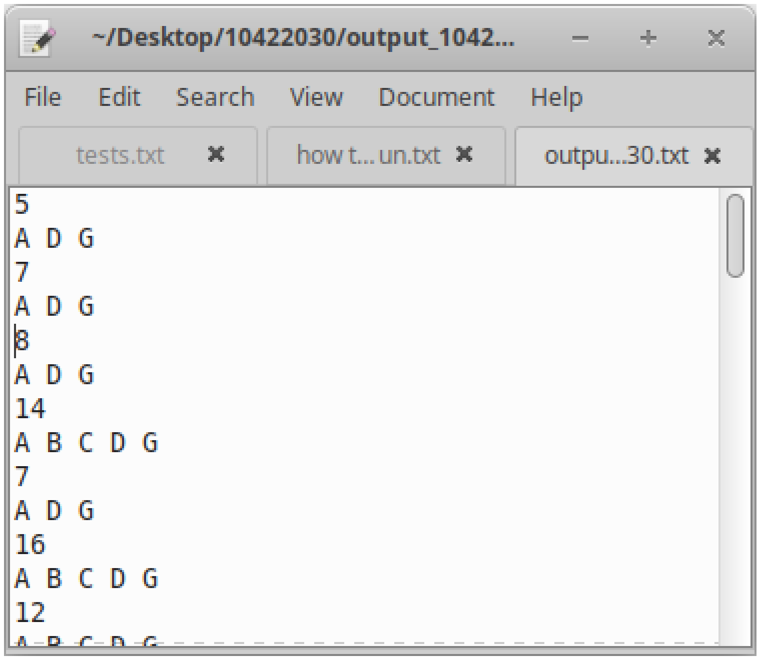
Sample Section of the 100-Graph Input File



Terminal Output (Summary View)



Output File (output_10422030.txt)




```
5
A D G
7
A D G
8
A D G
14
A B C D G
7
A D G
16
A B C D G
12
```

This test confirmed that the implementation is robust, modular, and capable of processing large-scale input efficiently. The results showed consistent accuracy and performance across all cases.

5. Challenges and Solutions

- **Parsing Multiple Graphs**: Careful detection of when a new graph starts was implemented by tracking tellg() and checking if the next line starts with a digit.
- **Traffic Light Waiting Logic**: A key challenge was computing wait times correctly to align with green-light timings.
- **Edge Case Handling**: We ensured the program handles graphs with no path between A and G.

6. Conclusion

In this lab, we successfully implemented a traffic-aware shortest path algorithm using a customized Dijkstra's algorithm. The program correctly parses multiple graphs, handles edge weights and vertex delays due to traffic lights, and outputs valid shortest paths.

The logic is modular, scalable, and adheres strictly to the lab requirements. All stages (input reading, algorithm execution, and output writing) are fully integrated.

7. Source Code

The full implementation is included in the file: 10422030.cpp