OptiWay

FINAL PROJECT DOCUMENT

Team Members

MICHAEL ZHANG 21445 HARDY WEN 21447

BEENO HUANG 21208 DAVID LIU 21036

Table of Contents

1 – Problem Statement

Due to an increasing number of students in SCIE, stairs and corridors are becoming increasingly crowded. We design a system that takes the all-students' timetables as input at the same time, and automatically calculate the best route to move between rooms for each student trying to minimize crowded areas and reduce average traffic time.

We are not trying to develop a system that only minimizes travel distance; we attempt to minimize the congestion-corrected travel time for all students at the same time. The initial route of any student is unknown, and optimization should take place for all 2000 students at the same time.

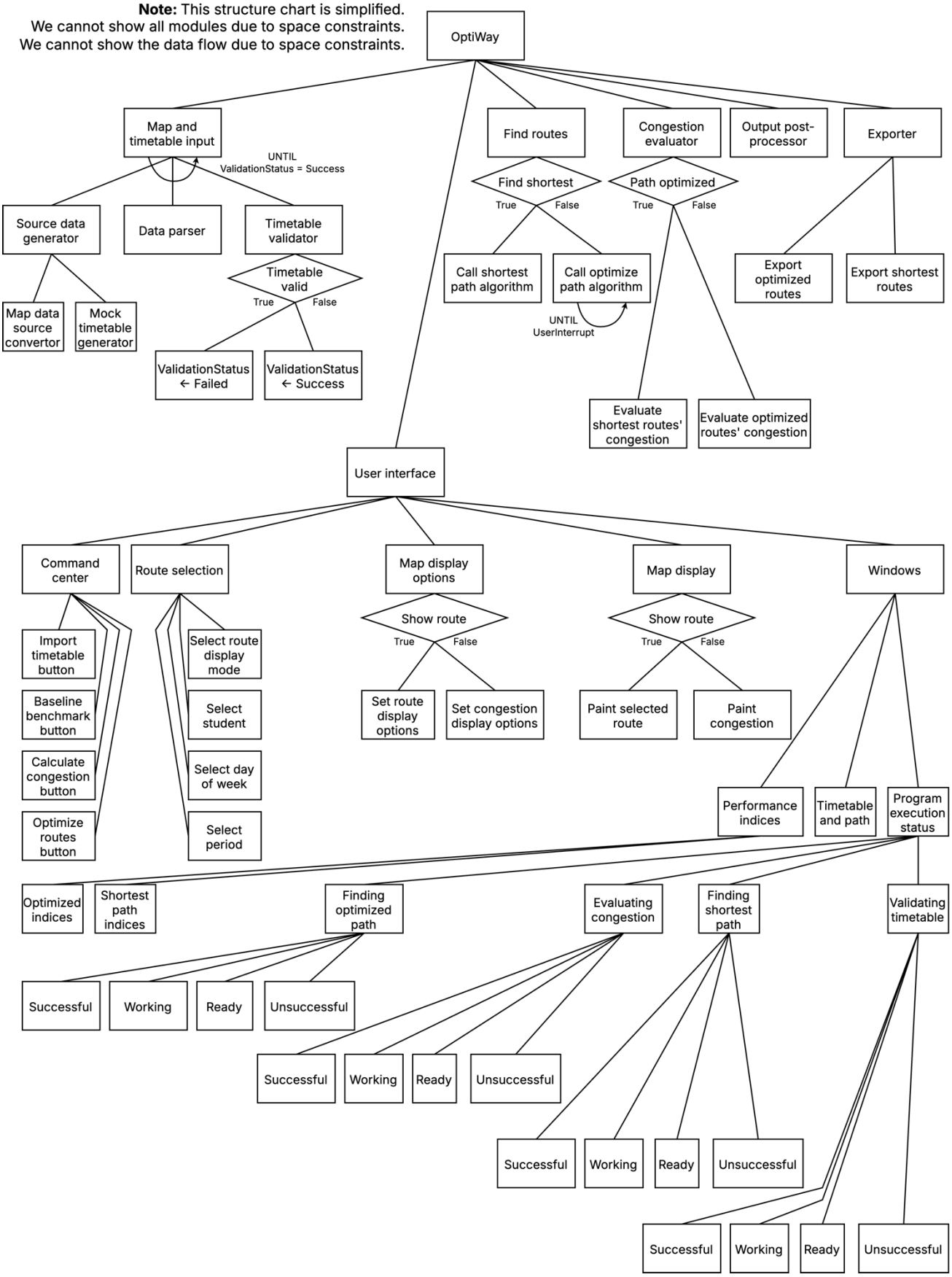
We will need to solve the following problems:

- How to obtain initial map data for the school teaching building;
- How to generate reasonable mock timetables for all students as test data input;
- How to find out an optimized plan for the entire school to minimize crowded areas;
- How to limit the execution time to an acceptable range;
- How to evaluate the effectiveness of reducing crowdedness using the optimized plan;
- How to store input and output data locally;
- How to receive user input in an efficient way;
- How to present the program execution progress in an intuitive way;
- How to present the program output in an intuitive way;
- How to handle erroneous inputs;
- How to manage multithreading properly to prevent freezing user interface;
- How to integrate different modules of the program.

2 – Program Structure Chart

The structure chart is displayed on the next page.

Note: This structure chart is simplified.
We cannot show all modules due to space constraints.
We cannot show the data flow due to space constraints.



3 – Algorithm Design

In this section, we display the pseudocode design three of the modules used in our program. Explanation is included at the start of each subsection.

3.1 – School's Graph Construction

In order to optimize the students' paths within a school, we propose the construction of an undirected, weighted graph $G = (N, E)$. Here, $N = \{n_i \mid i \in \{1,2,\dots,m\}\}$ denotes the set of nodes, with each node n_i representing a significant location within the school premises, such as a classroom, a staircase, or a terminal point on the bridge connecting buildings A and B. The parameter m signifies the total quantity of these pivotal locations at SCIE. This data structure helps us to run existing path-finding algorithms and make customizations on them easily.

The set of edges $E = \{e_j \mid j \in \{1,2,\dots,m-1\}\}$ encapsulates the relationships between pairs of nodes, indicating a direct connection between the locations they represent. Furthermore, each edge e_j is assigned a weight, which quantitatively represents the distance between the two nodes it connects.

The relational and distance data among all nodes has been precomputed and stored within a text file. The subsequent task entails parsing this text file to retrieve the data, and subsequently populating the aforementioned graph data structure within our computational model. The pseudocode provided below delineates this procedure in detail.

PSEUDOCODE

```
CONSTANT MaxNodes = 1000 // Max number of rooms in the school

// An edge on the graph, representing the relationship between rooms
TYPE Edge
    DECLARE Dest: STRING
    DECLARE Weight: INTEGER  // The distance between the rooms
    DECLARE EdgeType: INTEGER
ENDTYPE

 // A node on the graph, representing a room in the school
TYPE Room
    DECLARE Edges: ARRAY[1:MaxNodes] OF Edge
    DECLARE EdgeCount: INTEGER
ENDTYPE

FUNCTION CreateSchoolGraph(FilePath: STRING) RETURNS ARRAY[1:MaxNodes] OF Room
    OPENFILE FilePath FOR READ
    DECLARE Line: STRING
    DECLARE SchoolGraph: ARRAY[1:MaxNodes] OF Room

    // Initialize the count of edges
    FOR X ← 1 TO MaxNodes
        SchoolGraph[X].EdgeCount ← 0
    NEXT X

    WHILE NOT EOF(FilePath)
        READFILE FilePath, Line
        // Parse the line
        DECLARE SeperatedIndex: ARRAY[1:4] OF INTEGER
        DECLARE SeperatedIndexCount: INTEGER
        SeperatedIndexCount ← 1
        FOR X ← 1 TO LENGTH(Line)
            IF Line[X] = ' '
              THEN
                SeperatedIndex[SeperatedIndexCount] ← X
                SeperatedIndexCount ← SeperatedIndexCount + 1
            ENDIF
        NEXT X
```

```
        // Extract the data from the line
        DECLARE Room1: INTEGER  // Assume that rooms are stored as integers for simplicity
        DECLARE Room2: INTEGER
        DECLARE Dist: INTEGER
        DECLARE EdgeType: INTEGER
        Room1 ← STR_TO_NUM(MID(Line, 1, SeperatedIndex[1] - 1))
        Room2 ←
          STR_TO_NUM(MID(Line, SeperatedIndex[1] + 1, SeperatedIndex[2] - SeperatedIndex[1] - 1))
        Dist ←
          STR_TO_NUM(MID(Line, SeperatedIndex[2] + 1, SeperatedIndex[3] - SeperatedIndex[2] - 1))
        EdgeType ←
          STR_TO_NUM(MID(Line, SeperatedIndex[3] + 1, SeperatedIndex[4] - SeperatedIndex[3] - 1))
        // Add data to the graph
        SchoolGraph[Room1].EdgeCount ← SchoolGraph[Room1].EdgeCount + 1
        SchoolGraph[Room1].Edges[SchoolGraph[Room1].EdgeCount].Dest ← Room2
        SchoolGraph[Room1].Edges[SchoolGraph[Room1].EdgeCount].Weight ← Dist
        SchoolGraph[Room1].Edges[SchoolGraph[Room1].EdgeCount].EdgeType ← EdgeType
        SchoolGraph[Room2].EdgeCount ← SchoolGraph[Room2].EdgeCount + 1
        SchoolGraph[Room2].Edges[SchoolGraph[Room2].EdgeCount].Dest ← Room1
        SchoolGraph[Room2].Edges[SchoolGraph[Room2].EdgeCount].Weight ← Dist
        SchoolGraph[Room2].Edges[SchoolGraph[Room2].EdgeCount].EdgeType ← EdgeType
    ENDWHILE

    CLOSEFILE FilePath
    RETURN SchoolGraph
ENDFUNCTION
```

3.2 – Floyd-Warshall's Algorithm

Floyd-Warshall's algorithm is utilized to generate the shortest paths from each room to another. These paths act as baselines for students' paths, as they do not take into account congestion. We pre-calculate the shortest path of every room pair and save them into a JSON file in reality. At runtime, only data retrieval is required, improving the efficiency of our approach.

The algorithm works as below:

1. **Initialization**: The algorithm starts by creating a distance matrix that represents the distances between all pairs of vertices. If there is a direct edge between two vertices, the distance is set to the weight of that edge. If there is no direct edge, the distance is set to infinity (or a very large number). The distance from each vertex to itself is set to 0.
2. **Iterative Updates**: The algorithm then performs a series of iterations to update the distance matrix. In each iteration, it considers all possible pairs of vertices (i, j) and tries to find a vertex k such that the path from i to j through k is shorter than the current known shortest path from i to j. If such a k is found, the distance matrix is updated.

The pseudocode for the calculation of Floyd-Warshall shortest paths are shown in the pseudocode below. We don't include JSON handling in the pseudocode due to the syntax limitations; it provides insights into the principle of the algorithm, however.

PSEUDOCODE

```
CONSTANT INFINITY = 2147483647 // Correspond to INT_MAX
// The floyd distance between rooms 1 and j
DECLARE Distance: ARRAY[1:MaxNodes, 1:MaxNodes] OF INTEGER
// The predecessor of room j on the shortest path from room 1 to j
DECLARE Predecessor: ARRAY[1:MaxNodes, 1:MaxNodes] OF INTEGER

PROCEDURE InitializeDistanceAndPredecessor(BYREF Distance: ARRAY[1:MaxNodes, 1:MaxNodes] OF INTEGER,
  BYREF Predecessor: ARRAY[1:MaxNodes, 1:MaxNodes] OF INTEGER)
    // Initialize the distance and predecessor matrices
    FOR X ← 1 TO MaxNodes
        FOR Y ← 1 TO MaxNodes
            IF X = Y
              THEN
                Distance[X, Y] ← 0
```

```
                Predecessor[X, Y] ← 0
              ELSE
                Distance[X, Y] ← INFINITY
                Predecessor[X, Y] ← -1
            ENDIF
        NEXT Y
    NEXT X

    // Initialize the distance and predecessor matrices with the graph
    FOR X ← 1 TO MaxNodes
        FOR Y ← 1 TO Graph[X].EdgeCount
            Distance[X, Graph[X].Edges[Y].Dest] ← Graph[X].Edges[Y].Weight
            Predecessor[X, Graph[X].Edges[Y].Dest] ← X
        NEXT Y
    NEXT X
ENDPROCEDURE


PROCEDURE GetFloydWarshall(BYREF Distance: ARRAY[1:MaxNodes, 1:MaxNodes] OF INTEGER,
  BYREF Predecessor: ARRAY[1:MaxNodes, 1:MaxNodes] OF INTEGER, Graph: ARRAY[1:MaxNodes] OF Room)
    // Initialize the distance and predecessor matrices
    CALL InitializeDistanceAndPredecessor(Distance, Predecessor)

    // Floyd-Warshall
    FOR K ← 1 TO MaxNodes
        FOR I ← 1 TO MaxNodes
            FOR J ← 1 TO MaxNodes
                IF Distance[I, J] > Distance[I, K] + Distance[K, J] AND Distance[I, K] <> INFINITY
                AND Distance[K, J] <> INFINITY
                  THEN
                    Distance[I, J] ← Distance[I, K] + Distance[K, J]
                    Predecessor[I, J] ← Predecessor[K, J]
                ENDIF
            NEXT J
        NEXT I
    NEXT K
ENDPROCEDURE
```

3.3 – Customized Multi-Objective Path Optimization Algorithm

We have developed a customized algorithm for optimizing the shortest paths based on multiple objectives, i.e., congestion and distance, based on the performance index defined below. Congestion often leads to lateness and complaints, extending the time to go through the path despite their shortness in distances. By taking congestion into account, this algorithm aims to mitigate this issue.

Objective

The algorithm aims to achieve a minimum value of $\sum r_{\text{perf}}$ considering all the students' paths.

For each route r for a student between two specific periods, we give the route a performance index r_{perf}, where a smaller performance index indicates better performance. Consider all the paths between nodes p_i $(i \in \{1,2,3,\dots,n\})$ in the route r, let the length of the path be w_i, and the number of students passing the path between the two periods (congestion) be c_i.

$$r_{\text{perf}} = \sum_{i=1}^{n} \left[w_i \cdot \left(2 + \frac{e^{(c_i-300)/200} - e^{-(c_i-300)/200}}{e^{(c_i-300)/200} + e^{-(c_i-300)/200}} \right) \right]$$

The algorithm works as below:

Initializations

1. Calculate congestion c_i for each edge
2. Assign precalculated shortest path to each student

3. Calculate the performance r_{perf} for each student's path, and sum to get an initial total $\sum r_{\text{perf}}$
4. Store each student, their relative path, and the path's r_{perf} score as a struct, and push the struct into a priority queue ordered by r_{perf} decreasingly.

Heuristic Iterations

1. **For each iteration**, choose the student whose path has the greatest r_{perf}
2. Recalculate the path of the student using **Dijkstra's algorithm**, by including a **congestion weight** w_c
   - For each edge $e_i \in E$, we update it as $w_{e_i} \coloneqq w_{e_i} + w_c \times c_i$
   - Calculate an individual r_{perf} value for the updated path
     - Note that c_i is not updated here for efficiency, as a minor change in c_i has minor changes to $\sum r_{\text{perf}}$
3. **For each `batch_size` iterations**, we recalculated c_i at each edge, and deduce a new $\sum r_{\text{perf}}$. If the new $\sum r_{\text{perf}}$ is better (less) than that of the last batch, we update the paths.
4. The pseudocode for the path optimization algorithm is shown as below. Instead of using priority queues, we use arrays in the code; we make slight modifications for the data structure as well. These changes are pruned to syntax limitations.

PSEUDOCODE

```
// Part 1: The procedure for finding the shortest path considering the congestion,
//         using dijkstra's algorithm
CONSTANT INFINITY = 2147483647  // Correspond to INT_MAX
CONSTANT CONGESTION_PENALTY = 1000  // A user-defined value

FUNCTION GetPenaltiedDijsktra(Start: STRING, End: STRING, Graph: ARRAY[1:MaxNodes] OF Room,
  Congestion: ARRAY[1:MaxNodes][1:MaxNodes] OF INTEGER) RETURNS ARRAY[1:MaxNodes] OF INTEGER
  // returns the path
    // The dijsktra distance between rooms 1 and j
    DECLARE Distance: ARRAY[1:MaxNodes, 1:MaxNodes] OF INTEGER
    // The predecessor of room j on the shortest path from room 1 to j
    DECLARE Predecessor: ARRAY[1:MaxNodes, 1:MaxNodes] OF INTEGER
    DECLARE Visited: ARRAY[1:MaxNodes] OF BOOLEAN
    DECLARE Path: ARRAY[1:MaxNodes] OF INTEGER
    DECLARE PathCount: INTEGER
    PathCount ← 0

    // Initialize the distance and predecessor matrices
    CALL InitializeDistanceAndPredecessor(Distance, Predecessor)

    // Dijsktra
    DECLARE Current: INTEGER
    Current ← Start
    Visited[Current] ← TRUE
    WHILE Current <> End
        FOR X ← 1 TO Graph[Current].EdgeCount
            DECLARE PenaltiedEdgeWeight: INTEGER
            PenaltiedEdgeWeight ← Graph[Current].Edges[X].Weight +
              (CONGESTION_PENALTY * Congestion[Current][Graph[Current].Edges[X].Dest])
            IF NOT Visited[Graph[Current].Edges[X].Dest]
              AND Distance[Start, Graph[Current].Edges[X].Dest] > PenaltiedEdgeWeight
              THEN
                Distance[Start, Graph[Current].Edges[X].Dest] ←
                  Distance[Start, Current] + Graph[Current].Edges[X].Weight
                Predecessor[Start, Graph[Current].Edges[X].Dest] ← Current
            ENDIF
        NEXT X

        // Find the next room to visit
        DECLARE MinDistance: INTEGER
```

```
        MinDistance ← INFINITY
        FOR X ← 1 TO MaxNodes
            IF NOT Visited[X] AND Distance[Start, X] < MinDistance
              THEN
                MinDistance ← Distance[Start, X]
                Current ← X
            ENDIF
        NEXT X
        Visited[Current] ← TRUE
    ENDWHILE

    // Find the path
    Current ← End
    WHILE Current <> Start
        PathCount ← PathCount + 1
        Path[PathCount] ← Current
        Current ← Predecessor[Start, Current]
    ENDWHILE

    RETURN Path
ENDFUNCTION

// Part 2: Calculate the performance index of a specific path
CONSTANT e = 2.71828182845904523536028747135 27  // The mathematical constant e
FUNCTION ComputePowersOfE(Power: REAL) RETURNS REAL
    DECLARE Result: REAL
    Result ← 1
    FOR X ← 1 TO Power
        Result ← Result * e
    NEXT X
    RETURN Result
ENDFUNCTION

FUNCTION ComputePerformanceIndex(Path: ARRAY[1:MaxNodes] OF INTEGER,
  Graph: ARRAY[1:MaxNodes] OF Room,
  Congestion: ARRAY[1:MaxNodes][1:MaxNodes] OF INTEGER) RETURNS REAL
    DECLARE Rperf: REAL
    DECLARE Wi: INTEGER
    DECLARE Ci: INTEGER
    DECLARE Start: INTEGER
    DECLARE End: INTEGER
    Rperf ← 0

    FOR X ← 1 TO LENGTH(Path)
        Start ← Path[X]
        End ← Path[X + 1]
        FOR Y ← 1 TO Graph[Start].EdgeCount
            IF Graph[Start].Edges[Y].Dest = END
              THEN
                Wi ← Graph[Start].Edges[Y].Weight
            ENDIF
        NEXT Y

        Ci ← Congestion[Start][End]
        Rperf ← Rperf + Wi * (2 + ComputePowersOfE((Ci - 300.0) / 200.0)
          - ComputePowersOfE(-(Ci - 300.0) / 200.0))
          / (ComputePowersOfE((Ci - 300.0) / 200.0)
          + ComputePowersOfE(-(Ci - 300.0) / 200.0))  // Rperf is a Tanh function
    NEXT X

    RETURN Rperf
ENDFUNCTION

// Part 3: A single iteration of the program
TYPE StudentPath
    DECLARE Rperf: REAL  // Performance index
    DECLARE Path: ARRAY[1:MaxNodes] OF INTEGER
ENDTYPE
```

```
DECLARE Paths: ARRAY[1:MaxNodes] OF StudentPath
DECLARE SumRperf: REAL

PROCEDURE Iter(BYREF Paths: ARRAY[1:MaxNodes] OF StudentPath, BYREF SumRperf: REAL,
  BYREF LastStart: INTEGER, BYREF LastEnd: INTEGER,
  Congestion: ARRAY[1:MaxNodes, 1:MaxNodes] OF INTEGER, Graph: ARRAY[1:MaxNodes] OF Room)
    // Find the path with the highest rperf value to optimize
    DECLARE WorstPath: StudentPath
    DECLARE WorstIndex: INTEGER
    WorstPath.Rperf ← 0
    FOR X ← 1 TO LENGTH(Paths)
        // Last start and last end are used to track the path cannot be optimized anymore
        IF Paths[X].Rperf > WorstPath.Rperf AND Paths[X].Path[1] <> LastStart
          AND Paths[X].Path[LENGTH(Paths[X].Path)] <> LastEnd
          THEN
            WorstPath ← Paths[X]
            WorstIndex ← X
        ENDIF
    NEXT X
    // Optimize the path
    SumRperf ← SumRperf - WorstPath.Rperf
    DECLARE NewPath: ARRAY[1:MaxNodes] OF INTEGER
    DECLARE NewRperf: REAL
    NewPath ← GetPenaltiedDijsktra(WorstPath.Path[1],
      WorstPath.Path[LENGTH(WorstPath.Path)], Graph, Congestion)
    NewRperf ← ComputePerformanceIndex(NewPath, Graph, Congestion)
    // Decide whether to update the path
    IF NewRperf < WorstPath.Rperf
      THEN
        WorstPath.Path ← NewPath
        WorstPath.Rperf ← NewRperf
        SumRperf ← SumRperf + NewRperf
        Paths[WorstIndex] ← WorstPath  // Update the path to the paths array
      ELSE
        // This path cannot be optimized anymore, so keep track of it
        LastStart ← WorstPath.Path[1]
        LastEnd ← WorstPath.Path[LENGTH(WorstPath.Path)]
        SumRperf ← SumRperf + WorstPath.Rperf
    ENDIF
ENDPROCEDURE

// Part 4: Optimization for a given day
TYPE PeriodData
    // The congestion between rooms i and j
    DECLARE Congestion: ARRAY[1:MaxNodes][1:MaxNodes] OF INTEGER
    DECLARE SumRperf: REAL
    DECLARE PrevSumRperf: REAL
    DECLARE LastStart: INTEGER
    DECLARE LastEnd: INTEGER
ENDTYPE

PROCEDURE IterSingleDay(Day: INTEGER, BYREF PathsPeriods: ARRAY[0:11][1:MaxNodes] OF StudentPath,
  Graph: ARRAY[1:MaxNodes] OF Room)
    // PathsPeriods stores each student's shortest path at each period by floyd-warshall,
    // with a precalculated sum of rperfs.
    // We achieve this using a JSON file in our C++ program
    DECLARE PeriodsData: ARRAY[0:11] OF PeriodData
    DECLARE PathsPeriodsCopy: ARRAY[0:11][1:MaxNodes] OF StudentPath
    // Keep track of the original path before this batch
    // Initialize the congestion matrix
    FOR Period ← 0 TO 11
        PathsPeriodsCopy[Period] ← PathsPeriods[Period]
        FOR X ← 1 TO MaxNodes
            FOR Y ← 1 TO MaxNodes
                PeriodsData[Period].Congestion[X][Y] ← 0
            NEXT Y
        NEXT X
    NEXT Period
    // Calculate the congestion matrix
```

```
FOR Period ← 0 TO 11
    FOR X ← 1 TO LENGTH(PathsPeriods[Period])  // For every student at this period
        FOR Y ← 1 TO LENGTH(PathsPeriods[Period][X].Path) - 1
            // Undirected graph: update congestion for both directions
            PeriodsData[Period].Congestion[PathsPeriods[Period][X].Path[Y]]
              [PathsPeriods[Period][X].Path[Y + 1]] ←
              PeriodsData[Period].Congestion[PathsPeriods[Period][X].Path[Y]]
              [PathsPeriods[Period][X].Path[Y + 1]] + 1
            PeriodsData[Period].Congestion[PathsPeriods[Period][X].Path[Y + 1]]
              [PathsPeriods[Period][X].Path[Y]] ←
              PeriodsData[Period].Congestion[PathsPeriods[Period][X].Path[Y + 1]]
              [PathsPeriods[Period][X].Path[Y]] + 1
        NEXT Y
    NEXT X
NEXT Period
// Calculate the sum of rperfs
FOR Period ← 0 TO 11
    PeriodsData[Period].SumRperf ← 0
    FOR X ← 1 TO LENGTH(PathsPeriods[Period])
        PeriodsData[Period].SumRperf ←
          PeriodsData[Period].SumRperf + PathsPeriods[Period][X].Rperf
        PeriodsData[Period].PrevSumRperf ← PeriodsData[Period].SumRperf
    NEXT X
NEXT Period
// Optimization
// The number of iterations done before, a user-defined value
DECLARE ITERCOUNT: INTEGER
// The number of iterations to do in total, set as infinity in practice
DECLARE ITERNUM: INTEGER
// The number of iterations before updating the congestion matrix, a user-defined value
DECLARE BATCHSIZE: INTEGER
ITERCOUNT ← 0
ITERNUM ← INFINITY
BATCHSIZE ← 256
FOR Step ← IterCount TO IterNum
    FOR Period ← 0 TO 11
        CALL Iter(PathsPeriods[Period], PeriodsData[Period].SumRperf,
          PeriodsData[Period].LastStart, PeriodsData[Period].LastEnd,
          PeriodsData[Period].Congestion, Graph)
        IF MOD(Step, BATCHSIZE) <> 0
          THEN
            // Following the output format
            OUTPUT "0 " & NUM_TO_STR(Step) & " " & NUM_TO_STR(Day) & " " & NUM_TO_STR(Period)
              & " " & NUM_TO_STR(PeriodsData[Period].SumRperf) & " " &
              NUM_TO_STR(PeriodsData[Period].PrevSumRperf)
          ELSE
            // Update the congestion matrix
            FOR X ← 1 TO MaxNodes
                FOR Y ← 1 TO MaxNodes
                    PeriodsData[Period].Congestion[X][Y] ← 0
                NEXT Y
            NEXT X
            FOR X ← 1 TO LENGTH(PathsPeriods[Period])
                FOR Y ← 1 TO LENGTH(PathsPeriods[Period][X].Path) - 1
                    PeriodsData[Period].Congestion[PathsPeriods[Period][X].Path[Y]]
                      [PathsPeriods[Period][X].Path[Y + 1]] ←
                      PeriodsData[Period].Congestion[PathsPeriods[Period][X].Path[Y]]
                      [PathsPeriods[Period][X].Path[Y + 1]] + 1
                    PeriodsData[Period].Congestion[PathsPeriods[Period][X].Path[Y + 1]]
                      [PathsPeriods[Period][X].Path[Y]] ←
                      PeriodsData[Period].Congestion[PathsPeriods[Period][X].Path[Y + 1]]
                      [PathsPeriods[Period][X].Path[Y]] + 1
                NEXT Y
            NEXT X
            // Update the sum of rperfs
            PeriodsData[Period].SumRperf ← 0
            FOR X ← 1 TO LENGTH(PathsPeriods[Period])
                PathsPeriods[Period][X].Rperf ←
                  ComputePerformanceIndex(PathsPeriods[Period][X].Path, Graph,
```

```
                    PeriodsData[Period].Congestion)
                  PeriodsData[Period].SumRperf ←
                    PeriodsData[Period].SumRperf + PathsPeriods[Period][X].Rperf
                NEXT X
                // Check whether to update the path
                IF PeriodsData[Period].SumRperf < PeriodsData[Period].PrevSumRperf
                  THEN
                    PeriodsData[Period].PrevSumRperf ← PeriodsData[Period].SumRperf
                    PathsPeriodsCopy[Period] ← PathsPeriods[Period]
                  ELSE
                    PeriodsData[Period].SumRperf ← PeriodsData[Period].PrevSumRperf
                    PathsPeriods[Period] ← PathsPeriodsCopy[Period]
                ENDIF
                // Following the output format
                OUTPUT "1 " & NUM_TO_STR(Step) & " " & NUM_TO_STR(Day) & " " & NUM_TO_STR(Period)
                  & " " & NUM_TO_STR(PeriodsData[Period].SumRperf) & " " &
                  NUM_TO_STR(PeriodsData[Period].PrevSumRperf)
            ENDIF
        NEXT Period
    NEXT Step
ENDPROCEDURE
```

4 – Module Description

OptiWay contains a large number of modules. The first-level modules are listed as follows:

1. **Map data source converter** (Python). This module converts the school teaching building map data source into a machine-readable format, and makes any adjustments to handle extreme data where necessary. The data source file types include JSON, YAML and plain text. This module provides the functionality to convert between different file types.
2. **Mock timetable generator** (Python). This module generates mock timetables for all students in SCIE in a reasonable way, using known rules for course arrangements in the school. These mock timetables will be used as test data to test the program.
3. **Data parser** (C++). This module parses the data input (mainly JSON) and returns a C++ data structure that is directly usable in the code.
4. **Data parser** (Rust). This module parses the data input (mainly JSON) and returns a Rust data structure that is directly usable in the code.
5. **Shortest path algorithm** (C++). This module generates the shortest paths between different pairs in SCIE, and is therefore used as a baseline algorithm to test the effectiveness of our optimization algorithm.
6. **Route optimization algorithm** (C++). This module optimizes the previously generates shortest routes for all students to reduce the congestion, and hence the average travelling time between each period for all students at the same time.
7. **Timetable validator** (Rust). This module validates the timetable input, and reports an error message if the input is malformed.
8. **Cross-language caller** (Rust). This module calls C++ programs from Rust, in order to integrate different programming languages used by our teammates.
9. **Congestion evaluator** (Rust). This module evaluates the overall congestion in a given route plan, and stores the result into a complex data structure. It also returns a performance index, used to measure the overall quality of the plan.
10. **User interface** (Rust). This module accepts user input, display program execution status, and show program results in a user-friendly way.
11. **Output post-processor** (Rust). This module integrates useful information from multiple program

outputs, and stores all meaningful outputs in a complex data structure.

12. **Exporter** (Rust). This module exports program output as local files for future offline usage.

The respective submodules are not listed here due to the number of modules included in the program. Important submodules are included in the structure chart.

5 – User Interface

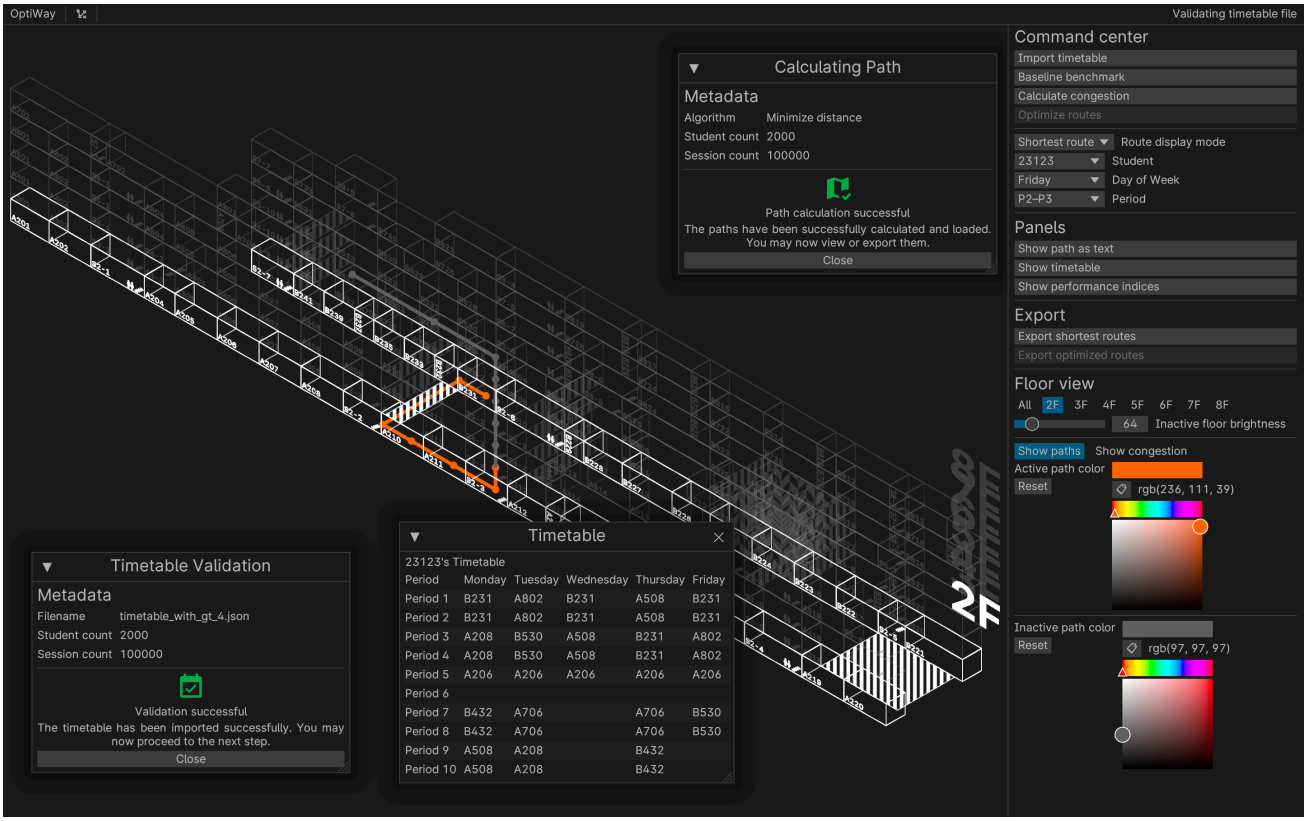
This section discusses the design of user interface of OptiWay.

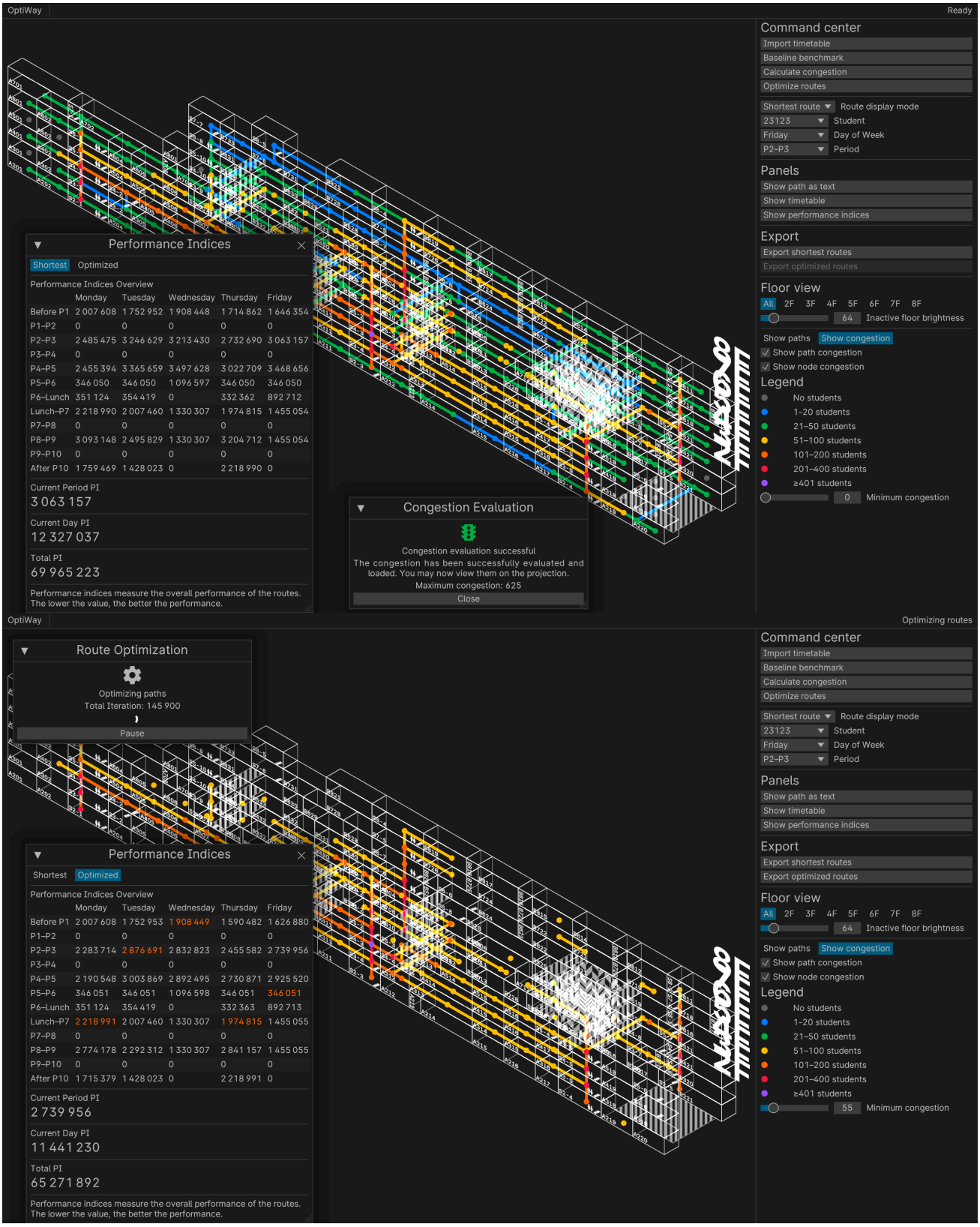
5.1 – Introduction

Considering the fact that OptiWay generates routes for the entire school, it is expected that this program should only be used by the school IT department who have the real student timetable data. Therefore, this program targets expert users and uses a more efficient approach to maximize the amount and clarity of the information presented on screen, rather than trying to be very friendly for new users. Nevertheless, the user interface is still intuitive for those with a minimal amount of training.

Some screenshots with explanation of the user interface are shown in the following subsection. See the supplementary video for a full demonstration of the user interface.

5.2 – Screenshots





6 – Coding & Functionality

The code is displayed in a separate 100-page document. The code for each language follows their own style guide, using

- **autopep8** for Python,
- **clang-format** for C++,
- **rustfmt** for Rust.

The naming conventions are consistent within a single language. All common unexpected inputs are handled.

The code is also accessible at https://github.com/micfong-z/OptiWay. Be aware that this program has over 5000 lines of code (with tens of thousands of lines of dependencies), so it may not be too easy to navigate within the repository.

See a separate supplementary video for functionality demonstration.

7 – Division of Task

We have tried to divide the task as evenly as possible among our teammates.

Member	Tasks
Michael	Timetable validation, user interface, cross-language integration, program entry point, congestion evaluation, output post-processor, map asset manual recording.
Hardy	Shortest path (Floyd-Warshall's) algorithm, customized multi-objective path optimization algorithm, congestion evaluation, output post-processor.
Beeno	Map data source converter, map asset manual recording, project idea contribution, initial project proposal writing, overall project structure.
David	Mock timetable generator, output data exporter, map data source converter, general algorithm optimization, final project document checking.

8 – Complexity & Usefulness

In this section we will discuss the complexity and usefulness of OptiWay.

8.1 – Complexity

This program utilizes multiple programing languages and ideologies, with extensive use of subroutines, file handling, multithreading, arrays, enumerated types, complex data structures (classes), iteration, recursion and much more.

Moreover, considering the fact that we have around 100 pages of code and tens of modules, with multiple files arranged in multiple directories and the need for a build script to compile our program, it is proven that our program reaches an extremely high level of complexity.

8.2 – Usefulness

Currently, the program is completely useful as it fully accomplishes all problems stated in Section 1, where simply giving the program timetables from CMS, the program is able to find the route requiring minimal travel time and avoid congestion.

Furthermore, the program compiles on multiple platforms, such as macOS, Windows, Linux and

potentially (although not tested yet) WASM, Android and iOS, OptiWay is useful whatever the underlying operating system is.

References

No references are made directly in this project. However, a number of pre-written modules (dependencies) are used, including:

1. **JSON for Modern C++** by Niels Lohmann. Used for JSON file handling in C++.
2. **egui** by Emil Ernerfeldt. Used as the GUI library in Rust.
3. **Serde** by the community. Used as the serialization framework for data structures in Rust.
    a) **Serde YAML** by the community. Used for YAML file handling in Rust.
    b) **Serde JSON** by the community. Used for JSON file handling in Rust.
4. **num-format** by Brain Myers. Used for number formatting in Rust.
5. **PyYAML** by Kirill Simonov. Used for YAML file handling in Python.

See the project source code for the specific versions used to build this project.

END OF DOCUMENT