Project directory structure

```
OptiWay/
assets/                              assets (YAML, SVG, etc.) used in the program
    └ (24 files and directories)
multi_agent_path_finding/            C++ pathfinding algorithms
    ├ floyd.cpp                      shortest path algorithm
    ├ json.cpp                       (dependency) JSON parsing library
    └ multi-objective-agent.cpp      path optimization algorithm
optiway/                             program entry point; Rust project root folder
    ├ assets/                        assets (YAML, SVG, etc.) used in the program
    │  └ (3 files and directories)
    ├ bin/                           final binaries produced and final output
    │  └ (8 files)
    ├ src/                           main source code
    │  ├ md_icons/                   (dependency) icon packs
    │  │  ├ material_design_icons.rs
    │  │  ├ material_symbols.rs
    │  │  └ mod.rs
    │  ├ app_init.rs                 application initialization configurations
    │  ├ app.rs                      main program and UI
    │  ├ lib.rs                      project structure configuration
    │  └ main.rs                     program entry point
    ├ target/                        intermediate files during compilation
    │  └ (100+ files and directories)
    ├ bundle/                        final output bundle
    │  └ (3 files and directories)
    ├ Cargo.lock                     auto-generated dependency configuration
    ├ build.rs                       Rust project build script
    ├ Cargo.toml                     Rust project configuration
    ├ Makefile.toml                  compilation rules and scripts
    └ rustfmt.toml                   Rust code formatter configuration
scripts/                             Python scripts to aid development
    ├ coord-flatten.py               flattens a specific YAML file
    ├ coord-generator.py             generates x-coordinates of specific nodes
    ├ crossing-paths.py              crossing paths generation helper
    ├ path-converter.py              converts YAML path data to TXT
    └ path-generator.py              generates paths on the same floor on one block
timetable_generation                 Python script to generate test data
    ├ return-structure.txt           notes on return structure
    └ timetable_generator.py         Python script to generate test data
.gitignore                           Git ignored files configuration
TECHNICAL_NOTES.md                   technical notes used in development
```

It should be noted that only files that are one of the following formats are included in this document, as these are the only code files:

- C++
- Rust
- TOML
- Python

Only underlined files are included as they are the only code-related files.

The source code can be found at https://github.com/micfong-z/OptiWay/ (limited access before project due date).

The project is fully tested using the following system environment:

Operating System

macOS Sonoma 14.0

Rust

Compiler: rustc 1.75.0-nightly (2e5a9dd6c 2023-10-02)
Toolchain: nightly-aarch64-apple-darwin
Cargo components:
    cargo 1.75.0-nightly (59596f0f3 2023-09-29)
    cargo-make 0.36.12
    clippy 0.1.74 (2e5a9dd6 2023-10-02)
Formatter: rustfmt 1.6.0-nightly (2e5a9dd6 2023-10-02)

C++

Compiler: Apple clang version 15.0.0 (clang-1500.0.40.1)
Standard: C++17
Formatter: clang-format

Python

Version: Python 3.11.5
Formatter: autopep8 2.0.2

Note that is it not possible to compile this program only with the code included in this document due to missing assets and dependencies. Due to the extremely large file sizes, they are not shown in this document. For a complete source code repository, see https://github.com/micfong-z/OptiWay/ (limited access before project due date).

multi_agent_path_finding/floyd.cpp

```cpp
#include <fstream>
#include <iostream>
#include <limits>
#include <sstream>
#include <string>
#include <unordered_map>
#include <vector>

#include "json.hpp"
using json = nlohmann::json;

using namespace std;

struct Edge {
    string dest;  // destination node
    int weight;   // edge weight (in this case, distance)
    int type;     // edge type
};

using Graph = unordered_map<string, vector<Edge> >;
using DistanceMatrix = unordered_map<string, unordered_map<string, int> >;
using PredecessorMatrix = unordered_map<string, unordered_map<string, string> >;

Graph createSchoolGraph(const string& file_path) {
    /* Construct the school's layout graph from the path.txt */
    Graph graph;
    ifstream file(file_path);

    if (!file.is_open()) {
        cerr << "Failed to open the file." << endl;
        return graph;
    }

    string line;
    while (getline(file, line)) {
        istringstream iss(line);

        string node1, node2;
        int distance, type;

        iss >> node1 >> node2 >> distance >> type;

        Edge edge1 = {node2, distance, type};
        Edge edge2 = {node1, distance, type};

        graph[node1].push_back(edge1);
        graph[node2].push_back(edge2);  // Because it's an undirected graph
    }

    file.close();
    return graph;
}

void initializeDistanceAndPredecessorMatrices(const Graph& graph, DistanceMatrix&
dist,
                                              PredecessorMatrix& pred) {
```

```cpp
    /* Initialize the dist and predecessor's matrices for floyd */
    const int INF = numeric_limits<int>::max();

    for (const auto& [node, _] : graph) {
        for (const auto& [node2, __] : graph) {
            if (node == node2) {
                dist[node][node2] = 0;
            } else {
                dist[node][node2] = INF;
            }
            pred[node][node2] = "";
        }
    }

    for (const auto& [node, edges] : graph) {
        for (const auto& edge : edges) {
            dist[node][edge.dest] = edge.weight;
            pred[node][edge.dest] = node;
        }
    }
}

string concatenate(const vector<string>& vec) {
    string result;
    for (const auto& str : vec) {
        if (!result.empty()) {
            result += " ";  // Add space before appending, but not for the first string
        }
        result += str;
    }
    return result;
}

void floydWarshall(const Graph& graph, DistanceMatrix& dist, PredecessorMatrix& pred) {
    /* Main Floyd's program */
    const int INF = numeric_limits<int>::max();

    for (const auto& [k, _] : graph) {
        for (const auto& [i, __] : graph) {
            for (const auto& [j, ___] : graph) {
                if (dist[i][k] != INF && dist[k][j] != INF &&
                    dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                    pred[i][j] = pred[k][j];
                }
            }
        }
    }
}

vector<string> reconstructPath(const PredecessorMatrix& pred, const string& start,
                               const string& end) {
    /* Reconstruct the path from the predecessor's list */
    vector<string> path;

    if (pred.at(start).at(end) == "") {
        return path;  // Empty path means no path exists
```

```
    }

    string at = end;
    while (at != start) {
        path.push_back(at);
        at = pred.at(start).at(at);
    }
    path.push_back(start);
    reverse(path.begin(), path.end());
    return path;
}

json getRoutesfromTimetable(const json& timetables, const Graph& graph,
                            const json& shortest_paths) {
    /* Obtain the routes as a json file from the timetables's json*/
    json routes;

    for (const auto& [student, timetable] : timetables.items()) {
        json week;
        for (const auto& [day, classes] : timetable.items()) {
            json today;

            // Start of the day, going from G_floor to the first class
            string path_str = shortest_paths["G" + classes["1"].get<string>()];
            today["0"] = path_str;

            // Lunch time is denoted by period "6", and periods after are delayed
by one
            // index
            path_str =
                shortest_paths["G" +
                               classes["7"].get<string>()];  // Originally, P7 is
the
                                                             // first afternoon
class
            today["7"] = path_str;

            int offset = 0;
            for (int period = 1; period <= 11; period++) {
                // Skip P7 as it's already been handled
                if (period == 7) {
                    offset = 1;  // Add offset to be minused because P7 is now P6
                    continue;
                }

                const string& current_period = to_string(period - offset);
                const string& current_room = classes[current_period];

                // cout << current_room << endl;

                if (current_room[0] != 'A' && current_room[0] != 'B' &&
                    current_room[0] != 'G')
                    continue;
                // If not the last class
                if (period != 11) {
                    // If AS/AL students have P6s, the should go to G_floor in the
end
                    if (period == 6 && stoi(student) < 22000) {
                        if (current_room == "G") {
```

```
                            today["6"] = "";
                            continue;
                        }
                        string path_str = shortest_paths[current_room + "G"];
                        today["6"] = path_str;
                        continue;
                    }

                    if (period == 6 && stoi(student) >= 22000) {
                        today["6"] = "";
                        continue;
                    }

                    const string& next_period = to_string(period - offset + 1);
                    const string& next_room = classes[next_period];

                    if (next_room[0] != 'A' && next_room[0] != 'B' && next_room[0] != 'G')
                        continue;

                    if (next_room == current_room)
                        today[to_string(period)] = "";
                    else {
                        string path_str = shortest_paths[current_room + next_room];
                        today[to_string(period)] = path_str;
                    }
                } else {
                    string path_str = shortest_paths[current_room + "G"];
                    today[to_string(period)] = path_str;
                }
            }
            week[day] = today;
        }
        routes[student] = week;
    }

    return routes;
}

// pre-calculated all the shortest paths in advance to avoid run-time calculations
void generate_all_paths(const Graph& graph) {
    DistanceMatrix dist;
    PredecessorMatrix pred;

    initializeDistanceAndPredecessorMatrices(graph, dist, pred);
    floydWarshall(graph, dist, pred);

    json routes;
    for (const auto [room1, _] : graph) {
        for (const auto [room2, __] : graph) {
            if (room1[0] != 'A' && room1[0] != 'B' && room1[0] != 'G') continue;
            if (room2[0] != 'A' && room2[0] != 'B' && room2[0] != 'G') continue;
            if (room1 == room2) continue;
            vector<string> path = reconstructPath(pred, room1, room2);
            string path_str = concatenate(path);
            routes[room1 + room2] = path_str;
        }
    }
```

```
225
226     std::ofstream file("shortest_paths.json");
227     if (file.is_open()) {
228         file << routes.dump(4);  // 4 spaces for indentation
229         file.close();
230         std::cout << "JSON data written to routes.json successfully." << endl;
231     } else {
232         std::cerr << "Failed to open the file for writing." << endl;
233     }
234 }
235
236 void generate_all_floyd_distances(const Graph& graph) {
237     DistanceMatrix dist;
238     PredecessorMatrix pred;
239
240     initializeDistanceAndPredecessorMatrices(graph, dist, pred);
241     floydWarshall(graph, dist, pred);
242
243     json distances;
244     for (const auto [room1, _] : graph) {
245         for (const auto [room2, __] : graph) {
246             if (room1[0] ≠ 'A' && room1[0] ≠ 'B' && room1[0] ≠ 'G') continue;
247             if (room2[0] ≠ 'A' && room2[0] ≠ 'B' && room2[0] ≠ 'G') continue;
248             if (room1 == room2) continue;
249             distances[room1 + room2] = dist[room1][room2];
250         }
251     }
252
253     std::ofstream file("../assets/distances.json");
254     if (file.is_open()) {
255         file << distances.dump(4);  // 4 spaces for indentation
256         file.close();
257         std::cout << "JSON data written to distances.json successfully." << endl;
258     } else {
259         std::cerr << "Failed to open the file for writing." << endl;
260     }
261 }
262
263 int main() {
264     string timetable_path;
265
266     ios::sync_with_stdio(false);
267     cin.tie(0);
268
269     // general input: the path to the time table json
270     // cin >> timetable_path;  // "../assets/timetable_0.json"
271     getline(cin, timetable_path);
272
273     // parse the JSON file
274     ifstream input_file(timetable_path);
275     if (!input_file.is_open()) {
276         cerr << "Failed to open the timetable file." << endl;
277         return 1;
278     }
279
280     json timetables;
281     input_file >> timetables;
282     input_file.close();
```

```
283
284     Graph graph = createSchoolGraph("../assets/paths.txt");
285     // generate_all_paths(graph);
286
287     ifstream paths_file("../assets/shortest_paths.json");
288     if (!paths_file.is_open()) {
289         cerr << "Failed to open shortest_paths.json" << endl;
290         return 1;
291     }
292
293     json shortest_paths;
294     paths_file >> shortest_paths;
295     paths_file.close();
296
297     json routes = getRoutesfromTimetable(timetables, graph, shortest_paths);
298
299     ofstream file("routes.json");
300     if (file.is_open()) {
301         file << routes.dump();  // faster i/o
302         file.close();
303         cout << "JSON data written to routes.json successfully." << endl;
304     } else {
305         cerr << "Failed to open the file for writing." << endl;
306     }
307
308     return 0;
309 }
310
```

multi_agent_path_finding/multi-objective-agent.cpp

```cpp
#include <climits>
#include <cmath>
#include <fstream>
#include <iostream>
#include <queue>
#include <sstream>
#include <string>
#include <unordered_map>
#include <vector>

#include "json.hpp"
using json = nlohmann::json;

using namespace std;

// Edge for the school's graph
struct Edge {
    string dest;  // destination node
    int weight;   // edge weight (in this case, distance)
    int type;     // edge type
};
// Struct to store the student's path at a period
struct StudentPath {
    string id;
    double rperf;
    vector<string> path;
};
// Override the comparison for the priority queue
struct CompareStudentPath {
    bool operator()(const StudentPath& s1, const StudentPath& s2) {
        return s1.rperf < s2.rperf;
    }
};

using Graph = unordered_map<string, vector<Edge>>;
using PathPQ = priority_queue<StudentPath, vector<StudentPath>,
CompareStudentPath>;

int BATCH_SIZE = 10;
const double CONGESTION_PENALTY = 10000.0;
int ITER_NUM = INT_MAX;
int ITER_COUNT = 1;
string ROUTE_FILE_PATH;
int ITER_SAVE_STEPS = 500;

// Code to create the school's layout graph from the path.txt
Graph createSchoolGraph(const string& file_path) {
    Graph graph;
    ifstream file(file_path);

    if (!file.is_open()) {
        cerr << "Failed to open the file." << endl;
        return graph;
    }

    string line;
```

```
    while (getline(file, line)) {
        istringstream iss(line);

        string node1, node2;
        int distance, type;

        iss >> node1 >> node2 >> distance >> type;

        Edge edge1 = {node2, distance, type};
        Edge edge2 = {node1, distance, type};

        graph[node1].push_back(edge1);
        graph[node2].push_back(edge2);  // Because it's an undirected graph
    }

    file.close();
    return graph;
}

// Codes for getting Floyd's shortest path for all the students
json getRoutesfromTimetable(const json& timetables, const Graph& graph,
                            const json& shortest_paths) {
    /* Obtain the routes as a json file from the timetables's json*/
    json routes;

    for (const auto& [student, timetable] : timetables.items()) {
        json week;
        for (const auto& [day, classes] : timetable.items()) {
            json today;

            // Start of the day, going from G_floor to the first class
            string path_str = shortest_paths["G" + classes["1"].get<string>()];
            today["0"] = path_str;

            // Lunch time is denoted by period "6", and periods after are delayed by one
            // index
            path_str =
                shortest_paths["G" +
                               classes["7"].get<string>()];  // Originally, P7 is the
                                                             // first afternoon class
            today["7"] = path_str;

            int offset = 0;
            for (int period = 1; period <= 11; period++) {
                // Skip P7 as it's already been handled
                if (period == 7) {
                    offset = 1;  // Add offset to be minused because P7 is now P6
                    continue;
                }

                const string& current_period = to_string(period - offset);
                const string& current_room = classes[current_period];

                // cout << current_room << endl;

                if (current_room[0] != 'A' && current_room[0] != 'B' &&
```

```
                    current_room[0] =/= 'G')
                    continue;
                // If not the last class
                if (period =/= 11) {
                    // If AS/AL students have P6s, the should go to G_floor in the end
                    if (period == 6 && stoi(student) < 22000) {
                        if (current_room == "G") {
                            today["6"] = "";
                            continue;
                        }
                        string path_str = shortest_paths[current_room + "G"];
                        today["6"] = path_str;
                        continue;
                    }

                    if (period == 6 && stoi(student) >= 22000) {
                        today["6"] = "";
                        continue;
                    }

                    const string& next_period = to_string(period - offset + 1);
                    const string& next_room = classes[next_period];

                    if (next_room[0] =/= 'A' && next_room[0] =/= 'B' && next_room[0] =/= 'G')
                        continue;

                    if (next_room == current_room)
                        today[to_string(period)] = "";
                    else {
                        string path_str = shortest_paths[current_room + next_room];
                        today[to_string(period)] = path_str;
                    }
                } else {
                    string path_str = shortest_paths[current_room + "G"];
                    today[to_string(period)] = path_str;
                }
            }
            week[day] = today;
        }
        routes[student] = week;
    }

    return routes;
}

// Code to slice a route string into a vector of strings
vector<string> vectorizeString(const string s) {
    vector<string> tokens;
    int start = 0;
    int end = s.find(' ');

    while (end =/= string::npos) {
        tokens.push_back(s.substr(start, end - start));
        start = end + 1;
        end = s.find(' ', start);
    }
```

```
168
169     tokens.push_back(s.substr(start, end));
170
171     return tokens;
172 }
173
174 // Code to concatenate a vector of strings into a single string
175 string concatenate(const vector<string>& vec) {
176     string result;
177     for (const auto& str : vec) {
178         if (!result.empty()) {
179             result += " ";  // Add space before appending, but not for the first
    string
180         }
181         result += str;
182     }
183     return result;
184 }
185
186 // Code to get the shortest path from start to end using Dijkstra's algorithm with
187 // congestion penalty
188 vector<string> getDijkstraPenaltiedPath(const string& start, const string& end,
189                                         const Graph& graph,
190                                         const unordered_map<string, int>&
    congestion) {
191     unordered_map<string, double> distances;
192     unordered_map<string, bool> visited;
193     unordered_map<string, string> prev;  // to store the path
194
195     vector<string> path;
196     if (start == end) return path = {"G", "G"};
197
198     for (const auto& [node, _] : graph) {
199         distances[node] = INT_MAX;
200         visited[node] = false;
201     }
202     distances[start] = 0;
203
204     priority_queue<pair<double, string>> pq;
205     pq.push({0, start});
206
207     while (!pq.empty()) {
208         string current = pq.top().second;
209         pq.pop();
210
211         if (visited[current]) continue;
212         visited[current] = true;
213
214         string destination;
215         try {
216             for (const auto& edge : graph.at(current)) {
217                 destination = edge.dest;
218                 double edgeWeight = edge.weight + (CONGESTION_PENALTY *
219                                                    congestion.at(current +
    edge.dest));
220                 if (!visited[edge.dest] &&
221                     distances[current] + edgeWeight < distances[edge.dest]) {
222                     distances[edge.dest] = distances[current] + edgeWeight;
223                     pq.push({-distances[edge.dest], edge.dest});
```

```
                    prev[edge.dest] =
                        current;  // save the previous node for path
reconstruction
                }
            }
        } catch (const out_of_range& e) {
            cout << current << endl;
            cerr << "Error: " << e.what() << endl;
        }
    }

    // Reconstruct path from 'end' to 'start' using the 'prev' mapping
    for (string at = end; at != ""; at = prev[at]) {
        path.push_back(at);
    }
    reverse(path.begin(), path.end());  // since the path is constructed in
reverse

    return path;
}

// Code to calculate r_perf using a whole congestion matrix
double computePerformanceIndex(const vector<string>& route,
                               const unordered_map<string, int>& congestion,
                               const Graph& graph) {
    double r_perf = 0.0;
    int c = 0;

    if (route[0] == "G" && route[1] == "G") return 0.0;  // spare class

    for (int i = 0; i < route.size() - 1; i++) {
        string start = route[i];
        string end = route[i + 1];

        if (start == "G" || end == "G") continue;  // Ignore the paths to and from
G_floor

        int w_i;
        for (const auto& edge : graph.at(start)) {
            if (edge.dest == end) {
                w_i = edge.weight;
                break;
            }
        }

        int c_i = congestion.at(start + end);
        r_perf +=
            w_i * (2 + (exp((c_i - 300.0) / 200.0) - exp(-(c_i - 300.0) / 200.0))
/
                           (exp((c_i - 300.0) / 200.0) + exp(-(c_i - 300.0) /
200.0)));
        c += c_i;
    }

    return r_perf;
}

// Code for a single iteration of the code for a single period
void iter(PathPQ& paths, double& sum_rperf, unordered_map<string, int>&
congestion,
```

```
          const Graph& graph, string& last_start, string& last_end,
          vector<StudentPath>& temp) {
    // A single iteration
    StudentPath worst_path;
    bool flag = false;
    while (!paths.empty()) {
        worst_path = paths.top();  // The student whose route has the greatest r_perf
        paths.pop();
        if (worst_path.path[0] == last_start &&
            worst_path.path[worst_path.path.size() - 1] == last_end) {
            temp.push_back(worst_path);
        } else {
            flag = true;
            break;
        }
    }
    if (!flag) return;
    sum_rperf -= worst_path.rperf;
    vector<string> new_route = getDijkstraPenaltiedPath(
        worst_path.path[0], worst_path.path[worst_path.path.size() - 1], graph,
        congestion);
    // for (const auto v: worst_path.path) cout << v << " ";
    // cout << endl;
    // for (const auto v: new_route) cout << v << " ";
    double new_rperf = computePerformanceIndex(new_route, congestion, graph);
    // cout << worst_path.rperf << " " << new_rperf << endl;
    if (new_rperf < worst_path.rperf) {
        StudentPath new_path = {worst_path.id, new_rperf, new_route};
        paths.push(new_path);
        sum_rperf += new_rperf;
        // cout << "UPDATED PATH " << worst_path.id << endl;
    } else {
        // if the path doesn't change, the student's path cannot be further optimized as
        // the congestion penalty is already too high
        temp.push_back(worst_path);
        sum_rperf += worst_path.rperf;
        last_start = worst_path.path[0],
        last_end = worst_path.path[worst_path.path.size() - 1];
    }
    // TODO:  validate last_start and last_end
}

// Code for multiple iterations of the code for a single period
void iterMultiple(PathPQ& paths, double& sum_rperf,
                  unordered_map<string, int>& congestion, const Graph& graph) {
    string last_start, last_end;
    vector<StudentPath> temp;
    PathPQ paths_copy = paths;
    double sum_rperf_copy = sum_rperf;
    for (int i = ITER_COUNT; i < ITER_NUM; i++) {
        // A single iteration
        iter(paths, sum_rperf, congestion, graph, last_start, last_end, temp);

        if (i % BATCH_SIZE | (i == 0))
            cout << "ITER " << i << " APPR " << sum_rperf << endl;
        else {  // When the batch size is met
            for (const auto& [node, edges] : graph) {
```

```
                for (const auto& edge : edges) {
                    congestion[node + edge.dest] = 0;
                }
            }

            vector<StudentPath>
                all_paths;  // contain all the paths from paths pq and temp vector
            PathPQ new_paths;
            // Get the congestion for the period
            while (!paths.empty()) {
                StudentPath path = paths.top();
                vector<string> route = path.path;
                paths.pop();
                for (int i = 0; i < route.size() - 1; i++)
                    congestion[route[i] + route[i + 1]]++;
                new_paths.push(path);
                all_paths.push_back(path);
            }

            for (const StudentPath& path : temp) {
                vector<string> route = path.path;
                for (int i = 0; i < route.size() - 1; i++)
                    congestion[route[i] + route[i + 1]]++;
                all_paths.push_back(path);
            }

            sum_rperf = 0.0;
            for (const StudentPath& path : all_paths) {
                double rperf = computePerformanceIndex(path.path, congestion,
graph);
                sum_rperf += rperf;
            }

            paths = new_paths;

            if (sum_rperf < sum_rperf_copy) {
                sum_rperf_copy = sum_rperf;
                paths_copy = paths;
            } else {
                temp.push_back(paths_copy.top());
                paths_copy.pop();
                paths = paths_copy;
                sum_rperf = sum_rperf_copy;
            }

            cout << "ITER " << i << " ACC" << sum_rperf << endl;
        }
    }
    return;
}

// Code for generating the route for a single period
void iterSinglePeriod(const int day, const int period, json& route_tables,
                      const Graph& graph) {
    // route_tables is the floyd_warshall's routes for all students at all periods
    // generated from gerRoutesfromTimetable() Initialize the congestion matrix
    unordered_map<string, int> congestion;
    for (const auto& [node, edges] : graph) {
        for (const auto& edge : edges) {
```

```
            congestion[node + edge.dest] = 0;
        }
    }

    // Get the congestion for the period
    for (const auto& [student, route_table] : route_tables.items()) {
        vector<string> route =
            vectorizeString(route_table[to_string(day)][to_string(period)].get<
string>());
        for (int i = 0; i < route.size() - 1; i++) {
            congestion[route[i] + route[i + 1]]++;
            congestion[route[i + 1] + route[i]]++;
        }
    }

    // Update the pq of paths
    PathPQ paths;
    double sum_rperf = 0;
    for (const auto& [student, route_table] : route_tables.items()) {
        vector<string> route =
            vectorizeString(route_table[to_string(day)][to_string(period)].get<
string>());
        double rperf = computePerformanceIndex(route, congestion, graph);
        StudentPath path = {student, rperf, route};
        paths.push(path);
        sum_rperf += rperf;
    }
    cout << "INITIAL PERFORMACE: " << sum_rperf << endl;

    // Run the iterations
    iterMultiple(paths, sum_rperf, congestion, graph);

    // Save the routes
    while (!paths.empty()) {
        StudentPath path = paths.top();
        paths.pop();
        route_tables[path.id][to_string(day)][to_string(period)] =
concatenate(path.path);
    }

    cout << "FINAL PERFORMANCE: " << sum_rperf << endl;

    return;
}

// Code for generating the route for a single day, with each iter covering all
periods
void iterSingleDay(const int day, json& route_tables, const Graph& graph,
                   json& perf_indices) {
    PathPQ paths[15], paths_copy[15];
    double sum_rperf[15] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
           sum_rperf_copy[15];
    unordered_map<string, int> congestions[15];
    double prev_best_rperf[15];

    for (int period = 0; period <= 11; period++) {
        if (period == 1 || period == 3 || period == 8 || period == 10)
            continue;  // All the students are having classes right now

```

```
        for (const auto& [node, edges] : graph) {
            for (const auto& edge : edges) {
                congestions[period][node + edge.dest] = 0;
            }
        }

        // Get the congestion for the period
        for (const auto& [student, route_table] : route_tables.items()) {
            const string route_str = route_table[to_string(day)][to_string(period)
];
            if (route_str == "") continue;
            vector<string> route = vectorizeString(route_str);
            for (int i = 0; i < route.size() - 1; i++) {
                congestions[period][route[i] + route[i + 1]]++;
                congestions[period][route[i + 1] + route[i]]++;
            }
        }

        for (const auto& [student, route_table] : route_tables.items()) {
            const string route_str = route_table[to_string(day)][to_string(period)
];
            vector<string> route;
            if (route_str == "")
                route = {"G", "G"};  // if this student is having a spare class
            else
                route = vectorizeString(route_str);
            double rperf = computePerformanceIndex(route, congestions[period],
graph);
            StudentPath path = {student, rperf, route};
            paths[period].push(path);
            sum_rperf[period] += rperf;
        }

        paths_copy[period] = paths[period];
        sum_rperf_copy[period] = sum_rperf[period];
        prev_best_rperf[period] = sum_rperf[period];

        // cout << "PERIOD " << period << " INITIAL PERFORMANCE: " <<
sum_rperf[period] <<
        // endl;
    }

    string last_start[15], last_end[15];
    vector<StudentPath> temp[15];
    for (int i = ITER_COUNT + 1; i <= ITER_NUM; i++) {
        for (int period = 0; period <= 11; period++) {
            if (period == 1 || period == 3 || period == 8 || period == 10)
                continue;  // All the students are having classes right now

            iter(paths[period], sum_rperf[period], congestions[period], graph,
                 last_start[period], last_end[period], temp[period]);
            prev_best_rperf[period] = min(sum_rperf[period],
prev_best_rperf[period]);

            if (i % BATCH_SIZE | (i == 0)) {
                // cout << fixed << setprecision(0) << "0 " << i << ' ' << day <<
' ' <<
                // period << ' ' << sum_rperf[period] << ' ' <<
prev_best_rperf[period] <<
                // endl;
```

```
            } else {
                for (const auto& [node, edges] : graph) {
                    for (const auto& edge : edges) {
                        congestions[period][node + edge.dest] = 0;
                    }
                }

                vector<StudentPath>
                    all_paths;  // contain all the paths from paths pq and temp
vector
                PathPQ new_paths;
                // Get the congestion for the period
                while (!paths[period].empty()) {
                    StudentPath path = paths[period].top();
                    vector<string> route = path.path;
                    paths[period].pop();
                    for (int i = 0; i < route.size() - 1; i++)
                        congestions[period][route[i] + route[i + 1]]++;
                    new_paths.push(path);
                    all_paths.push_back(path);
                }

                for (const StudentPath& path : temp[period]) {
                    vector<string> route = path.path;
                    for (int i = 0; i < route.size() - 1; i++)
                        congestions[period][route[i] + route[i + 1]]++;
                    all_paths.push_back(path);
                }

                sum_rperf[period] = 0.0;
                for (const StudentPath& path : all_paths) {
                    double rperf =
                        computePerformanceIndex(path.path, congestions[period],
graph);
                    sum_rperf[period] += rperf;
                }

                paths[period] = new_paths;

                if (sum_rperf[period] > sum_rperf_copy[period]) {
                    temp[period].push_back(paths_copy[period].top());
                    paths_copy[period].pop();
                    paths[period] = paths_copy[period];
                    sum_rperf[period] = sum_rperf_copy[period];
                } else {
                    paths_copy[period] = paths[period];
                    sum_rperf_copy[period] = sum_rperf[period];
                }
                prev_best_rperf[period] = min(sum_rperf[period],
prev_best_rperf[period]);
                perf_indices[to_string(day)][to_string(period)] =
int(sum_rperf[period]);
                cout << fixed << setprecision(0) << "0 " << i << ' ' << day << ' '
                     << period << ' ' << sum_rperf[period] << ' '
                     << prev_best_rperf[period] << endl;
            }

            if (!(i % ITER_SAVE_STEPS) && period == 0 &&
                i) {  // for every 500 iterations dump the json file
```

```
                vector<pair<string, string>>
                    all_paths;  // contain the students and their related paths

                while (!paths[period].empty()) {
                    StudentPath path = paths[period].top();
                    paths[period].pop();
                    all_paths.push_back({path.id, concatenate(path.path)});
                }

                for (const StudentPath& path : temp[period])
                    all_paths.push_back({path.id, concatenate(path.path)});

                for (const auto& path : all_paths)
                    route_tables[path.first][to_string(day)][to_string(period)] =
                        path.second;

                json iter_output;
                iter_output["iter"] = i;
                iter_output["indices"] = perf_indices;
                iter_output["routes"] = route_tables;

                ofstream iter_output_file(ROUTE_FILE_PATH + "_" + to_string(day) +
                                          ".json");
                if (iter_output_file.is_open()) {
                    iter_output_file << iter_output.dump();  // minimize size
                    iter_output_file.close();
                    // cout << "JSON data written to routes.json successfully." <<
endl;
                    cout << fixed << setprecision(0) << "1 " << i << ' ' << day <<
' '
                         << period << ' ' << sum_rperf[period] << ' '
                         << prev_best_rperf[period] << endl;
                } else {
                    // cerr << "Failed to open the file for writing." << endl;
                    cout << fixed << setprecision(0) << "! " << i << ' ' << day <<
' '
                         << period << ' ' << sum_rperf[period] << ' '
                         << prev_best_rperf[period] << endl;
                }
            }
        }
    }
}

int main(int argc, char** argv) {
    // args parsing
    int day;
    for (int i = 0; i < argc; ++i) {
        if (strcmp(argv[i], "-b") == 0 &&
            i + 1 < argc) {  // Batch Size: the amount of iterations before
updating the
                             // congestion
            BATCH_SIZE = stoi(argv[i + 1]);
        }
        if (strcmp(argv[i], "-f") == 0 &&
            i + 1 < argc) {  // Route File Path: the path to the routes file
            ROUTE_FILE_PATH = argv[i + 1];
        }
        if (strcmp(argv[i], "-d") == 0 &&
```

```
            i + 1 < argc) {  // Day: the day to run the algorithm for
            day = stoi(argv[i + 1]);
        }
        if (strcmp(argv[i], "-s") == 0 &&
            i + 1 < argc) {  // Save Iteration Step: the amount of iterations to
before
                             // each save
            ITER_SAVE_STEPS = stoi(argv[i + 1]);
        }
    }

    // parse the shortest paths json file
    ifstream route_tables_file(ROUTE_FILE_PATH);
    if (!route_tables_file.is_open()) {
        cerr << "Failed to open routes file" << endl;
        return 1;
    }

    // save the shortest paths
    json route_tables, perf_indices;
    route_tables_file >> route_tables;
    route_tables_file.close();

    ITER_COUNT = route_tables["iter"][day - 1];
    perf_indices = route_tables["indices"];
    route_tables = route_tables["routes"];

    // create the layout graph for the school
    Graph graph = createSchoolGraph("../assets/paths.txt");

    // run the algorithm for a period
    iterSingleDay(day, route_tables, graph, perf_indices);

    return 0;
}
```

optiway/src/app_init.rs

```rust
pub fn setup_custom_fonts(ctx: &egui::Context) {
    let mut fonts = egui::FontDefinitions::default();

    fonts.font_data.insert(
        "inter_font".to_owned(),
        egui::FontData::from_static(include_bytes!("../assets/fonts/Inter-micfong.ttf"))
    );
    fonts.font_data.insert(
        "jetbrains_mono_font".to_owned(),
        egui::FontData::from_static(include_bytes!("../assets/fonts/JetBrainsMono-Regular.ttf"))
    );
    fonts.font_data.insert(
        "source_han_sans_font".to_owned(),
        egui::FontData::from_static(include_bytes!("../assets/fonts/SourceHanSansSC-Regular.otf"))
    );
    fonts.font_data.insert(
        "material_design_icons_font".to_owned(),
        egui::FontData::from_static(include_bytes!("../assets/fonts/MaterialDesignIcons.ttf"))
    );
    fonts.font_data.insert(
        "material_symbols_font".to_owned(),
        egui::FontData::from_static(include_bytes!("../assets/fonts/MaterialSymbolsOutlined.ttf"))
    );
    fonts.families
        .entry(egui::FontFamily::Proportional)
        .or_default()
        .insert(0, "inter_font".to_owned());
    fonts.families
        .entry(egui::FontFamily::Proportional)
        .or_default()
        .insert(1, "source_han_sans_font".to_owned());
    fonts.families
        .entry(egui::FontFamily::Proportional)
        .or_default()
        .insert(2, "material_design_icons_font".to_owned());
    fonts.families
        .entry(egui::FontFamily::Proportional)
        .or_default()
        .insert(3, "material_symbols_font".to_owned());

    fonts.families
        .entry(egui::FontFamily::Monospace)
        .or_default()
        .insert(0, "jetbrains_mono_font".to_owned());

    // Tell egui to use these fonts:
    ctx.set_fonts(fonts);
}

pub fn setup_custom_styles(ctx: &egui::Context) {
    let mut style: egui::Style = (*ctx.style()).clone();
    style.visuals.window_rounding = (0.0).into();
```

```
    style.visuals.menu_rounding = (0.0).into();
    style.visuals.widgets.noninteractive.rounding = (0.0).into();
    style.visuals.widgets.inactive.rounding = (0.0).into();
    style.visuals.widgets.hovered.rounding = (0.0).into();
    style.visuals.widgets.active.rounding = (0.0).into();
    style.visuals.widgets.open.rounding = (0.0).into();
    style.visuals.slider_trailing_fill = true;
    style.visuals.override_text_color = Some(egui::Color32::from_rgb(180, 180, 180)
);
    // style.animation_time = 1.0;
    ctx.set_style(style);
}

```

optiway/src/app.rs

```rust
use std::{
    collections::HashMap,
    f32::consts::PI,
    fmt::Display,
    fs::{ self, File },
    io::{ BufRead, BufReader, Read, Write },
    path::{ Path, PathBuf },
    process::{ Command, Stdio },
    sync::{ Arc, Mutex },
    thread,
};

use egui::{
    color_picker,
    emath,
    pos2,
    CentralPanel,
    Color32,
    ColorImage,
    ComboBox,
    Grid,
    Layout,
    ProgressBar,
    Rect,
    RichText,
    Slider,
    Stroke,
    TextureHandle,
    Window,
};
use num_format::{ Locale, ToFormattedString };
use rfd::FileDialog;

use crate::{ md_icons::material_design_icons, setup_custom_fonts, setup_custom_styles };

#[derive(Default, Clone, PartialEq, Eq)]
enum PathDisplay {
    #[default]
    Shortest,
    Optimized,
}

impl Display for PathDisplay {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            PathDisplay::Shortest => write!(f, "Shortest route"),
            PathDisplay::Optimized => write!(f, "Optimized route"),
        }
    }
}

#[derive(Default, Clone, PartialEq, Eq)]
enum TimetableValidationStatus {
    #[default]
    Ready,
```

```
    Validating(i32, String),
    Failed(String),
    Successful,
}

impl TimetableValidationStatus {
    fn get_error_message(&self) -> String {
        match self {
            TimetableValidationStatus::Failed(message) => message.clone(),
            _ => String::new(),
        }
    }
}

#[derive(Default, Clone, PartialEq, Eq, serde::Deserialize, serde::Serialize)]
struct MainJSONOutputFile {
    iter: [u64; 5],
    indices: HashMap<u32, HashMap<usize, u128>>,
    routes: HashMap<String, HashMap<u32, HashMap<usize, String>>>,
}

#[derive(Default, Clone, PartialEq, Eq, serde::Deserialize, serde::Serialize)]
struct SubJSONOutputFile {
    iter: u64,
    indices: HashMap<u32, HashMap<usize, u128>>,
    routes: HashMap<String, HashMap<u32, HashMap<usize, String>>>,
}

#[derive(Default, Clone, PartialEq, Eq)]
enum OptimizationStatus {
    #[default]
    ParamInput,
    Ready,
    Calculating,
    AbortSignal,
    Failed(String),
}

#[derive(Default)]
struct TimetableFileInfo {
    filename: String,
    filepath: PathBuf,
    student_count: Arc<Mutex<Option<i32>>>,
    session_count: Arc<Mutex<Option<i32>>>,
    validation_status: Arc<Mutex<TimetableValidationStatus>>,
    timetable: Arc<Mutex<Option<serde_json::Value>>>,
}

#[derive(Default, Clone, PartialEq, Eq)]
enum PathGenerationStatus {
    #[default]
    Ready,
    Generating(i32, String),
    Failed(String),
    LoadingJSON,
    Successful,
}

```

```rust
impl PathGenerationStatus {
    fn is_generating(&self) -> bool {
        matches!(self, PathGenerationStatus::Generating(_, _))
    }

    fn is_loading_json(&self) -> bool {
        matches!(self, PathGenerationStatus::LoadingJSON)
    }
}

#[derive(Default, Clone, PartialEq, Eq)]
enum CongestionStatus {
    #[default]
    Ready,
    Generating(i32, String),
    Failed(String),
    GeneratingPI(i32, String),
    Successful,
}
impl CongestionStatus {
    fn is_generating(&self) -> bool {
        if let CongestionStatus::Generating(_, _) = self {
            true
        } else {
            matches!(self, CongestionStatus::GeneratingPI(_, _))
        }
    }
}

type Routes = HashMap<String, HashMap<u32, HashMap<usize, String>>>;
type CongestionPoint = HashMap<u32, HashMap<usize, HashMap<String, u32>>>;
type CongestionPath = HashMap<u32, HashMap<usize, HashMap<(String, String), u32>>
>;

struct CongestionStatistics {
    point_count: HashMap<u32, HashMap<usize, Vec<u32>>>,
    path_count: HashMap<u32, HashMap<usize, Vec<u32>>>,
}

impl Default for CongestionStatistics {
    fn default() -> Self {
        Self {
            point_count: {
                let mut point_count = HashMap::new();
                for day in 1..=5 {
                    point_count.insert(day, HashMap::new());
                    for period in 0..=11 {
                        point_count
                            .get_mut(&day)
                            .unwrap()
                            .insert(period, vec![0; 7]);
                    }
                }
                point_count
            },
            path_count: {
                let mut path_count = HashMap::new();
                for day in 1..=5 {
                    path_count.insert(day, HashMap::new());
```

```
                    for period in 0..=11 {
                        path_count
                            .get_mut(&day)
                            .unwrap()
                            .insert(period, vec![0; 7]);
                    }
                }
                path_count
            },
        }
    }
}

pub struct OptiWayApp {
    selected_student: Option<String>,
    student_list: Arc<Mutex<Vec<String>>>,
    selected_period: usize,
    selected_day: u32,
    selected_floor: [bool; 9],
    selected_floor_index: usize,
    textures: Vec<Option<TextureHandle>>,
    inactive_brightness: u8,
    projection_coords: HashMap<String, [i32; 3]>,
    active_path_color: Color32,
    inactive_path_color: Color32,
    show_path_window: bool,
    show_json_validation: bool,
    timetable_file_info: TimetableFileInfo,
    student_number_search: String,
    path_generation_status: Arc<Mutex<PathGenerationStatus>>,
    show_path_gen_window: bool,
    student_routes_shortest: Arc<Mutex<Option<Routes>>>,
    student_routes_optimized: Arc<Mutex<Option<Routes>>>,
    show_timetable_window: bool,
    show_congestion_window: bool,
    congestion_status: Arc<Mutex<CongestionStatus>>,
    congestion_point_data: Arc<Mutex<CongestionPoint>>,
    congestion_path_data: Arc<Mutex<CongestionPath>>,
    congestion_point_data_opt: Arc<Mutex<CongestionPoint>>,
    congestion_path_data_opt: Arc<Mutex<CongestionPath>>,
    maximum_congestion: Arc<Mutex<u32>>,
    maximum_congestion_opt: Arc<Mutex<u32>>,
    congestion_statistics: Arc<Mutex<CongestionStatistics>>,
    congestion_statistics_opt: Arc<Mutex<CongestionStatistics>>,
    show_congestion: bool,
    congestion_filter: u32,
    show_congestion_path: bool,
    show_congestion_point: bool,
    show_pi_window: bool,
    #[allow(unused)]
    shortest_paths_json: HashMap<String, String>,
    shortest_paths_content: Arc<Mutex<String>>,
    show_pi_shortest: bool,
    performance_indices_shortest: Arc<Mutex<HashMap<u32, HashMap<usize, u128>>>>,
    performance_indices_optimized: Arc<Mutex<HashMap<u32, HashMap<usize, u128>>>>
,
    path_distances: Arc<Mutex<HashMap<String, HashMap<String, u32>>>>,
    optimization_status: Arc<Mutex<OptimizationStatus>>,
    show_optimization_window: bool,
```

```
    param_batch_size: u32,
    param_save_every: u32,
    param_use_shortest_path: bool,
    param_day: u32,
    param_filename: String,
    param_filepath: PathBuf,
    current_iter: Arc<Mutex<[u64; 5]>>,
    current_period_iter: Arc<Mutex<[u64; 5]>>,
    path_display: PathDisplay,
}

impl Default for OptiWayApp {
    fn default() -> Self {
        let mut floors = [false; 9];
        floors[0] = true;
        Self {
            selected_student: Default::default(),
            student_list: Default::default(),
            selected_period: 0,
            selected_day: 1,
            selected_floor: floors,
            selected_floor_index: 0,
            textures: vec![None; 9],
            inactive_brightness: 64,
            projection_coords: serde_yaml
                ::from_str(include_str!("../assets/projection-coords-
flatten.yaml"))
                .unwrap(),
            active_path_color: Color32::from_rgb(0xec, 0x6f, 0x27),
            inactive_path_color: Color32::from_gray(0x61),
            show_path_window: false,
            show_json_validation: false,
            timetable_file_info: Default::default(),
            student_number_search: Default::default(),
            path_generation_status: Default::default(),
            show_path_gen_window: false,
            student_routes_shortest: Default::default(),
            student_routes_optimized: Default::default(),
            show_timetable_window: false,
            show_congestion_window: false,
            congestion_status: Default::default(),
            congestion_point_data: Arc::new(
                Mutex::new({
                    let mut congestion_data = CongestionPoint::new();
                    for day in 1..=5 {
                        congestion_data.insert(day, HashMap::new());
                        for period in 0..=11 {
                            congestion_data.get_mut(&day).unwrap().insert(period,
HashMap::new());
                        }
                    }
                    congestion_data
                })
            ),
            congestion_path_data: Arc::new(
                Mutex::new({
                    let mut congestion_data = CongestionPath::new();
                    for day in 1..=5 {
                        congestion_data.insert(day, HashMap::new());
```

```
                        for period in 0..=11 {
                            congestion_data.get_mut(&day).unwrap().insert(period, HashMap::new());
                        }
                    }
                    congestion_data
                })
            ),
            congestion_point_data_opt: Arc::new(
                Mutex::new({
                    let mut congestion_data = CongestionPoint::new();
                    for day in 1..=5 {
                        congestion_data.insert(day, HashMap::new());
                        for period in 0..=11 {
                            congestion_data.get_mut(&day).unwrap().insert(period, HashMap::new());
                        }
                    }
                    congestion_data
                })
            ),
            congestion_path_data_opt: Arc::new(
                Mutex::new({
                    let mut congestion_data = CongestionPath::new();
                    for day in 1..=5 {
                        congestion_data.insert(day, HashMap::new());
                        for period in 0..=11 {
                            congestion_data.get_mut(&day).unwrap().insert(period, HashMap::new());
                        }
                    }
                    congestion_data
                })
            ),
            congestion_statistics: Default::default(),
            congestion_statistics_opt: Default::default(),
            maximum_congestion: Default::default(),
            maximum_congestion_opt: Default::default(),
            show_congestion: false,
            congestion_filter: 0,
            show_congestion_path: true,
            show_congestion_point: true,
            show_pi_window: false,
            shortest_paths_json: serde_json
                ::from_str(include_str!("../assets/shortest_paths.json"))
                .unwrap(),
            performance_indices_shortest: {
                let mut performance_indices = HashMap::new();
                for day in 1..=5 {
                    performance_indices.insert(day, HashMap::new());
                    for period in 0..=11 {
                        performance_indices.get_mut(&day).unwrap().insert(period, 0);
                    }
                }
                Arc::new(Mutex::new(performance_indices))
            },
            performance_indices_optimized: {
                let mut performance_indices = HashMap::new();
```

```
                for day in 1..=5 {
                    performance_indices.insert(day, HashMap::new());
                    for period in 0..=11 {
                        performance_indices.get_mut(&day).unwrap().insert(period, 0);
                    }
                }
                Arc::new(Mutex::new(performance_indices))
            },
            show_pi_shortest: true,
            path_distances: Arc::new(
                Mutex::new({
                    let mut path_distances = HashMap::new();
                    for line in include_str!("../assets/paths.txt").lines() {
                        let line = line.to_owned();
                        let mut line = line.split(' ');
                        let room1 = line.next().unwrap().to_owned();
                        let room2 = line.next().unwrap().to_owned();
                        let distance = line.next().unwrap().parse::<u32>().unwrap();
                        // insert if not present
                        if !path_distances.contains_key(&room1) {
                            path_distances.insert(room1.clone(), HashMap::new());
                        }
                        path_distances.get_mut(&room1).unwrap().insert(room2.clone(), distance);
                        if !path_distances.contains_key(&room2) {
                            path_distances.insert(room2.clone(), HashMap::new());
                        }
                        path_distances.get_mut(&room2).unwrap().insert(room1, distance);
                    }
                    path_distances
                })
            ),
            optimization_status: Default::default(),
            show_optimization_window: false,
            param_batch_size: 100,
            param_save_every: 2500,
            param_use_shortest_path: true,
            param_day: 1,
            param_filename: Default::default(),
            param_filepath: Default::default(),
            shortest_paths_content: Default::default(),
            current_iter: Default::default(),
            current_period_iter: Default::default(),
            path_display: Default::default(),
        }
    }
}

impl OptiWayApp {
    pub fn new(cc: &eframe::CreationContext<'_>) -> Self {
        setup_custom_fonts(&cc.egui_ctx);
        setup_custom_styles(&cc.egui_ctx);

        // if let Some(storage) = cc.storage {
        //     return eframe::get_value(storage, eframe::APP_KEY).unwrap_or_default();
        // }
```

```
        Default::default()
    }

    fn show_path_generation_window(
        &mut self,
        ctx: &egui::Context,
        current_path_status: PathGenerationStatus
    ) {
        Window::new("Calculating Path").show(ctx, |ui| {
            ui.heading("Metadata");
            Grid::new("path_generation_grid")
                .num_columns(2)
                .show(ui, |ui| {
                    ui.label("Algorithm");
                    ui.label("Minimize distance");
                    ui.end_row();

                    ui.label("Student count");
                    if
                        let Some(student_count) =
*self.timetable_file_info.student_count
                            .lock()
                            .unwrap()
                    {
                        ui.label(format!("{}", student_count));
                    } else {
                        ui.label("-");
                    }
                    ui.end_row();

                    ui.label("Session count");
                    if
                        let Some(session_count) =
*self.timetable_file_info.session_count
                            .lock()
                            .unwrap()
                    {
                        ui.label(format!("{}", session_count));
                    } else {
                        ui.label("-");
                    }
                    ui.end_row();
                });
            ui.separator();
            match current_path_status {
                PathGenerationStatus::Ready => {
                    *self.path_generation_status.lock().unwrap() =
PathGenerationStatus::Generating(
                        0,
                        "Calculating path".to_owned()
                    );
                    let filepath = self.timetable_file_info.filepath.clone();
                    let path_generation_status_arc =
self.path_generation_status.clone();
                    let student_paths_arc = self.student_routes_shortest.clone();
                    let shortest_paths_content_arc =
self.shortest_paths_content.clone();
                    thread::spawn(move || {
```

```
                        *shortest_paths_content_arc.lock().unwrap() = run_floyd_algorithm_cpp(
                            filepath,
                            path_generation_status_arc,
                            student_paths_arc
                        );
                    });
                    // let shortest_paths_json = self.shortest_paths_json.clone();
                    // thread::spawn(move || run_floyd_algorithm_rust(filepath, shortest_paths_json, path_generation_status_arc, student_paths_arc));
                }
                PathGenerationStatus::Generating(_progress, message) => {
                    ui.with_layout(Layout::top_down_justified(egui::Align::Center), |ui| {
                        ui.label(RichText::new(material_design_icons::MDI_MAP_SEARCH).size(32.0));
                        ui.label(message);
                        ui.spinner();
                    });
                }
                PathGenerationStatus::Failed(message) => {
                    ui.with_layout(Layout::top_down_justified(egui::Align::Center), |ui| {
                        ui.label(
                            RichText::new(material_design_icons::MDI_SIGN_DIRECTION_REMOVE)
                                .size(32.0)
                                .color(Color32::from_rgb(0xe4, 0x37, 0x48))
                        );
                        ui.label("Path calculation failed");
                        ui.label(message);
                        if ui.button("Close").clicked() {
                            self.show_path_gen_window = false;
                        }
                    });
                }
                PathGenerationStatus::LoadingJSON => {
                    ui.with_layout(Layout::top_down_justified(egui::Align::Center), |ui| {
                        ui.label(
                            RichText::new(material_design_icons::MDI_MAP_CHECK)
                                .size(32.0)
                                .color(Color32::from_rgb(0x14, 0xae, 0x52))
                        );
                        ui.label("Loading paths");
                        ui.label(
                            "The paths have been successfully calculated. OptiWay is now loading the paths."
                        );
                        ui.spinner();
                    });
                }
                PathGenerationStatus::Successful => {
                    ui.with_layout(Layout::top_down_justified(egui::Align::Center), |ui| {
                        ui.label(
                            RichText::new(material_design_icons::MDI_MAP_CHECK)
                                .size(32.0)
                                .color(Color32::from_rgb(0x14, 0xae, 0x52))
                        );
```

```
                        ui.label("Path calculation successful");
                        ui.label(
                            "The paths have been successfully calculated and loaded. You may now view or export them."
                        );
                        if ui.button("Close").clicked() {
                            self.show_path_gen_window = false;
                        }
                    });
                }
            }
        });
    }

    fn show_optimization_window(
        &mut self,
        ctx: &egui::Context,
        current_optimization_status: OptimizationStatus
    ) {
        Window::new("Route Optimization").show(ctx, |ui| {
            match current_optimization_status {
                OptimizationStatus::ParamInput => {
                    ui.heading("Parameters");
                    ui.add(
                        Slider::new(&mut self.param_batch_size, 10..=200)
                            .step_by(10.0)
                            .text("Batch size")
                    );
                    ui.add(
                        Slider::new(&mut self.param_save_every, 100..=5000)
                            .step_by(100.0)
                            .text("Iterations per save")
                    );
                    // ComboBox::from_label("Day of Week")
                    //     .selected_text(convert_day_of_week(self.param_day))
                    //     .show_ui(ui, |ui| {
                    //         for i in 1..=5 {
                    //             ui.selectable_value(
                    //                 &mut self.param_day,
                    //                 i,
                    //                 convert_day_of_week(i),
                    //             );
                    //         }
                    //     });
                    ui.horizontal(|ui| {
                        ui.checkbox(&mut self.param_use_shortest_path, "Use shortest routes");
                        ui.add_enabled_ui(!self.param_use_shortest_path, |ui| {
                            if ui.button("Select route file").clicked() {
                                let file = FileDialog::new()
                                    .add_filter("JSON", &["json"])
                                    .pick_file();
                                if let Some(file) = file {
                                    self.param_filename = file
                                        .file_name()
                                        .unwrap()
                                        .to_str()
                                        .unwrap()
                                        .to_owned();
```

```
                                    self.param_filepath = file;
                                }
                            }
                        });
                        if !self.param_use_shortest_path {
                            ui.label(
                                if self.param_filename.is_empty() {
                                    "No file selected.".to_owned()
                                } else {
                                    self.param_filename.clone()
                                }
                            );
                        }
                    });
                    ui.separator();

ui.with_layout(Layout::top_down_justified(egui::Align::Center), |ui| {
                        if ui.button("Reset parameters").clicked() {
                            self.param_batch_size = 100;
                            self.param_save_every = 2500;
                            self.param_use_shortest_path = true;
                            self.param_day = 1;
                            self.param_filename = Default::default();
                            self.param_filepath = Default::default();
                        }
                        if ui.button("Start").clicked() {
                            self.show_pi_window = true;
                            self.show_pi_shortest = false;
                            *self.optimization_status.lock().unwrap() =
OptimizationStatus::Ready;
                        }
                    });
                }
                OptimizationStatus::Ready => {
                    *self.optimization_status.lock().unwrap() =
OptimizationStatus::Calculating;
                    let param_use_shortest_path = self.param_use_shortest_path;
                    let param_batch_size = self.param_batch_size;
                    let param_save_every = self.param_save_every;
                    for day in 1..=5 {
                        let param_day = day;
                        let mut param_filepath = self.param_filepath.clone();
                        let optimization_status_arc =
self.optimization_status.clone();
                        let shortest_paths_content_arc =
self.shortest_paths_content.clone();
                        let current_iter_arc = self.current_iter.clone();
                        let performance_indices_shortest_json = serde_json
                            ::to_string(&
*self.performance_indices_shortest.lock().unwrap())
                            .unwrap();
                        let performance_indices_optimized_arc =
                            self.performance_indices_optimized.clone();
                        let performance_indices_shortest_arc =
                            self.performance_indices_shortest.clone();
                        let current_period_iter_arc =
self.current_period_iter.clone();
                        if param_use_shortest_path {
                            self.param_filepath = fs::canonicalize("
./bin/routes.json").unwrap();
```

```
                            self.param_filename = "routes.json".to_owned();
                            self.param_use_shortest_path = false;
                        } else {
                            let mut file = File::open(&param_filepath).unwrap();
                            let mut content = String::new();
                            file.read_to_string(&mut content).unwrap();
                            let json: MainJSONOutputFile = serde_json::from_str(&
content).unwrap();
                            for day in 1..=5 {
                                *self.current_iter
                                    .lock()
                                    .unwrap()
                                    .get_mut((day - 1) as usize)
                                    .unwrap() = *json.iter.get((day - 1) as
usize).unwrap();
                            }
                            *self.performance_indices_optimized.lock().unwrap() =
json.indices;
                        }
                        thread::spawn(move || {
                            if param_use_shortest_path {
                                let content =
                                    "{\"iter\":[0,0,0,0,0],\"indices\":"
.to_owned() +
                                    &performance_indices_shortest_json +
                                    ",\"routes\":" +
                                    &shortest_paths_content_arc.lock().unwrap()
.to_owned() +
                                    "}";
                                *performance_indices_optimized_arc.lock()
.unwrap() =
                                    performance_indices_shortest_arc.lock()
.unwrap().clone();
                                let mut file = File::create("./bin/routes.json")
.unwrap();
                                file.write_all(content.as_bytes()).unwrap();
                                param_filepath = fs::canonicalize("
./bin/routes.json").unwrap();
                            }
                            let bin_dir = fs::canonicalize("./bin").unwrap();
                            if
                                let Ok(mut opt_command) = Command::new("
./optimization.out")
                                    .current_dir(bin_dir)
                                    .stdout(Stdio::piped())
                                    .stdin(Stdio::piped())
                                    .args([
                                        "-f",
                                        param_filepath.to_str().unwrap(),
                                        "-b",
                                        &param_batch_size.to_string(),
                                        "-s",
                                        &(param_save_every / 5).to_string(),
                                        "-d",
                                        &param_day.to_string(),
                                    ])
                                    .spawn()
                            {
                                if let Some(stdout) = opt_command.stdout.take() {
                                    let reader = BufReader::new(stdout);
                                    reader
```

```
                                    .lines()
                                    .map_while(Result::ok)
                                    .for_each(|line| {
                                        let report: Vec<&str> = line.split('
').collect();
                                        if report[0] == "1" {
                                            (*current_iter_arc.lock()
.unwrap())[
                                                (param_day - 1) as usize
                                            ] = report[1].parse::<u64>()
.unwrap();
                                            if
                                                optimization_status_arc
                                                    .lock()
                                                    .unwrap()
                                                    .clone() ==
OptimizationStatus::AbortSignal
                                            {
                                                opt_command
                                                    .kill()
                                                    .expect(
                                                        "Failed to interrupt
optimization algorithm"
                                                    );
                                            }
                                            (*current_period_iter_arc.lock()
.unwrap())[
                                                (param_day - 1) as usize
                                            ] = report[3].parse::<u64>()
.unwrap();
*performance_indices_optimized_arc
                                                .lock()
                                                .unwrap()
                                                .get_mut(&param_day)
                                                .unwrap()
                                                .get_mut(
                                                    &(
                                                        report[3]
                                                            .parse::<u64>()
                                                            .unwrap() as
usize
                                                    )
                                                )
                                                .unwrap() = report[4].parse::
<u128>().unwrap();
                                        }
                                    });
                            } else {
                                *optimization_status_arc.lock().unwrap() =
                                    OptimizationStatus::Failed(
                                        "Optimization algorithm terminated
unexpectedly".to_owned()
                                    );
                            }
                        } else {
                            *optimization_status_arc.lock().unwrap() =
                                OptimizationStatus::Failed(
                                    "Failed to start optimization algorithm"
.to_owned()
                                );
```

```
                            }
                        });
                    }
                }
                OptimizationStatus::Calculating => {
ui.with_layout(Layout::top_down_justified(egui::Align::Center), |ui| {
                        ui.label(RichText::new(material_design_icons::MDI_COG)
.size(32.0));
                        ui.label("Optimizing paths");
                        ui.label(
                            format!(
                                "Total Iteration: {}",
                                self.current_iter
                                    .lock()
                                    .unwrap()
                                    .iter()
                                    .sum::<u64>()
                                    .to_formatted_string(&Locale::fr)
                            )
                        );
                        ui.spinner();
                        if ui.button("Pause").clicked() {
                            *self.optimization_status.lock().unwrap() =
                                OptimizationStatus::AbortSignal;
                            // read from routes.json and load the json
                            let mut file = File::open("./bin/routes.json")
.unwrap();
                            let mut content = String::new();
                            file.read_to_string(&mut content).unwrap();
                            let mut json: MainJSONOutputFile = serde_json
                                ::from_str(&content)
                                .unwrap();
                            let mut sub_jsons: [SubJSONOutputFile; 5] =
Default::default();
                            for i in 1..=5 {
                                let mut file = File::open(
                                    format!("./bin/routes.json_{}.json", i)
                                ).unwrap();
                                let mut content = String::new();
                                file.read_to_string(&mut content).unwrap();
                                sub_jsons[i - 1] = serde_json::from_str(&content)
.unwrap();
                                json.iter[i - 1] = sub_jsons[i - 1].iter;
                                *json.indices.get_mut(&(i as u32)).unwrap() =
sub_jsons[
                                    i - 1
                                ].indices
                                    .get(&(i as u32))
                                    .unwrap()
                                    .clone();
                                for student_number in sub_jsons[i - 1]
.routes.keys() {
                                    *json.routes
                                        .get_mut(student_number)
                                        .unwrap()
                                        .get_mut(&(i as u32))
                                        .unwrap() = sub_jsons[i - 1].routes
                                        .get(student_number)
                                        .unwrap()
```

```
                                        .get(&(i as u32))
                                        .unwrap()
                                        .clone();
                                }
                            }
                            let mut file = File::create("./bin/routes.json")
.unwrap();
                            file.write_all(
                                serde_json::to_string(&json).unwrap().as_bytes()
                            ).unwrap();
                            *self.student_routes_optimized.lock().unwrap() =
Some(json.routes);
                        }
                    });
                }
                OptimizationStatus::AbortSignal => {

ui.with_layout(Layout::top_down_justified(egui::Align::Center), |ui| {
                        ui.label(
                            RichText::new(material_design_icons::MDI_COG_PAUSE)
                                .size(32.0)
                                .color(Color32::from_rgb(0xff, 0xc1, 0x07))
                        );
                        ui.label("Optimization paused");
                        ui.label("You may resume optimization at any time.");
                        if ui.button("Resume").clicked() {
                            *self.optimization_status.lock().unwrap() =
OptimizationStatus::Ready;
                        }
                        if ui.button("Close").clicked() {
                            self.show_optimization_window = false;
                            *self.optimization_status.lock().unwrap() =
                                OptimizationStatus::ParamInput;
                        }
                    });
                }
                OptimizationStatus::Failed(message) => {

ui.with_layout(Layout::top_down_justified(egui::Align::Center), |ui| {
                        ui.label(
                            RichText::new(material_design_icons::MDI_COG_OFF)
                                .size(32.0)
                                .color(Color32::from_rgb(0xe4, 0x37, 0x48))
                        );
                        ui.label("Route optimization failed");
                        ui.label(message);
                        if ui.button("Close").clicked() {
                            self.show_optimization_window = false;
                        }
                    });
                }
            }
        });
    }

    fn show_congestion_window(
        &mut self,
        ctx: &egui::Context,
        current_congestion_status: CongestionStatus
    ) {
```

```
        Window::new("Congestion Evaluation").show(ctx, |ui| {
            match current_congestion_status {
                CongestionStatus::Ready => {
                    *self.maximum_congestion.lock().unwrap() = 0;
                    *self.congestion_status.lock().unwrap() = CongestionStatus::Generating(
                        0,
                        "Evaluating congestion".to_owned()
                    );
                    *self.congestion_statistics.lock().unwrap() = Default::default();
                    *self.congestion_point_data.lock().unwrap() = {
                        let mut congestion_data = CongestionPoint::new();
                        for day in 1..=5 {
                            congestion_data.insert(day, HashMap::new());
                            for period in 0..=11 {
                                congestion_data
                                    .get_mut(&day)
                                    .unwrap()
                                    .insert(period, HashMap::new());
                            }
                        }
                        congestion_data
                    };
                    *self.congestion_point_data_opt.lock().unwrap() = {
                        let mut congestion_data = CongestionPoint::new();
                        for day in 1..=5 {
                            congestion_data.insert(day, HashMap::new());
                            for period in 0..=11 {
                                congestion_data
                                    .get_mut(&day)
                                    .unwrap()
                                    .insert(period, HashMap::new());
                            }
                        }
                        congestion_data
                    };
                    *self.congestion_path_data.lock().unwrap() = {
                        let mut congestion_data = CongestionPath::new();
                        for day in 1..=5 {
                            congestion_data.insert(day, HashMap::new());
                            for period in 0..=11 {
                                congestion_data
                                    .get_mut(&day)
                                    .unwrap()
                                    .insert(period, HashMap::new());
                            }
                        }
                        congestion_data
                    };
                    *self.congestion_path_data_opt.lock().unwrap() = {
                        let mut congestion_data = CongestionPath::new();
                        for day in 1..=5 {
                            congestion_data.insert(day, HashMap::new());
                            for period in 0..=11 {
                                congestion_data
                                    .get_mut(&day)
                                    .unwrap()
                                    .insert(period, HashMap::new());
```

```
                            }
                        }
                        congestion_data
                    };
                    let congestion_point_data_arc = self.congestion_point_data.clone();
                    let congestion_path_data_arc = self.congestion_path_data.clone();
                    let congestion_point_data_opt_arc = self.congestion_point_data_opt.clone();
                    let congestion_path_data_opt_arc = self.congestion_path_data_opt.clone();
                    let congestion_status_arc = self.congestion_status.clone();
                    let max_congestion_arc = self.maximum_congestion.clone();
                    let max_congestion_opt_arc = self.maximum_congestion_opt.clone();
                    let congestion_statistics_arc = self.congestion_statistics.clone();
                    let congestion_statistics_opt_arc = self.congestion_statistics_opt.clone();
                    let path_distances_arc = self.path_distances.clone();
                    let mut rooms: Vec<String> = self.projection_coords
                        .clone()
                        .keys()
                        .map(|k| k.to_owned())
                        .collect();
                    let performance_indices_shortest_arc =
                        self.performance_indices_shortest.clone();
                    rooms.push("G".to_owned());
                    let student_routes = self.student_routes_shortest.lock().unwrap().clone();
                    if student_routes.is_none() {
                        *congestion_status_arc.lock().unwrap() = CongestionStatus::Failed(
                            "No path data available".to_owned()
                        );
                        return;
                    }
                    let student_routes = student_routes.unwrap();
                    let student_routes_opt = self.student_routes_optimized.lock().unwrap().clone();
                    thread::spawn(move || {
                        if let Some(student_routes) = student_routes_opt {
                            for day in 1..=5 {
                                for period in 0..=11 {
                                    for room in &rooms {
                                        congestion_point_data_opt_arc
                                            .lock()
                                            .unwrap()
                                            .get_mut(&day)
                                            .unwrap()
                                            .get_mut(&period)
                                            .unwrap()
                                            .insert(room.to_owned(), 0);
                                    }
                                }
                            }
                            for student in &student_routes {
                                for day in 1..=5 {
                                    for period in 0..=11 {
                                        let rooms = student_routes
```

```
                                        .get(student.0)
                                        .unwrap()
                                        .get(&day)
                                        .unwrap()
                                        .get(&period)
                                        .unwrap()
                                        .split(' ')
                                        .collect::<Vec<&str>>();
                                    let mut previous_room = "";
                                    for room in rooms {
                                        if room.is_empty() || room == "G" {
                                            continue;
                                        }
                                        if !previous_room.is_empty() {
                                            let contains = congestion_path_data_opt_arc
                                                .lock()
                                                .unwrap()
                                                .get(&day)
                                                .unwrap()
                                                .get(&period)
                                                .unwrap()
                                                .contains_key(
                                                    &(previous_room.to_owned(), room.to_owned())
                                                );
                                            if contains {
                                                *congestion_path_data_opt_arc
                                                    .lock()
                                                    .unwrap()
                                                    .get_mut(&day)
                                                    .unwrap()
                                                    .get_mut(&period)
                                                    .unwrap()
                                                    .get_mut(
                                                        &(
                                                            previous_room.to_owned(),
                                                            room.to_owned(),
                                                        )
                                                    )
                                                    .unwrap() += 1;
                                            } else {
                                                congestion_path_data_opt_arc
                                                    .lock()
                                                    .unwrap()
                                                    .get_mut(&day)
                                                    .unwrap()
                                                    .get_mut(&period)
                                                    .unwrap()
                                                    .insert(
                                                        (
                                                            previous_room.to_owned(),
                                                            room.to_owned(),
                                                        ),
                                                        1
                                                    );
                                            }
```

```
                                                let contains =
congestion_path_data_opt_arc
                                                    .lock()
                                                    .unwrap()
                                                    .get(&day)
                                                    .unwrap()
                                                    .get(&period)
                                                    .unwrap()
                                                    .contains_key(
                                                        &(room.to_owned(),
previous_room.to_owned())
                                                    );
                                                if contains {
                                                    *congestion_path_data_opt_arc
                                                        .lock()
                                                        .unwrap()
                                                        .get_mut(&day)
                                                        .unwrap()
                                                        .get_mut(&period)
                                                        .unwrap()
                                                        .get_mut(
                                                            &(
                                                                room.to_owned(),
previous_room.to_owned(),
                                                            )
                                                        )
                                                        .unwrap() += 1;
                                                } else {
                                                    congestion_path_data_opt_arc
                                                        .lock()
                                                        .unwrap()
                                                        .get_mut(&day)
                                                        .unwrap()
                                                        .get_mut(&period)
                                                        .unwrap()
                                                        .insert(
                                                            (
                                                                room.to_owned(),
previous_room.to_owned(),
                                                            ),
                                                            1
                                                        );
                                                }
                                            }
                                            previous_room = room;
                                            *congestion_point_data_opt_arc
                                                .lock()
                                                .unwrap()
                                                .get_mut(&day)
                                                .unwrap()
                                                .get_mut(&period)
                                                .unwrap()
                                                .get_mut(room)
                                                .unwrap() += 1;
                                            let cur_max_congestion =
*max_congestion_opt_arc
                                                .lock()
                                                .unwrap();
```

```
                                        *max_congestion_opt_arc.lock()
.unwrap() =
                                            cur_max_congestion.max(
*congestion_point_data_opt_arc
                                                    .lock()
                                                    .unwrap()
                                                    .get_mut(&day)
                                                    .unwrap()
                                                    .get_mut(&period)
                                                    .unwrap()
                                                    .get_mut(room)
                                                    .unwrap()
                                            );
                                    }
                                }
                            }
                        }
                        for day in 1..=5 {
                            for period in 0..=11 {
                                for room in &rooms {
                                    let room_congestion =
*congestion_point_data_opt_arc
                                        .lock()
                                        .unwrap()
                                        .get_mut(&day)
                                        .unwrap()
                                        .get_mut(&period)
                                        .unwrap()
                                        .get_mut(room)
                                        .unwrap();
                                    congestion_statistics_opt_arc
                                        .lock()
                                        .unwrap()
                                        .point_count.get_mut(&day)
                                        .unwrap()
                                        .get_mut(&period)
                                        .unwrap()
[congestion_range_index(room_congestion)] += 1;
                                }
                            }
                        }
                        for day in 1..=5 {
                            for period in 0..=11 {
                                congestion_path_data_opt_arc
                                    .lock()
                                    .unwrap()
                                    .get_mut(&day)
                                    .unwrap()
                                    .get_mut(&period)
                                    .unwrap()
                                    .iter_mut()
                                    .for_each(|(_, congestion)| {
                                        congestion_statistics_opt_arc
                                            .lock()
                                            .unwrap()
                                            .path_count.get_mut(&day)
                                            .unwrap()
                                            .get_mut(&period)
```

```
                                        .unwrap()
[congestion_range_index(*congestion)] += 1;
                                    });
                            }
                        }
                    }
                    for day in 1..=5 {
                        for period in 0..=11 {
                            for room in &rooms {
                                congestion_point_data_arc
                                    .lock()
                                    .unwrap()
                                    .get_mut(&day)
                                    .unwrap()
                                    .get_mut(&period)
                                    .unwrap()
                                    .insert(room.to_owned(), 0);
                            }
                        }
                    }
                    for student in &student_routes {
                        for day in 1..=5 {
                            for period in 0..=11 {
                                let rooms = student_routes
                                    .get(student.0)
                                    .unwrap()
                                    .get(&day)
                                    .unwrap()
                                    .get(&period)
                                    .unwrap()
                                    .split(' ')
                                    .collect::<Vec<&str>>();
                                let mut previous_room = "";
                                for room in rooms {
                                    if room.is_empty() || room == "G" {
                                        continue;
                                    }
                                    if !previous_room.is_empty() {
                                        let contains =
congestion_path_data_arc
                                            .lock()
                                            .unwrap()
                                            .get(&day)
                                            .unwrap()
                                            .get(&period)
                                            .unwrap()
                                            .contains_key(
                                                &(previous_room.to_owned(),
room.to_owned())
                                            );
                                        if contains {
                                            *congestion_path_data_arc
                                                .lock()
                                                .unwrap()
                                                .get_mut(&day)
                                                .unwrap()
                                                .get_mut(&period)
                                                .unwrap()
                                                .get_mut(
```

```
                                                    &
(previous_room.to_owned(), room.to_owned())
                                                )
                                                .unwrap() += 1;
                                        } else {
                                            congestion_path_data_arc
                                                .lock()
                                                .unwrap()
                                                .get_mut(&day)
                                                .unwrap()
                                                .get_mut(&period)
                                                .unwrap()
                                                .insert(
                                                    (previous_room.to_owned()
, room.to_owned()),
                                                    1
                                                );
                                        }
                                        let contains =
congestion_path_data_arc
                                            .lock()
                                            .unwrap()
                                            .get(&day)
                                            .unwrap()
                                            .get(&period)
                                            .unwrap()
                                            .contains_key(
                                                &(room.to_owned(),
previous_room.to_owned())
                                            );
                                        if contains {
                                            *congestion_path_data_arc
                                                .lock()
                                                .unwrap()
                                                .get_mut(&day)
                                                .unwrap()
                                                .get_mut(&period)
                                                .unwrap()
                                                .get_mut(
                                                    &(room.to_owned(),
previous_room.to_owned())
                                                )
                                                .unwrap() += 1;
                                        } else {
                                            congestion_path_data_arc
                                                .lock()
                                                .unwrap()
                                                .get_mut(&day)
                                                .unwrap()
                                                .get_mut(&period)
                                                .unwrap()
                                                .insert(
                                                    (room.to_owned(),
previous_room.to_owned()),
                                                    1
                                                );
                                        }
                                    }
                                    previous_room = room;
                                    *congestion_point_data_arc
```

```
                                        .lock()
                                        .unwrap()
                                        .get_mut(&day)
                                        .unwrap()
                                        .get_mut(&period)
                                        .unwrap()
                                        .get_mut(room)
                                        .unwrap() += 1;
                                    let cur_max_congestion =
*max_congestion_arc
                                        .lock()
                                        .unwrap();
                                    *max_congestion_arc.lock().unwrap() =
                                        cur_max_congestion.max(
                                            *congestion_point_data_arc
                                                .lock()
                                                .unwrap()
                                                .get_mut(&day)
                                                .unwrap()
                                                .get_mut(&period)
                                                .unwrap()
                                                .get_mut(room)
                                                .unwrap()
                                        );
                                }
                            }
                        }
                    }
                    for day in 1..=5 {
                        for period in 0..=11 {
                            for room in &rooms {
                                let room_congestion =
*congestion_point_data_arc
                                    .lock()
                                    .unwrap()
                                    .get_mut(&day)
                                    .unwrap()
                                    .get_mut(&period)
                                    .unwrap()
                                    .get_mut(room)
                                    .unwrap();
                                congestion_statistics_arc
                                    .lock()
                                    .unwrap()
                                    .point_count.get_mut(&day)
                                    .unwrap()
                                    .get_mut(&period)
                                    .unwrap()
[congestion_range_index(room_congestion)] += 1;
                            }
                        }
                    }
                    for day in 1..=5 {
                        for period in 0..=11 {
                            congestion_path_data_arc
                                .lock()
                                .unwrap()
                                .get_mut(&day)
                                .unwrap()
```

```
                                    .get_mut(&period)
                                    .unwrap()
                                    .iter_mut()
                                    .for_each(|(_, congestion)| {
                                        congestion_statistics_arc
                                            .lock()
                                            .unwrap()
                                            .path_count.get_mut(&day)
                                            .unwrap()
                                            .get_mut(&period)
                                            .unwrap()
[congestion_range_index(*congestion)] += 1;
                                    });
                            }
                        }
                        *congestion_status_arc.lock().unwrap() =
CongestionStatus::GeneratingPI(
                            0,
                            "Calculating performance indices".to_owned()
                        );
                        let mut performance_indices_shortest = HashMap::new();
                        let path_distances = path_distances_arc.lock().unwrap()
.clone();
                        for day in 1..=5 {
                            performance_indices_shortest.insert(day,
HashMap::new());
                            for period in 0..=11 {
                                let mut index = 0f64;
                                for student in &student_routes {
                                    let rooms = student_routes
                                        .get(student.0)
                                        .unwrap()
                                        .get(&day)
                                        .unwrap()
                                        .get(&period)
                                        .unwrap()
                                        .split(' ')
                                        .collect::<Vec<&str>>();
                                    let mut previous_room = "";
                                    for room in rooms {
                                        if room.is_empty() || room == "G" {
                                            continue;
                                        }
                                        if !previous_room.is_empty() {
                                            let path_distance = *path_distances
                                                .get(previous_room)
                                                .unwrap()
                                                .get(room)
                                                .unwrap_or_else(|| {
                                                    println!(
                                                        "{:?}",

path_distances.get(previous_room).unwrap()
                                                    );
                                                    panic!(
                                                        "Path distance not found:
{} → {}",
                                                        previous_room,
                                                        room
                                                    )
```

```
                                                }) as f64;
                                            let path_congestion =
*congestion_path_data_arc
                                                .lock()
                                                .unwrap()
                                                .get(&day)
                                                .unwrap()
                                                .get(&period)
                                                .unwrap()
                                                .get(&(previous_room.to_owned(),
room.to_owned()))
                                                .unwrap() as f64;
                                            index +=
                                                path_distance *
                                                (2.0 + ((path_congestion - 300.0)
/ 200.0).tanh());
                                        }
                                        previous_room = room;
                                    }
                                }
                                performance_indices_shortest
                                    .get_mut(&day)
                                    .unwrap()
                                    .insert(period, index as u128);
                            }
                        }
                        // println!("{}", serde_json::to_string(&
performance_indices_shortest).unwrap());
                        *performance_indices_shortest_arc.lock().unwrap() =
                            performance_indices_shortest;
                        *congestion_status_arc.lock().unwrap() =
CongestionStatus::Successful;
                    });
                }
                CongestionStatus::GeneratingPI(_, message) => {

ui.with_layout(Layout::top_down_justified(egui::Align::Center), |ui| {
                        ui.label(RichText::new(material_design_icons::MDI_TIMER)
.size(32.0));
                        ui.label(message);
                        ui.spinner();
                    });
                }
                CongestionStatus::Generating(_, message) => {

ui.with_layout(Layout::top_down_justified(egui::Align::Center), |ui| {
                        ui.label(

RichText::new(material_design_icons::MDI_TRAFFIC_LIGHT).size(32.0)
                        );
                        ui.label(message);
                        ui.spinner();
                    });
                }
                CongestionStatus::Failed(message) => {

ui.with_layout(Layout::top_down_justified(egui::Align::Center), |ui| {
                        ui.label(

RichText::new(material_design_icons::MDI_TRAFFIC_LIGHT)
                                .size(32.0)
```

```
                                .color(Color32::from_rgb(0xe4, 0x37, 0x48))
                        );
                        ui.label("Congestion evaluation failed");
                        ui.label(message);
                        if ui.button("Close").clicked() {
                            self.show_congestion_window = false;
                        }
                    });
                }
                CongestionStatus::Successful => {
ui.with_layout(Layout::top_down_justified(egui::Align::Center), |ui| {
                        ui.label(
RichText::new(material_design_icons::MDI_TRAFFIC_LIGHT)
                                .size(32.0)
                                .color(Color32::from_rgb(0x14, 0xae, 0x52))
                        );
                        ui.label("Congestion evaluation successful");
                        ui.label(
                            "The congestion has been successfully evaluated and loaded. You may now view them on the projection."
                        );
                        ui.label(
                            format!(
                                "Maximum congestion: {}",
                                self.maximum_congestion.lock().unwrap()
                            )
                        );
                        if ui.button("Close").clicked() {
                            self.show_congestion_window = false;
                        }
                    });
                }
            }
        });
    }

    fn show_json_validation_window(
        &mut self,
        ctx: &egui::Context,
        current_validation_status: TimetableValidationStatus
    ) {
        Window::new("Timetable Validation").show(ctx, |ui| {
            ui.heading("Metadata");
            Grid::new("timetable_validation_grid")
                .num_columns(2)
                .show(ui, |ui| {
                    ui.label("Filename");
                    ui.label(self.timetable_file_info.filename.to_string());
                    ui.end_row();

                    ui.label("Student count");
                    if
                        let Some(student_count) =
*self.timetable_file_info.student_count
                            .lock()
                            .unwrap()
                    {
```

```
                        ui.label(format!("{}", student_count));
                    } else {
                        ui.label("-");
                    }
                    ui.end_row();

                    ui.label("Session count");
                    if
                        let Some(session_count) =
*self.timetable_file_info.session_count
                            .lock()
                            .unwrap()
                    {
                        ui.label(format!("{}", session_count));
                    } else {
                        ui.label("-");
                    }
                    ui.end_row();
                });
            ui.separator();
            match current_validation_status {
                TimetableValidationStatus::Ready => {
                    *self.timetable_file_info.validation_status.clone().lock()
.unwrap() =
                        TimetableValidationStatus::Validating(0, "Ready to
validate".to_owned());
                    let filepath = self.timetable_file_info.filepath.clone();
                    let projection_coords = self.projection_coords.clone();
                    let validation_status_arc =
self.timetable_file_info.validation_status.clone();
                    let student_count_arc =
self.timetable_file_info.student_count.clone();
                    let session_count_arc =
self.timetable_file_info.session_count.clone();
                    let timetable_arc =
self.timetable_file_info.timetable.clone();
                    let student_list_arc = self.student_list.clone();
                    thread::spawn(move || {
                        let mut rooms: Vec<String> = projection_coords
                            .keys()
                            .map(|k| k.to_owned())
                            .collect();
                        *validation_status_arc.lock().unwrap() =
                            TimetableValidationStatus::Validating(
                                0,
                                "Validating student numbers...".to_owned()
                            );
                        rooms.push("G".into());
                        let mut timetable_file = std::fs::File::open(filepath)
.unwrap();
                        let mut timetable_json = String::new();
                        if timetable_file.read_to_string(&mut timetable_json)
.is_err() {
                            *validation_status_arc.lock().unwrap() =
                                TimetableValidationStatus::Failed(
                                    "Failed to read timetable file".to_owned()
                                );
                            return;
                        }
                        let timetable: serde_json::Value = match
                            serde_json::from_str(&timetable_json)
```

```
                        {
                            Ok(t) => t,
                            Err(e) => {
                                *validation_status_arc.lock().unwrap() =
                                    TimetableValidationStatus::Failed(
                                        format!("Invalid JSON format in timetable file: {}", e)
                                    );
                                return;
                            }
                        };
                        let mut student_count = 0;

                        if timetable.as_object().is_none() {
                            *validation_status_arc.lock().unwrap() =
                                TimetableValidationStatus::Failed(
                                    "Invalid timetable file format: the JSON file is not a map".to_owned()
                                );
                            return;
                        }
                        for student_key in timetable.as_object().unwrap().keys() {
                            if
                                student_key.chars().all(char::is_numeric) &&
                                4 <= student_key.len() &&
                                student_key.len() <= 5
                            {
                                student_count += 1;
                                *student_count_arc.lock().unwrap() = Some(student_count);
                            } else {
                                *validation_status_arc.lock().unwrap() =
                                    TimetableValidationStatus::Failed(
                                        format!("Invalid student number: \"{}\"", student_key)
                                    );
                                return;
                            }
                        }

                        *validation_status_arc.lock().unwrap() =
                            TimetableValidationStatus::Validating(
                                10,
                                "Validating days of the week...".to_owned()
                            );
                        for (student_key, week_timetable) in timetable.as_object().unwrap() {
                            if week_timetable.as_object().is_none() {
                                *validation_status_arc.lock().unwrap() =
                                    TimetableValidationStatus::Failed(
                                        format!("Invalid timetable file format: student {}'s timetable is not a map", student_key)
                                    );
                                return;
                            }
                            let mut days_of_week = [false; 5];
                            for day_key in week_timetable.as_object().unwrap().keys() {
                                if
```

```
                                    day_key.chars().all(char::is_numeric) &&
                                    day_key.parse::<i32>().unwrap() <= 5 &&
                                    day_key.parse::<i32>().unwrap() >= 1
                                {
                                    days_of_week[
                                        (day_key.parse::<i32>().unwrap() as
usize) - 1
                                    ] = true;
                                } else {
                                    *validation_status_arc.lock().unwrap() =
                                        TimetableValidationStatus::Failed(
                                            format!(
                                                "Student {} has an invalid day of
week: \"{}\"",
                                                student_key,
                                                day_key
                                            )
                                        );
                                    return;
                                }
                            }
                            if days_of_week.contains(&false) {
                                *validation_status_arc.lock().unwrap() =
                                    TimetableValidationStatus::Failed(
                                        format!(
                                            "Student {} has an incomplete
timetable: missing day {}",
                                            student_key,
                                            days_of_week
                                                .iter()
                                                .enumerate()
                                                .filter(|(_, &b)| !b)
                                                .map(|(i, _)| i + 1)
                                                .collect::<Vec<usize>>()
                                                .iter()
                                                .map(|i| i.to_string())
                                                .collect::<Vec<String>>()
                                                .join(", ")
                                        )
                                    );
                                return;
                            }
                        }

                        *validation_status_arc.lock().unwrap() =
                            TimetableValidationStatus::Validating(
                                20,
                                "Validating periods...".to_owned()
                            );
                        for (student_key, week_timetable) in
timetable.as_object().unwrap() {
                            for (day_key, day_timetable) in
week_timetable.as_object().unwrap() {
                                if day_timetable.as_object().is_none() {
                                    *validation_status_arc.lock().unwrap() =
                                        TimetableValidationStatus::Failed(
                                            format!(
                                                "Invalid timetable file format:
student {}'s timetable on day {} is not a map",
                                                student_key,
```

```
                                                day_key
                                            )
                                        );
                                    return;
                                }
                                let mut periods = [false; 10];
                                for period_key in day_timetable.as_object()
.unwrap().keys() {
                                    if
                                        period_key.chars().all(char::is_numeric)
&&
                                        period_key.parse::<i32>().unwrap() <= 10
&&
                                        period_key.parse::<i32>().unwrap() >= 1
                                    {
                                        periods[
                                            (period_key.parse::<i32>().unwrap()
as usize) - 1
                                        ] = true;
                                    } else {
                                        *validation_status_arc.lock().unwrap() =
                                            TimetableValidationStatus::Failed(
                                                format!(
                                                    "Student {} has an invalid
period on day {}: \"{}\"",
                                                    student_key,
                                                    day_key,
                                                    period_key
                                                )
                                            );
                                        return;
                                    }
                                }
                                if periods.contains(&false) {
                                    *validation_status_arc.lock().unwrap() =
                                        TimetableValidationStatus::Failed(
                                            format!(
                                                "Student {} has an incomplete
timetable on day {}: missing periods {}",
                                                student_key,
                                                day_key,
                                                periods
                                                    .iter()
                                                    .enumerate()
                                                    .filter(|(_, &b)| !b)
                                                    .map(|(i, _)| i + 1)
                                                    .collect::<Vec<usize>>()
                                                    .iter()
                                                    .map(|i| i.to_string())
                                                    .collect::<Vec<String>>()
                                                    .join(", ")
                                            )
                                        );
                                    return;
                                }
                            }
                        }

                        *validation_status_arc.lock().unwrap() =
                            TimetableValidationStatus::Validating(
```

```
                                20,
                                "Validating classrooms...".to_owned()
                            );
                        let mut sessions = 0;
                        for (student_key, week_timetable) in
timetable.as_object().unwrap() {
                            for (day_key, day_timetable) in
week_timetable.as_object().unwrap() {
                                for (period_key, room) in
day_timetable.as_object().unwrap() {
                                    if !rooms.contains(&room.as_str().unwrap()
.to_owned()) {
                                        *validation_status_arc.lock().unwrap() =
                                            TimetableValidationStatus::Failed(
                                                format!(
                                                    "Student {} has an invalid
classroom on day {} period {}: {}",
                                                    student_key,
                                                    day_key,
                                                    period_key,
                                                    room
                                                )
                                            );
                                        return;
                                    }
                                    sessions += 1;
                                    if sessions % 1000 == 0 {
                                        *session_count_arc.lock().unwrap() =
Some(sessions);
                                        *validation_status_arc.lock().unwrap() =

TimetableValidationStatus::Validating(
                                                20 +
                                                    (
                                                        (((sessions as f32) /
                                                            ((student_count * 10
* 5) as f32)) *
                                                            80.0) as i32
                                                    ),
                                                "Validating classrooms..."
.to_owned()
                                            );
                                    }
                                }
                            }
                        }
                        *timetable_arc.lock().unwrap() = Some(timetable.clone());

                        *student_list_arc.lock().unwrap() = timetable
                            .as_object()
                            .unwrap()
                            .keys()
                            .map(|k| k.to_owned())
                            .collect();

                        *validation_status_arc.lock().unwrap() =
                            TimetableValidationStatus::Successful;
                    });
                }
                TimetableValidationStatus::Validating(progress, message) => {
```

```
1680
ui.with_layout(Layout::top_down_justified(egui::Align::Center), |ui| {
1681                         ui.label(
1682
RichText::new(material_design_icons::MDI_CALENDAR_SEARCH).size(32.0)
1683                         );
1684                         ui.label(message);
1685                         ui.add(ProgressBar::new((progress as f32) / 100.0));
1686                     });
1687                 }
1688                 TimetableValidationStatus::Failed(_) => {
1689
ui.with_layout(Layout::top_down_justified(egui::Align::Center), |ui| {
1690                         ui.label(
1691
RichText::new(material_design_icons::MDI_CALENDAR_REMOVE)
1692                                 .size(32.0)
1693                                 .color(Color32::from_rgb(0xe4, 0x37, 0x48))
1694                         );
1695                         ui.label("Validation failed");
1696                         ui.label(current_validation_status.get_error_message());
1697                         if ui.button("Close").clicked() {
1698                             self.show_json_validation = false;
1699                         }
1700                     });
1701                 }
1702                 TimetableValidationStatus::Successful => {
1703
ui.with_layout(Layout::top_down_justified(egui::Align::Center), |ui| {
1704                         ui.label(
1705
RichText::new(material_design_icons::MDI_CALENDAR_CHECK)
1706                                 .size(32.0)
1707                                 .color(Color32::from_rgb(0x14, 0xae, 0x52))
1708                         );
1709                         ui.label("Validation successful");
1710                         ui.label(
1711                             "The timetable has been imported successfully. You
may now proceed to the next step."
1712                         );
1713                         if ui.button("Close").clicked() {
1714                             self.show_json_validation = false;
1715                         }
1716                     });
1717                 }
1718             }
1719         });
1720     }
1721
1722     fn show_timetable_window(&mut self, ctx: &egui::Context) {
1723         Window::new("Timetable")
1724             .open(&mut self.show_timetable_window)
1725             .show(ctx, |ui| {
1726                 if
1727                     self.timetable_file_info.validation_status.lock().unwrap()
.clone() ==
1728                     TimetableValidationStatus::Successful
1729                 {
1730                     if let Some(student) = &self.selected_student {
1731                         ui.label(format!("{}'s Timetable", student));
```

```
                        egui::Grid
                            ::new("timetable_grid")
                            .striped(true)
                            .show(ui, |ui| {
                                ui.label("Period");
                                ui.label("Monday");
                                ui.label("Tuesday");
                                ui.label("Wednesday");
                                ui.label("Thursday");
                                ui.label("Friday");
                                ui.end_row();

                                let timetable =
self.timetable_file_info.timetable.lock().unwrap();

                                for period in 1..=10 {
                                    ui.label(format!("Period {}", period));
                                    for weekday in 1..=5 {
                                        let mut session = timetable
                                            .as_ref()
                                            .unwrap()
                                            .get(student)
                                            .unwrap()
                                            .get(weekday.to_string())
                                            .unwrap()
                                            .get(period.to_string())
                                            .unwrap()
                                            .as_str()
                                            .unwrap();
                                        if session == "G" {
                                            session = " ";
                                        }
                                        ui.label(session);
                                    }
                                    ui.end_row();
                                }
                            });
                    } else {
                        ui.label("No student selected.");
                    }
                } else {
                    ui.label("Timetable not imported.");
                }
            });
    }

    fn show_pi_window(&mut self, ctx: &egui::Context) {
        egui::Window
            ::new("Performance Indices")
            .open(&mut self.show_pi_window)
            .show(ctx, |ui| {
                let performance_indices_shortest =
self.performance_indices_shortest
                    .lock()
                    .unwrap();
                let performance_indices_optimized =
self.performance_indices_optimized
                    .lock()
                    .unwrap();
```

```
                ui.horizontal(|ui| {
                    if ui.selectable_label(self.show_pi_shortest, "Shortest")
.clicked() {
                        self.show_pi_shortest = true;
                    }

                    if ui.selectable_label(!self.show_pi_shortest, "Optimized")
.clicked() {
                        self.show_pi_shortest = false;
                    }
                });
                ui.separator();
                ui.label("Performance Indices Overview");
                let pi_matrix = if self.show_pi_shortest {
                    performance_indices_shortest
                } else {
                    performance_indices_optimized
                };
                let hightlights = *self.current_period_iter.lock().unwrap();
                let optimization_status = self.optimization_status.lock()
.unwrap().clone();
                Grid::new("shortest_pi_grid")
                    .striped(true)
                    .num_columns(6)
                    .show(ui, |ui| {
                        ui.label("");
                        ui.label("Monday");
                        ui.label("Tuesday");
                        ui.label("Wednesday");
                        ui.label("Thursday");
                        ui.label("Friday");
                        ui.end_row();
                        for index in 0..=11 {
                            ui.label(convert_periods(index));
                            for day in 1..=5 {
                                if
                                    (hightlights[(day - 1) as usize] as usize) ==
index &&
                                    optimization_status ==
OptimizationStatus::Calculating
                                {
                                    ui.label(
                                        RichText::new(
                                            pi_matrix[&day][&index]
.to_formatted_string(&Locale::fr)
                                        ).color(Color32::from_rgb(0xec, 0x6f,
0x27))
                                    );
                                } else {
                                    ui.label(
                                        pi_matrix[&day][&index]
.to_formatted_string(&Locale::fr)
                                    );
                                }
                            }
                            ui.end_row();
                        }
                    });
                ui.separator();
                ui.label("Current Period PI");
```

```
                ui.heading(
                    pi_matrix[&self.selected_day][&self.selected_period]
.to_formatted_string(
                        &Locale::fr
                    )
                );
                ui.separator();
                ui.label("Current Day PI");
                ui.heading(
                    pi_matrix[&self.selected_day]
                        .values()
                        .sum::<u128>()
                        .to_formatted_string(&Locale::fr)
                );
                ui.separator();
                ui.label("Total PI");
                ui.heading(
                    pi_matrix
                        .values()
                        .map(|day| day.values().sum::<u128>())
                        .sum::<u128>()
                        .to_formatted_string(&Locale::fr)
                );
                ui.separator();
                ui.label(
                    "Performance indices measure the overall performance of the
routes. The lower the value, the better the performance."
                );
            });
    }
}

#[allow(dead_code)]
fn run_floyd_algorithm_cpp(
    filepath: PathBuf,
    path_generation_status_arc: Arc<Mutex<PathGenerationStatus>>,
    student_paths_arc: Arc<Mutex<Option<Routes>>>
) -> String {
    let bin_dir = fs::canonicalize("./bin").unwrap();
    if let Ok(binding) = fs::canonicalize(filepath) {
        let json_path = binding.to_str().unwrap();
        println!("{:?}\n{:?}", bin_dir, json_path);
        if
            let Ok(mut floyd_command) = Command::new("./floyd.out")
                .current_dir(&bin_dir)
                .stdin(Stdio::piped())
                .spawn()
        {
            floyd_command.stdin.as_mut().unwrap().write_all(json_path.as_bytes())
.unwrap();
            let output = floyd_command.wait_with_output();
            if output.is_err() {
                *path_generation_status_arc.lock().unwrap() =
PathGenerationStatus::Failed(
                    "Failed to execute algorithm binary [floyd.out].".to_owned()
                );
            } else if output.as_ref().unwrap().status.success() {
                *path_generation_status_arc.lock().unwrap() =
PathGenerationStatus::LoadingJSON;
                let result_path = bin_dir.join("routes.json");
```

```
                let Ok(file_content) = fs::read_to_string(result_path) else {
                    *path_generation_status_arc.lock().unwrap() =
PathGenerationStatus::Failed(
                        "Failed to read result file [routes.json].".to_owned()
                    );
                    return "".to_owned();
                };
                let Ok(routes) = serde_json::from_str::<Routes>(&file_content)
else {
                    *path_generation_status_arc.lock().unwrap() =
PathGenerationStatus::Failed(
                        "Failed to parse result file [routes.json].".to_owned()
                    );
                    return "".to_owned();
                };
                *student_paths_arc.lock().unwrap() = Some(routes);
                *path_generation_status_arc.lock().unwrap() =
PathGenerationStatus::Successful;
                return file_content;
            } else {
                *path_generation_status_arc.lock().unwrap() =
PathGenerationStatus::Failed(
                    format!(
                        "Algorithm binary [floyd.out] exited with code {}.",
                        output.unwrap().status.code().unwrap()
                    )
                );
            }
        } else {
            *path_generation_status_arc.lock().unwrap() =
PathGenerationStatus::Failed(
                "Failed to execute algorithm binary [floyd.out].".to_owned()
            );
        }
    } else {
        *path_generation_status_arc.lock().unwrap() =
PathGenerationStatus::Failed(
            "Failed to find timetable file.".to_owned()
        );
    }
    "".to_owned()
}

#[allow(dead_code)]
fn run_floyd_algorithm_rust(
    filepath: PathBuf,
    shortest_paths_json: HashMap<String, String>,
    path_generation_status_arc: Arc<Mutex<PathGenerationStatus>>,
    student_paths_arc: Arc<Mutex<Option<Routes>>>
) {
    type Timetable = HashMap<String, HashMap<u32, HashMap<usize, String>>>;
    let mut result: Routes = HashMap::new();
    let Ok(timetable_file) = File::open(filepath) else {
        *path_generation_status_arc.lock().unwrap() =
PathGenerationStatus::Failed(
            "Failed to find timetable file.".to_owned()
        );
        return;
    };
    let Ok(timetable_json) = serde_json::from_reader::<_, Timetable>
(timetable_file) else {
```

```
        *path_generation_status_arc.lock().unwrap() = PathGenerationStatus::Failed(
            "Failed to parse timetable file. Please make sure the file is in JSON format.".to_owned()
        );
        return;
    };
    for student_number in timetable_json.keys() {
        let mut student_hashmap: HashMap<u32, HashMap<usize, String>> = HashMap::new();
        let mut student_timetable = timetable_json.get(student_number).unwrap().clone();
        for day in 1..=5 {
            let mut day_hashmap: HashMap<usize, String> = HashMap::new();
            student_timetable.entry(day).or_insert_with(HashMap::new);
            day_hashmap.insert(0, {
                let first_room = student_timetable.get(&day).unwrap().get(&1).unwrap().to_owned();
                if first_room == "G" {
                    "".to_owned()
                } else {
                    shortest_paths_json.get(format!("G{}", first_room).as_str()).unwrap().to_owned()
                }
            });
            for period in 1..=5 {
                day_hashmap.insert(period, {
                    let room_a = student_timetable
                        .get(&day)
                        .unwrap()
                        .get(&period)
                        .unwrap()
                        .to_owned();
                    let room_b = student_timetable
                        .get(&day)
                        .unwrap()
                        .get(&(period + 1))
                        .unwrap()
                        .to_owned();
                    if room_a == room_b {
                        "".to_owned()
                    } else {
                        shortest_paths_json
                            .get(format!("{}{}", room_a, room_b).as_str())
                            .unwrap()
                            .to_owned()
                    }
                });
            }
            day_hashmap.insert(6, {
                let first_room = student_timetable.get(&day).unwrap().get(&6).unwrap().to_owned();
                if first_room == "G" {
                    "".to_owned()
                } else {
                    shortest_paths_json.get(format!("{}G", first_room).as_str()).unwrap().to_owned()
                }
            });
            day_hashmap.insert(7, {
```

```
                let first_room = student_timetable.get(&day).unwrap().get(&7)
.unwrap().to_owned();
                if first_room == "G" {
                    "".to_owned()
                } else {
                    shortest_paths_json.get(format!("G{}", first_room).as_str())
.unwrap().to_owned()
                }
            });
            for period in 7..=9 {
                day_hashmap.insert(period + 1, {
                    let room_a = student_timetable
                        .get(&day)
                        .unwrap()
                        .get(&period)
                        .unwrap()
                        .to_owned();
                    let room_b = student_timetable
                        .get(&day)
                        .unwrap()
                        .get(&(period + 1))
                        .unwrap()
                        .to_owned();
                    if room_a == room_b {
                        "".to_owned()
                    } else {
                        shortest_paths_json
                            .get(format!("{}{}", room_a, room_b).as_str())
                            .unwrap()
                            .to_owned()
                    }
                });
            }
            day_hashmap.insert(11, {
                let first_room = student_timetable.get(&day).unwrap().get(&10)
.unwrap().to_owned();
                if first_room == "G" {
                    "".to_owned()
                } else {
                    shortest_paths_json.get(format!("{}G", first_room).as_str())
.unwrap().to_owned()
                }
            });
            student_hashmap.insert(day, day_hashmap);
        }
        result.insert(student_number.to_owned(), student_hashmap);
    }
    *student_paths_arc.lock().unwrap() = Some(result);
    *path_generation_status_arc.lock().unwrap() =
PathGenerationStatus::Successful;
}

impl eframe::App for OptiWayApp {
    // fn save(&mut self, storage: &mut dyn eframe::Storage) {
    //     eframe::set_value(storage, eframe::APP_KEY, self);
    // }

    fn update(&mut self, ctx: &egui::Context, _frame: &mut eframe::Frame) {
        let current_validation_status =
self.timetable_file_info.validation_status
```

```
            .lock()
            .unwrap()
            .clone();
        let current_congestion_status = self.congestion_status.lock().unwrap()
.clone();
        let current_path_status = self.path_generation_status.lock().unwrap()
.clone();
        let current_optimization_status = self.optimization_status.lock()
.unwrap().clone();
        egui::TopBottomPanel::top("top_panel").show(ctx, |ui| {
            egui::menu::bar(ui, |ui| {
                ui.label("OptiWay");
                egui::warn_if_debug_build(ui);
                ui.separator();
                if current_validation_status ≠
TimetableValidationStatus::Successful {
                    ui.label(material_design_icons::MDI_CALENDAR_ALERT)
.on_hover_text(
                        "Timetable not imported."
                    );
                    ui.separator();
                }
                if self.selected_student.is_none() {
                    ui.label(material_design_icons::MDI_ACCOUNT_ALERT)
.on_hover_text(
                        "No student selected."
                    );
                    ui.separator();
                }
                if self.student_routes_shortest.lock().unwrap().is_none() {
                    ui.label(material_design_icons::MDI_SIGN_DIRECTION_REMOVE)
.on_hover_text(
                        "Path not calculated."
                    );
                    ui.separator();
                }
                if current_congestion_status ≠ CongestionStatus::Successful {
                    ui.label(material_design_icons::MDI_VECTOR_POLYLINE_REMOVE)
.on_hover_text(
                        "Congestion not calculated."
                    );
                    ui.separator();
                }
                ui.with_layout(Layout::right_to_left(egui::Align::Center), |ui| {
                    if self.show_json_validation {
                        ui.label("Validating timetable file");
                    } else if current_path_status.is_generating() {
                        ui.label("Calculating path");
                    } else if current_path_status.is_loading_json() {
                        ui.label("Loading paths");
                    } else if current_congestion_status.is_generating() {
                        ui.label("Evaluating congestion");
                    } else if current_optimization_status ==
OptimizationStatus::Calculating {
                        ui.label("Optimizing routes");
                    } else {
                        ui.label("Ready");
                    }
                });
            });
        });
```

```
2105
2106            egui::SidePanel::right("side_panel").show(ctx, |ui| {
2107                egui::ScrollArea::vertical().show(ui, |ui| {
2108                    ui.with_layout(Layout::top_down_justified(egui::Align::LEFT),
    |ui| {
2109                        ui.heading("Command center").on_hover_text(
2110                            "Follow the order of the buttons from top to bottom to
    complete the process."
2111                        );
2112                        if ui.button("Import timetable").clicked() {
2113                            let file = FileDialog::new().add_filter("JSON", &["json"]
    ).pick_file();
2114                            if let Some(file) = file {
2115                                self.timetable_file_info.filename = file
2116                                    .file_name()
2117                                    .unwrap()
2118                                    .to_str()
2119                                    .unwrap()
2120                                    .to_owned();
2121                                self.timetable_file_info.filepath = file;
2122                                self.show_json_validation = true;
2123                                *self.timetable_file_info.validation_status.lock()
    .unwrap() =
2124                                    TimetableValidationStatus::Ready;
2125                                self.selected_student = None;
2126                            }
2127                        }
2128                        ui.add_enabled_ui(
2129                            current_validation_status ==
    TimetableValidationStatus::Successful,
2130                            |ui| {
2131                                if
2132                                    ui
2133                                        .button("Baseline benchmark")
2134                                        .on_hover_text(
2135                                            "Generate shortest routes for all
    students as a baseline plan."
2136                                        )
2137                                        .on_disabled_hover_text("Import a timetable
    first.")
2138                                        .clicked()
2139                                {
2140                                    self.show_path_gen_window = true;
2141                                    *self.path_generation_status.lock().unwrap() =
2142                                        PathGenerationStatus::Ready;
2143                                }
2144                            }
2145                        );
2146                        ui.add_enabled_ui(
2147                            current_path_status == PathGenerationStatus::Successful,
2148                            |ui| {
2149                                if
2150                                    ui
2151                                        .button("Calculate congestion")
2152                                        .on_disabled_hover_text("Calculate routes
    first.")
2153                                        .clicked()
2154                                {
2155                                    self.show_congestion_window = true;
2156                                    *self.congestion_status.lock().unwrap() =
    CongestionStatus::Ready;
```

```
                            }
                        }
                    );
                    ui.add_enabled_ui(
                        current_congestion_status ==
CongestionStatus::Successful,
                        |ui| {
                            if
                                ui
                                    .button("Optimize routes")
                                    .on_disabled_hover_text("Calculate congestion
first.")
                                    .on_hover_text("Start the optimization
iteration process.")
                                    .clicked()
                            {
                                self.show_optimization_window = true;
                                *self.optimization_status.lock().unwrap() =
                                    OptimizationStatus::ParamInput;
                            }
                        }
                    );
                    ui.separator();
                    ComboBox::from_label("Route display mode")
                        .selected_text(format!("{}", self.path_display))
                        .show_ui(ui, |ui| {
                            ui.selectable_value(
                                &mut self.path_display,
                                PathDisplay::Shortest,
                                "Shortest route"
                            );
                            ui.add_enabled_ui(
                                self.student_routes_optimized.lock().unwrap()
.is_some(),
                                |ui| {
                                    ui.selectable_value(
                                        &mut self.path_display,
                                        PathDisplay::Optimized,
                                        "Optimized route"
                                    );
                                }
                            );
                        });
                    ComboBox::from_label("Student")
                        .selected_text(self.selected_student.clone().unwrap_or("
-".to_owned()))
                        .show_ui(ui, |ui| {
                            ui.add(
                                egui::TextEdit
                                    ::singleline(&mut self.student_number_search)
                                    .hint_text("Search")
                            );
                            ui.separator();
                            for student in self.student_list.lock().unwrap()
.iter() {
                                if student.contains(&self.student_number_search)
{
                                    ui.selectable_value(
                                        &mut self.selected_student,
                                        Some(student.to_owned()),
```

```
                                        student
                                    );
                                }
                            }
                        });
                    ComboBox::from_label("Day of Week")
                        .selected_text(convert_day_of_week(self.selected_day))
                        .show_ui(ui, |ui| {
                            for i in 1..=5 {
                                ui.selectable_value(
                                    &mut self.selected_day,
                                    i,
                                    convert_day_of_week(i)
                                );
                            }
                        });
                    ComboBox::from_label("Period")
                        .selected_text(convert_periods(self.selected_period))
                        .show_ui(ui, |ui| {
                            for i in 0..=11 {
                                ui.selectable_value(
                                    &mut self.selected_period,
                                    i,
                                    convert_periods(i)
                                );
                            }
                        });
                    ui.separator();
                    ui.heading("Panels");
                    if ui.button("Show path as text").clicked() {
                        self.show_path_window = true;
                    }
                    if ui.button("Show timetable").clicked() {
                        self.show_timetable_window = true;
                    }
                    if ui.button("Show performance indices").clicked() {
                        self.show_pi_window = true;
                    }
                    ui.separator();
                    ui.heading("Export");
                    if ui.button("Export shortest routes").clicked() {
                        let file = FileDialog::new().add_filter("JSON", &["json"]
).save_file();
                        if let Some(file) = file {
                            let mut file = File::create(file).unwrap();
                            let student_routes =
self.student_routes_shortest.lock().unwrap();
                            let student_routes = student_routes.as_ref().unwrap()
;
                            let json = serde_json::to_string_pretty(&
student_routes).unwrap();
                            file.write_all(json.as_bytes()).unwrap();
                        }
                    }
                    let enabled = self.student_routes_optimized.lock().unwrap()
.is_some();
                    ui.add_enabled_ui(enabled, |ui| {
                        if ui.button("Export optimized routes").clicked() {
                            let file = FileDialog::new().add_filter("JSON", &["
json"]).save_file();
```

```
                            if let Some(file) = file {
                                let mut file = File::create(file).unwrap();
                                let student_routes = self.student_routes_optimized.lock().unwrap();
                                let student_routes = student_routes.as_ref().unwrap();
                                let json = serde_json::to_string_pretty(&student_routes).unwrap();
                                file.write_all(json.as_bytes()).unwrap();
                            }
                        }
                    });
                    ui.separator();
                    ui.heading("Floor view");
                    ui.horizontal(|ui| {
                        if ui.toggle_value(&mut self.selected_floor[0], "All").clicked() {
                            self.selected_floor_index = 0;
                            for i in 1..=8 {
                                self.selected_floor[i] = false;
                            }
                            if !self.selected_floor.contains(&true) {
                                self.selected_floor[0] = true;
                            }
                        }
                        for i in 2..=8 {
                            if
                                ui
                                    .toggle_value(&mut self.selected_floor[i], format!("{}F", i))
                                    .clicked()
                            {
                                self.selected_floor_index = i;
                                for j in 0..=8 {
                                    if i != j {
                                        self.selected_floor[j] = false;
                                    }
                                }
                                if !self.selected_floor.contains(&true) {
                                    self.selected_floor[i] = true;
                                }
                            }
                        }
                    });
                    ui.add(
                        Slider::new(&mut self.inactive_brightness, 32..=255).text(
                            "Inactive floor brightness"
                        )
                    );
                    ui.separator();
                    ui.horizontal(|ui| {
                        if ui.selectable_label(!self.show_congestion, "Show paths").clicked() {
                            self.show_congestion = false;
                        }
                        if ui.selectable_label(self.show_congestion, "Show congestion").clicked() {
                            self.show_congestion = true;
                        }
```

```
                    });
                    if self.show_congestion {
                        ui.checkbox(&mut self.show_congestion_path, "Show path congestion");
                        ui.checkbox(&mut self.show_congestion_point, "Show node congestion");
                        ui.heading("Legend");
                        egui::Grid
                            ::new("congestion_legend")
                            .num_columns(2)
                            .show(ui, |ui| {
                                ui.label(RichText::new("●").color(congestion_color_scale(0)));
                                ui.label("No students");
                                ui.end_row();
                                ui.label(RichText::new("●").color(congestion_color_scale(1)));
                                ui.label("1-20 students");
                                ui.end_row();
                                ui.label(RichText::new("●").color(congestion_color_scale(21)));
                                ui.label("21-50 students");
                                ui.end_row();
                                ui.label(RichText::new("●").color(congestion_color_scale(51)));
                                ui.label("51-100 students");
                                ui.end_row();
                                ui.label(RichText::new("●").color(congestion_color_scale(101)));
                                ui.label("101-200 students");
                                ui.end_row();
                                ui.label(RichText::new("●").color(congestion_color_scale(201)));
                                ui.label("201-400 students");
                                ui.end_row();
                                ui.label(RichText::new("●").color(congestion_color_scale(401)));
                                ui.label("≥401 students");
                                ui.end_row();
                            });
                        ui.add(
                            Slider::new(&mut self.congestion_filter, 0..=400).text(
                                "Minimum congestion"
                            )
                        );
                    } else {
                        ui.horizontal(|ui| {
                            ui.vertical(|ui| {
                                ui.label("Active path color");
                                if ui.button("Reset").clicked() {
                                    self.active_path_color = Color32::from_rgb(0xec, 0x6f, 0x27);
                                }
                            });
                            color_picker::color_picker_color32(
                                ui,
                                &mut self.active_path_color,
                                color_picker::Alpha::Opaque
                            );
                        });
```

```
                        ui.separator();

                        ui.horizontal(|ui| {
                            ui.vertical(|ui| {
                                ui.label("Inactive path color");
                                if ui.button("Reset").clicked() {
                                    self.inactive_path_color = Color32::from_gray(0x61);
                                }
                            });
                            color_picker::color_picker_color32(
                                ui,
                                &mut self.inactive_path_color,
                                color_picker::Alpha::Opaque
                            );
                        });
                    }
                });
            });
        });

        CentralPanel::default().show(ctx, |ui| {
            if self.show_json_validation {
                self.show_json_validation_window(ctx, current_validation_status);
            }

            if self.show_path_gen_window {
                self.show_path_generation_window(ctx, current_path_status);
            }

            if self.show_timetable_window {
                self.show_timetable_window(ctx);
            }

            if self.show_congestion_window {
                self.show_congestion_window(ctx, current_congestion_status);
            }
            if self.show_pi_window {
                self.show_pi_window(ctx);
            }
            if self.show_optimization_window {
                self.show_optimization_window(ctx, current_optimization_status);
            }

            // Paths

            let path_list: Vec<String> = if self.selected_student.is_some() {
                let student_number = self.selected_student.clone().unwrap();
                let student_routes = if self.path_display == PathDisplay::Optimized {
                    self.student_routes_optimized.lock().unwrap().clone()
                } else {
                    self.student_routes_shortest.lock().unwrap().clone()
                };
                if let Some(student_routes) = student_routes {
                    student_routes[&student_number][&self.selected_day][&self.selected_period]
                        .split(' ')
                        .collect::<Vec<&str>>()
```

```
                        .iter()
                        .map(|s| (*s).to_owned())
                        .collect::<Vec<String>>()
                } else {
                    vec![]
                }
            } else {
                vec![]
            };

            let mut segments: Vec<&[i32; 3]> = vec![];

            for i in &path_list {
                if self.projection_coords.contains_key(i) {
                    segments.push(&self.projection_coords[i]);
                }
            }

            // Paths window
            if self.show_path_window {
                Window::new("Path")
                    .open(&mut self.show_path_window)
                    .show(ctx, |ui| {
                        let mut path_string = String::new();
                        for i in &path_list {
                            path_string.push_str(i);
                            path_string.push_str(" → ");
                        }
                        path_string.pop();
                        path_string.pop();
                        path_string.pop();
                        ui.label(path_string);
                    });
            }

            // Import textures if uninitialized

            let mut textures: Vec<TextureHandle> = Vec::new();
            for i in 2..=8 {
                let texture_cur: &TextureHandle = self.textures[i]
.get_or_insert_with(|| {
                    ui.ctx().load_texture(
                        format!("texture-floor-projection-{i}F"),
                        load_image_from_path(
                            Path::new(
                                format!(
                                    "assets/projection-transparent/projection_{i}
F.png"
                                ).as_str()
                            )
                        ).unwrap(),
                        Default::default()
                    )
                });
                textures.push(texture_cur.clone());
            }
            let desired_size = ui.available_size_before_wrap();
            if desired_size.y < (desired_size.x / 2243.0) * (1221.0 + 350.0) {
```

```
                ui.label("▲ There may not be enough space to display the floor plan.");
            }
            let (_id, rect) = ui.allocate_space(desired_size);
            let scale = rect.width() / 2243.0;

            // Paint floor projections

            let current_floor_z = if self.selected_floor_index == 0 {
                0
            } else {
                ((self.selected_floor_index - 2) * 50) as i32
            };

            if !self.show_congestion {
                for (i, point) in segments.iter().enumerate() {
                    if i != 0 {
                        ui.painter().circle_filled(convert_pos(&rect, point, scale), 4.0, if
                            self.selected_floor_index == 0 ||
                            (current_floor_z >= point[2].min(segments[i - 1][2]) &&
                                current_floor_z <= point[2].max(segments[i - 1][2]))
                        {
                            self.active_path_color
                        } else {
                            self.inactive_path_color
                        });
                        ui.painter().line_segment(
                            [
                                convert_pos(&rect, segments[i - 1], scale),
                                convert_pos(&rect, point, scale),
                            ],
                            if
                                self.selected_floor_index == 0 ||
                                (current_floor_z >= point[2].min(segments[i - 1][2]) &&
                                    current_floor_z <= point[2].max(segments[i - 1][2]))
                            {
                                Stroke::new(4.0, self.active_path_color)
                            } else {
                                Stroke::new(4.0, self.inactive_path_color)
                            }
                        );
                    } else {
                        ui.painter().circle_filled(convert_pos(&rect, point, scale), 4.0, if
                            self.selected_floor_index == 0 ||
                            current_floor_z == point[2].min(segments[i][2])
                        {
                            self.active_path_color
                        } else {
                            self.inactive_path_color
                        });
                    }
                }
            } else {
                if self.show_congestion_path {
```

```
                    if
                        self.path_display == PathDisplay::Optimized &&
                        self.student_routes_optimized.lock().unwrap().is_some()
                    {
                        for ((node1, node2), congestion) in
self.congestion_path_data_opt
                            .lock()
                            .unwrap()
                            .clone()
                            .get(&self.selected_day)
                            .unwrap()
                            .get(&self.selected_period)
                            .unwrap() {
                            if node1 == "G" || node2 == "G" || congestion < &
self.congestion_filter {
                                continue;
                            }
                            let node1_pos = self.projection_coords[node1];
                            let node2_pos = self.projection_coords[node2];
                            if
                                self.selected_floor_index == 0 ||
                                (current_floor_z >= node1_pos[2].min(node2_pos[2]
) &&
                                    current_floor_z <= node1_pos[2]
.max(node2_pos[2]))
                            {
                                ui.painter().line_segment(
                                    [
                                        convert_pos(&rect, &node1_pos, scale),
                                        convert_pos(&rect, &node2_pos, scale),
                                    ],
                                    Stroke::new(4.0,
congestion_color_scale(*congestion))
                                );
                            }
                        }
                    } else {
                        for ((node1, node2), congestion) in
self.congestion_path_data
                            .lock()
                            .unwrap()
                            .clone()
                            .get(&self.selected_day)
                            .unwrap()
                            .get(&self.selected_period)
                            .unwrap() {
                            if node1 == "G" || node2 == "G" || congestion < &
self.congestion_filter {
                                continue;
                            }
                            let node1_pos = self.projection_coords[node1];
                            let node2_pos = self.projection_coords[node2];
                            if
                                self.selected_floor_index == 0 ||
                                (current_floor_z >= node1_pos[2].min(node2_pos[2]
) &&
                                    current_floor_z <= node1_pos[2]
.max(node2_pos[2]))
                            {
                                ui.painter().line_segment(
                                    [
```

```
                                        convert_pos(&rect, &node1_pos, scale),
                                        convert_pos(&rect, &node2_pos, scale),
                                    ],
                                    Stroke::new(4.0,
congestion_color_scale(*congestion))
                                );
                            }
                        }
                    }
                }
                if self.show_congestion_point {
                    if
                        self.path_display == PathDisplay::Optimized &&
                        self.student_routes_optimized.lock().unwrap().is_some()
                    {
                        for (room, congestion) in self.congestion_point_data_opt
                            .lock()
                            .unwrap()
                            .clone()
                            .get(&self.selected_day)
                            .unwrap()
                            .get(&self.selected_period)
                            .unwrap() {
                            if room == "G" || room.is_empty() {
                                continue;
                            }
                            let coords = self.projection_coords.get(room)
.unwrap();
                            if
                                (self.selected_floor_index == 0 ||
current_floor_z == coords[2]) &&
                                *congestion >= self.congestion_filter
                            {
                                ui.painter().circle_filled(
                                    convert_pos(&rect, coords, scale),
                                    4.0,
                                    congestion_color_scale(*congestion)
                                );
                            }
                        }
                    } else {
                        for (room, congestion) in self.congestion_point_data
                            .lock()
                            .unwrap()
                            .clone()
                            .get(&self.selected_day)
                            .unwrap()
                            .get(&self.selected_period)
                            .unwrap() {
                            if room == "G" || room.is_empty() {
                                continue;
                            }
                            let coords = self.projection_coords.get(room)
.unwrap();
                            if
                                (self.selected_floor_index == 0 ||
current_floor_z == coords[2]) &&
                                *congestion >= self.congestion_filter
                            {
                                ui.painter().circle_filled(
```

```
                                    convert_pos(&rect, coords, scale),
                                    4.0,
                                    congestion_color_scale(*congestion)
                                );
                            }
                        }
                    }
                }

                for (i, texture) in textures.iter().enumerate().take(7) {
                    let rect = Rect::from_min_size(
                        rect.min,
                        emath::vec2(rect.width(), rect.width() /
texture.aspect_ratio())
                    ).translate(emath::vec2(0.0, ((7 - i) as f32) * 50.0 * scale));

                    ui.painter().image(
                        texture.into(),
                        rect,
                        Rect::from_min_max(pos2(0.0, 0.0), pos2(1.0, 1.0)),
                        if self.selected_floor[0] || self.selected_floor[i + 2] {
                            Color32::WHITE
                        } else {
                            Color32::from_gray(self.inactive_brightness)
                        }
                    );
                }
                // Special case: the floor is selected, so needs to be repainted last
                if self.selected_floor_index != 0 {
                    let texture = textures[self.selected_floor_index - 2].clone();
                    let rect = Rect::from_min_size(
                        rect.min,
                        emath::vec2(rect.width(), rect.width() /
texture.aspect_ratio())
                    ).translate(
                        emath::vec2(0.0, ((7 - (self.selected_floor_index - 2)) as
f32) * 50.0 * scale)
                    );

                    ui.painter().image(
                        (&texture).into(),
                        rect,
                        Rect::from_min_max(pos2(0.0, 0.0), pos2(1.0, 1.0)),
                        Color32::WHITE
                    );
                }
            });
        }
}

fn load_image_from_path(path: &Path) -> Result<ColorImage, image::ImageError> {
    let image = image::io::Reader::open(path)?.decode()?;
    let size = [image.width() as _, image.height() as _];
    let image_buffer = image.to_rgba8();
    let pixels = image_buffer.as_flat_samples();
    Ok(ColorImage::from_rgba_unmultiplied(size, pixels.as_slice()))
}

```

```rust
/// Converts 3D coordinates in projection-coords.yaml to 2D coordinates on screen.
fn convert_pos(rect: &Rect, pos: &[i32; 3], scale: f32) -> emath::Pos2 {
    /// Projection angle of the floor plan (radians)
    const ANGLE: f32 = PI / 6.0;

    (
        emath::vec2(
            rect.left() + 25.0 * ANGLE.cos() * scale,
            rect.top() + (50.0 + 350.0 + 25.0 * ANGLE.sin()) * scale
        ) +
        emath::vec2(
            ((pos[0] as f32) * ANGLE.cos() + (pos[1] as f32) * ANGLE.cos()) * scale,
            ((pos[0] as f32) * ANGLE.sin() - (pos[1] as f32) * ANGLE.sin() - (pos[2] as f32)) *
                scale
        )
    ).to_pos2()
}

fn convert_day_of_week(day: u32) -> String {
    (
        match day {
            1 => "Monday",
            2 => "Tuesday",
            3 => "Wednesday",
            4 => "Thursday",
            5 => "Friday",
            _ => "Unknown",
        }
    ).into()
}

fn convert_periods(index: usize) -> String {
    (
        match index {
            0 => "Before P1",
            1 => "P1-P2",
            2 => "P2-P3",
            3 => "P3-P4",
            4 => "P4-P5",
            5 => "P5-P6",
            6 => "P6-Lunch",
            7 => "Lunch-P7",
            8 => "P7-P8",
            9 => "P8-P9",
            10 => "P9-P10",
            11 => "After P10",
            _ => "Unknown",
        }
    ).into()
}
fn congestion_color_scale(congestion: u32) -> Color32 {
    match congestion {
        0 => Color32::from_rgb(0x61, 0x61, 0x61),
        1..=20 => Color32::from_rgb(0x00, 0x7a, 0xf5),
        21..=50 => Color32::from_rgb(0x14, 0xae, 0x52),
        51..=100 => Color32::from_rgb(0xff, 0xc1, 0x07),
```

```
        101..=200 => Color32::from_rgb(0xec, 0x6f, 0x27),
        201..=400 => Color32::from_rgb(0xe4, 0x37, 0x48),
        _ => Color32::from_rgb(0x91, 0x54, 0xff),
    }
}

fn congestion_range_index(congestion: u32) -> usize {
    match congestion {
        0 => 0,
        1..=20 => 1,
        21..=50 => 2,
        51..=100 => 3,
        101..=200 => 4,
        201..=400 => 5,
        _ => 6,
    }
}

```

optiway/src/lib.rs

```rust
#![warn(clippy::all, rust_2018_idioms)]

mod app;
pub use app::OptiWayApp;
mod app_init;
pub use app_init::{ setup_custom_fonts, setup_custom_styles };
pub mod md_icons;

```

optiway/src/main.rs

```rust
#![warn(clippy::all, rust_2018_idioms)]
#![cfg_attr(not(debug_assertions), windows_subsystem = "windows")]

#[cfg(compile_env = "bundle")]
use std::env;

#[cfg(not(target_arch = "wasm32"))]
fn main() -> eframe::Result<()> {
    #[cfg(compile_env = "bundle")]
    {
        env::set_current_dir(env::current_exe().unwrap().parent().unwrap())
.unwrap();
    }
    env_logger::init();

    let native_options = eframe::NativeOptions::default();
    eframe::run_native(
        "OptiWay",
        native_options,
        Box::new(|cc| {
            // egui_extras::install_image_loaders(&cc.egui_ctx);
            Box::new(optiway::OptiWayApp::new(cc))
        })
    )
}

#[cfg(target_arch = "wasm32")]
fn main() {
    eframe::WebLogger::init(log::LevelFilter::Debug).ok();

    let web_options = eframe::WebOptions::default();

    wasm_bindgen_futures::spawn_local(async {
        eframe::WebRunner
            ::new()
            .start(
                "the_canvas_id",
                web_options,
                Box::new(|cc| Box::new(optiway::OptiWayApp::new(cc)))
            ).await
            .expect("failed to start eframe");
    });
}

```

optiway/build.rs

```rust
fn main() {
    let env = std::env::var("OPTIWAY_COMPILE_ENV").unwrap_or("default".to_string());
    println!("cargo:rerun-if-env-changed=OPTIWAY_COMPILE_ENV");
    println!("cargo:rustc-cfg=compile_env=\"{}\"", env);
}

```

optiway/Cargo.toml

```toml
[package]
name = "optiway"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
egui = "0.22.0"
eframe = { version = "0.22.0", default-features = false, features = [
    # "accesskit",     # Make egui comptaible with screen readers. NOTE: adds a lot of dependencies.
    "default_fonts", # Embed the default egui fonts.
    "glow",          # Use the glow rendering backend. Alternative: "wgpu".
    # "persistence",   # Enable restoring app state when restarting the app.
] }
log = "0.4"
rfd = "0.12.0"
egui_extras = { version = "0.22.0" }
image = { version = "0.24", default-features = false, features = [
    "jpeg",
    "png",
] }
serde = { version = "1.0", features = ["derive"] }
serde_yaml = "0.8"
serde_json = "1.0"
num-format = "0.4.4"

# serde = { version = "1", features = ["derive"] }

[target.'cfg(not(target_arch = "wasm32"))'.dependencies]
env_logger = "0.10"

[target.'cfg(target_arch = "wasm32")'.dependencies]
wasm-bindgen-futures = "0.4"

[profile.release]
opt-level = 2

[profile.dev.package."*"]
opt-level = 2

```

optiway/Makefile.toml

```toml
[env]
G_PLUS_PLUS = "g++"

[tasks.floyd-build]
command = "${G_PLUS_PLUS}"
args = [
    "../multi_agent_path_finding/floyd.cpp",
    "-o",
    "bin/floyd.out",
    "-std=c++17",
    "-O0",
    "-Xlinker",
    "-ld_classic",
]

# Apple uses their new linker in macOS Sonoma 14... unfortuantely the linker crashes in a lot of cases (see https://developer.apple.com/forums/thread/737707)
# So we use the old linker for now. ["-Xlinker", "-ld_classic"]

[tasks.floyd-release-build]
command = "${G_PLUS_PLUS}"
args = [
    "../multi_agent_path_finding/floyd.cpp",
    "-o",
    "bin/floyd.out",
    "-std=c++17",
    "-O3",
    "-Xlinker",
    "-ld_classic",
]

[tasks.floyd-release-build-x64]
command = "${G_PLUS_PLUS}"
args = [
    "../multi_agent_path_finding/floyd.cpp",
    "-o",
    "bin/floyd.out",
    "-std=c++17",
    "-O3",
    "-arch",
    "x86_64",
    "-Xlinker",
    "-ld_classic",
]

[tasks.optimization-build]
command = "${G_PLUS_PLUS}"
args = [
    "../multi_agent_path_finding/multi-objective-agent.cpp",
    "-o",
    "bin/optimization.out",
    "-std=c++17",
    "-O0",
    "-Xlinker",
    "-ld_classic",
]
```

```toml

[tasks.optimization-release-build]
command = "${G_PLUS_PLUS}"
args = [
    "../multi_agent_path_finding/multi-objective-agent.cpp",
    "-o",
    "bin/optimization.out",
    "-std=c++17",
    "-O3",
    "-Xlinker",
    "-ld_classic",
]

[tasks.optimization-release-build-x64]
command = "${G_PLUS_PLUS}"
args = [
    "../multi_agent_path_finding/multi-objective-agent.cpp",
    "-o",
    "bin/optimization.out",
    "-std=c++17",
    "-O3",
    "-arch",
    "x86_64",
    "-Xlinker",
    "-ld_classic",
]

[tasks.entry-run]
command = "cargo"
args = ["run"]

[tasks.entry-build]
command = "cargo"
args = ["build"]

[tasks.entry-release-build]
command = "cargo"
args = ["build", "--release"]

[tasks.entry-release-run]
command = "cargo"
args = ["run", "--release"]

[tasks.entry-bundle-build]
env = { OPTIWAY_COMPILE_ENV = "bundle" }
command = "cargo"
args = ["build", "--release", "-Z", "unstable-options", "--out-dir", "bundle/"]

[tasks.entry-bundle-build-x64]
env = { OPTIWAY_COMPILE_ENV = "bundle" }
command = "cargo"
args = [
    "build",
    "--release",
    "-Z",
    "unstable-options",
    "--out-dir",
    "bundle/",
```

```
    "--target",
    "x86_64-apple-darwin",
]

[tasks.bundle-clean-dirs]
script_runner = "python"
script_extension = "py"
script = '''
import shutil
import os
for folder in ['bundle', 'bundle/bin', 'bundle/assets']:
    if not os.path.exists(folder):
        os.makedirs(folder)
    else:
        shutil.rmtree(folder)
        os.makedirs(folder)
'''

[tasks.bundle-script]
script_runner = "python"
script_extension = "py"
script = '''
import shutil
import os
shutil.copyfile('bin/floyd.out', 'bundle/bin/floyd.out')
shutil.copyfile('bin/optimization.out', 'bundle/bin/optimization.out')

for root, dirs, files in os.walk('assets'):
    for file in files:
        if file ≠ '.DS_Store':
            if not os.path.exists(os.path.join('bundle', root)):
                os.makedirs(os.path.join('bundle', root))
            shutil.copyfile(os.path.join(root, file), os.path.join('bundle',
os.path.join(root, file)))

os.remove('bundle/liboptiway.rlib')
'''

[tasks.bundle-chmod]
command = "chmod"
args = ["-R", "+x", "bundle/"]

[tasks.bundle]
dependencies = [
    "bundle-clean-dirs",
    "floyd-release-build",
    "optimization-release-build",
    "entry-bundle-build",
    "bundle-script",
    "bundle-chmod",
]

[tasks.bundle-x64]
dependencies = [
    "bundle-clean-dirs",
    "floyd-release-build-x64",
    "optimization-release-build-x64",
    "entry-bundle-build-x64",
    "bundle-script",
```

```
    "bundle-chmod",
]

[tasks.run]
dependencies = ["floyd-build", "optimization-build", "entry-run"]

[tasks.build]
dependencies = ["floyd-build", "optimization-build", "entry-build"]

[tasks.release]
dependencies = [
    "floyd-release-build",
    "optimization-release-build",
    "entry-release-run",
]

```

optiway/rustfmt.toml

```
max_width = 100
```

scripts/coord-flatten.py

```python
# coord-flatten.py
# Created by Michael Zhang

# This script flattens projection-coords.yaml file.


import yaml

with open("assets/projection-coords.yaml", "r") as f:
    try:
        source = yaml.safe_load(f)
    except yaml.YAMLError as e:
        print(e)

result = {}

for floor_key in source:
    for side_key in source[floor_key]:
        if source[floor_key][side_key] != None:
            for i in source[floor_key][side_key]:
                result[i] = source[floor_key][side_key][i]

with open("assets/projection-coords-flatten.yaml", "w") as f:
    yaml.safe_dump(result, f)

```

scripts/coord-generator.py

```python
# coord-generator.py
# Created by Michael Zhang

# This script generates the x-coordinates of the nodes
# by calculating the midpoint using the rooms' lengths.

A_BLOCK = False
FIRST_OFFSET = 540
LAST_OFFSET = 40

rooms = []
lengths = []
with open('assets/projection-length.yaml', 'r') as f:
    for line in f.readlines():
        room, length = line.split(': ')
        length = int(length)
        rooms.append(room)
        lengths.append(length)


if A_BLOCK:
    print("A block. Total length:", sum(lengths) + FIRST_OFFSET)
    if FIRST_OFFSET != 0:
        print(
            f"\033[1;34mNOTE: The first node offset is set to {FIRST_OFFSET}.\033[0m")
    if sum(lengths) + FIRST_OFFSET != 2290:
        print("\033[1;33mWARNING: Total length is not 2290!\033[0m")
    for i in rooms:
        if i[0] == 'B':
            print(
                "\033[1;33mWARNING: Using A block mode, but data contains B block rooms!\033[0m")
            break
    for i in range(len(rooms)):
        print(
            f'{rooms[i]}: {FIRST_OFFSET+int(sum(lengths[:i]) + lengths[i] / 2)}')
else:
    print("B block. Total length:", sum(lengths))
    if LAST_OFFSET != 0:
        print(
            f"\033[1;34mNOTE: The last node offset is set to {LAST_OFFSET}.\033[0m")
    for i in rooms:
        if i[0] == 'A':
            print(
                "\033[1;33mWARNING: Using B block mode, but data contains A block rooms!\033[0m")
            break
    for i in range(len(rooms)):
        print(
            f'{rooms[i]}: {2290-LAST_OFFSET-int(sum(lengths[:i]) + lengths[i] / 2)}')

```

scripts/crossing-paths.py

```python
# coord-paths.py
# Created by Michael Zhang

import yaml

with open("assets/projection-coords-flatten.yaml", "r") as f:
    try:
        source = yaml.safe_load(f)
    except yaml.YAMLError as e:
        print(e)

room_1 = input("Enter room 1   >>> ")
crossing = input("Enter crossing >>> ")
room_2 = input("Enter room 2   >>> ")

if room_1 not in source:
    print(f"Room {room_1} not found!")
    exit(1)
if room_2 not in source:
    print(f"Room {room_2} not found!")
    exit(1)
if crossing not in source:
    print(f"Crossing {crossing} not found!")
    exit(1)

x1 = source[room_1][0]
x2 = source[room_2][0]
xc = source[crossing][0]

print("  - dist:", abs(x1 - xc))
print("    nodes:")
print("    -", room_1)
print("    -", crossing)
print("    type: 0")
print("  - dist:", abs(x2 - xc))
print("    nodes:")
print("    -", crossing)
print("    -", room_2)
print("    type: 0")

```

scripts/path-converter.py

```python
# path-converter.py
# Created by Michael Zhang

import yaml

with open("assets/paths.yaml", "r") as f:
    try:
        source = yaml.safe_load(f)
    except yaml.YAMLError as e:
        print(e)

for floor_key in source:
    for side_key in source[floor_key]:
        if source[floor_key][side_key] =≠ None:
            for i in source[floor_key][side_key]:
                print(i["nodes"][0], i["nodes"][1], i["dist"], i["type"])

print("""S2-1 G 999999 2
S2-3 G 999999 2
S2-4 G 999999 2
S2-5 G 999999 2
S2-6 G 999999 2
S2-7 G 999999 2""")

```

scripts/path-generator.py

```python
# path-generator.py
# Created by Michael Zhang

# This script generates paths on the same floor on one block
# using data from assets/projection-coords.yaml

# Path types:
# 0: Normal path between two rooms / between a room and one end of a bridge
# 1: Bridge between two buildings
# 2: Normal staircases (e.g. S2-3)
# 3: Spiral staircases between two floors only (e.g. S2-2)
# 4: Other types of staircases (e.g. S4-6)

import yaml  # well yeah I just love YAML ;)

result = {}

with open("assets/projection-coords.yaml", "r") as f:
    try:
        source = yaml.safe_load(f)
    except yaml.YAMLError as e:
        print(e)

for floor_key in source:
    result[floor_key] = {}
    for block in ["A", "B"]:
        result[floor_key][block] = []
        for room_idx in range(len(source[floor_key][block].keys()) - 1):
            room_keys = list(source[floor_key][block].keys())
            x1 = source[floor_key][block][room_keys[room_idx]][0]
            x2 = source[floor_key][block][room_keys[room_idx + 1]][0]
            dist = abs(x2 - x1)
            result[floor_key][block].append(
                {"nodes": (room_keys[room_idx], room_keys[room_idx+1]), "dist": dist, "type": 0})

try:
    with open("assets/paths.yaml", "r") as f:
        response = input("assets/paths.yaml already exists. Overwrite? (y/n) ")
        if response == "y":
            print("Overwriting assets/paths.yaml...")
        else:
            print("Aborted.")
            exit()
except FileNotFoundError:
    pass

with open("assets/paths.yaml", "w") as f:
    yaml.safe_dump(result, f)

```

timetable_generation/timetable_generator.py

```
"""
Return File Structure (JSON):
{
    student_number1 (string): {
        day (string): {
            period (string): room (string)
        }
        ...
    }
}

Example:
{
    "21447": {
        "1": {
            "1": "531",
            "2": "531",
            "3": "430",
            "4": "430"
            ...
        },
        "2": {...},
        ...
    },
    "21448": {
        "1": {
            "1": "210",
            "2": "210",
            "3": "322",
            "4": "322"
            ...
        },
        "2": {...},
        ...
    }
}
"""

import random
import copy
import json


class TimetableGenerator:
    def __init__(self):
        # Init Data #
        # the frequency for elective classes
        self.glevel_elective_freq = {
            "Extra": 12,
            "Bus": 18,
            "Hum": 14,
            "Enr": 6,
            "Lan": 5,
            "Sci": 25
        }
        # the dict to map class to floor number
```

```
        self.floor_map = {
            "Extra": 1,
            "Bus": 2,
            "Hum": 3,
            "Enr": 4,
            "Lan": 5,
            "Chi": 5,
            "Eng": 5,
            # does not account for sci as it covers three floors
        }
        # all the available classrooms
        self.avail_rooms = {
            2: set(range(201, 242)) - {203, 209, 213, 218, 229, 230, 232, 234, 236, 237, 238, 240},
            3: set(range(301, 334)) - {303, 304, 309, 317, 321, 322, 328},
            4: set(range(401, 434)) - {403, 404, 412, 413, 418, 420, 422, 428},
            5: set(range(501, 535)) - {503, 512, 517, 518, 520, 527, 528, 529},
            6: set(range(601, 632)) - {603, 614, 615, 620, 623, 625, 626, 627},
            7: set(range(701, 734)) - {703, 714, 716, 720, 723, 725, 727, 729, 730, 732},
            8: set(range(801, 822)) - {808, 810, 813, 816, 818, 820}
        }
        # the dicts to map glevel schedule set to period number
        # formate: "<day><1st period><2nd period>"
        glevel_core_set_map = {
            0: ["1_7_8", "4_9_10"],
            1: ["1_9_10", "3_3_4"],
            2: ["2_1_2", "5_3_4"],
            3: ["2_3_4", "5_7_8"]
        }
        glevel_elect_set_map = {
            0: ["1_1_2", "4_3_4"],
            1: ["1_3_4", "2_9_10"],
            2: ["2_7_8", "4_7_8"],
            3: ["3_1_2", "5_1_2"]
        }
        self.glevel_set_map = [glevel_core_set_map, glevel_elect_set_map]
        # the currently filled rooms
        self.filled_rooms = [
            dict(zip(range(2, 9), [set() for _ in range(2, 9)])) for _ in range(9)
]
        self.time_table = {}

    # map rooms into different buildings
    @staticmethod
    def map_building(room):
        if 200 < room < 221 or 300 < room < 320 or 400 < room < 421 or 500 < room < 521 or 600 < room < 618 or 700 < room < 717 or 800 < room < 811:
            return "A"
        else:
            return "B"

    def generate_g_level(self, grade):
        """
        Generate G-level classes
        1: Find the total number of classes for each of the categories: Extra(1F), Bus(2F), Hum(3F), Enr(4F), Lan(5F), Sci(6F)
            1.1 assume no 1-period classes → 380 2-period classes per week → 360 non-PE 2-period classes (500 / 25 = 20)
            1.2 180 non-repetitive non-PE courses (1 course = 2 classes per week)
```

```
            1.3 100 non-repetitive elective courses (core courses: chinese, eng, math, chem) → true as everyone has 5 electives
                1.3.1 80 clean extra courses (there are one more 2-period english and enrichment class)
        2: distribute 80 courses into Extra, Bus, Hum, Enr, Lan, Sci following common-sense weightings (dict "elective_freq")
            2.1 Extra(art/drama/pe/music): 12
            2.2 Bus (econ/bus): 18
            2.3 Hum (geo+his+gp): 14
            2.4 Enr (cs): 6
            2.5 Lan (fren/span/ja): 5
            2.6 Sci (bio+phy): 25
        3: distribute timeslots for each course
           3.1 assign fixed scheduling
                - SCIE has fixed scheduling - it separates classes into different timeslot couples
                   e.g, courses at Monday P1 would repeat at Wednesday P1
                - 19 2-periods in a week, 18 non-PE, 2-periods in a week, 9 schedule sets,
                - 8 clean schedule sets (without enrichment and another eng)
                - schedule sets for core and electives are separated
                - hence, distribute each class into these sets
            3.2 define schedule sets
                - 4 core schedule sets (Enr, Chi, Eng, Sci)
                    1. Mon78, Thu90
                    2. Mon90, Wed34
                    3. Tue12, Fri34
                    4. Tue34, Fri78
                - 4 elective schedule sets (Extra, Bus, Hum, Lan)
                    1: Mon12, Thu34
                    2. Mon34, Tue90
                    3. Tue78, Thu78
                    4. Wed12, Fri12
        4: distribute the rooms for each course
            4.1 the rooms available for each floor is stored in the dict "avail_rooms"
            4.2 limitations: courses at each scheduling set should not have the same room
        5: distribute each student's list of courses taken
        6: project back to generate the timetable

        :param: grade: either G1 or G2
        :return: timetable as a JSON
        """
        # Procedures #
        # 3 distribute timeslots for each course, each slot only needs 20 courses
        core_slots = [[] for _ in range(4)]
        # assign the core classes
        for core in range(20):
            core_classes = ["Enr", "Chi", "Eng", "Sci"]
            for i in range(4):
                core_class = random.choice(core_classes)
                core_slots[i].append(core_class + "_core_" + str(core))
                core_classes.remove(core_class)
        # assign the elective classes
        elect_slots = [[] for _ in range(4)]
        avail_slots = list(range(4))
        for k, v in self.glevel_elective_freq.items():
            for i in range(v):
                slot = random.choice(avail_slots)
```

```
                elect_slots[slot].append(k + "_elect_" + str(i))
                if len(elect_slots[slot]) == 20:
                    avail_slots.remove(slot)

        # 4 distribute the rooms for each course
        if grade == "G1":
            slots = core_slots + elect_slots
        else:
            slots = elect_slots + core_slots
        rooms = {}
        filled_rooms = []  # track the filled rooms for each floor for each slot
        for index, slot in enumerate(slots):
            filled_room = dict(zip(range(2, 9), [set() for _ in range(2, 9)]))
            for course in slot:
                if course[:3] == "Sci":  # sci courses are distributed to three floors
                    # TODO: change to greedy floor
                    floor = random.choice([6, 7, 8])
                else:
                    floor = self.floor_map[course.split("_")[0]]
                    if floor == 1:  # if it is extra, just assign G
                        rooms[course] = "G"
                        continue
                # divide by the current filled rooms, use mod because the core slots for g1 are elective slots for g2
                room = random.choice(list(
                    self.avail_rooms[floor] - filled_room[floor] - self.filled_rooms[(index + 4) % 8][floor]))
                rooms[course] = self.map_building(room) + str(room)
                filled_room[floor].add(room)
            filled_rooms.append(filled_room)

        # update the global filled_rooms
        for i in range(8):
            for j in range(2, 9):
                self.filled_rooms[i][j].update(filled_rooms[i][j])
                # [514, 519, 521, 523, 530, 531, 532, 533, 534, 502, 504, 508, 511]

        # 5 distribute each student's list of courses taken
        if grade == "G1":
            offset = 23000
        else:
            offset = 22000
        students = dict(zip(range(offset+1, offset+501),
                            [[] for _ in range(500)]))
        # assign each student to a core class (#0 - 19)
        # track the number of students in each core class
        core_counts = [0 for _ in range(20)]
        cores = list(range(20))
        for student in students:
            core = random.choice(cores)
            students[student].extend(
                [f"Enr_core_{core}", f"Chi_core_{core}", f"Eng_core_{core}", f"Sci_core_{core}"])
            core_counts[core] += 1
            if core_counts[core] == 25:
                cores.remove(core)

        elect_slots_copy = copy.deepcopy(elect_slots)
```

```
        # assign each student to an elective class
        # track the filled courses for each slot
        filled_courses = {
            i: dict(zip(elect_slots_copy[i], [0] * 20)) for i in range(4)}
        for student in students:
            for i, slot in enumerate(elect_slots_copy):
                course = random.choice(slot)
                students[student].append(course)
                filled_courses[i][course] += 1
                if filled_courses[i][course] == 25:
                    elect_slots_copy[i].remove(course)

        # 6 projection back
        timetable = {str(i): {str(j): {str(k): "G" for k in range(1, 11)}
                              for j in range(1, 6)} for i in range(offset+1,
offset+501)}
        # additional slot to hold the extra english and PE class
        filled_rooms.append(
            dict(zip(range(2, 9), [set() for _ in range(2, 9)])))
        if grade == "G1":
            core_set = 0
            elect_set = 1
        else:
            core_set = 1
            elect_set = 0
        for k, student in students.items():
            for course in student[:4]:  # for the core classes
                for index, slot in enumerate(core_slots):
                    if course in slot:
                        day, p1, p2 = self.glevel_set_map[core_set][index][0]
.split(
                            "_")
                        timetable[str(k)][day][p1] = rooms[course]
                        timetable[str(k)][day][p2] = rooms[course]
                        day, p1, p2 = self.glevel_set_map[core_set][index][1]
.split(
                            "_")
                        timetable[str(k)][day][p1] = rooms[course]
                        timetable[str(k)][day][p2] = rooms[course]
                # handle the extra english and PE class to spare slots
                if course.split("_")[0] == 'Eng' and int(course.split("_")[-1]) <=
9:
                    timetable[str(k)]['4']['1'] = rooms[course]
                    timetable[str(k)]['4']['2'] = rooms[course]
                    timetable[str(k)]['3']['7'] = 'G'
                    timetable[str(k)]['3']['8'] = 'G'
                    filled_rooms[-1][5].add(rooms[course])
                elif course.split("_")[0] == 'Eng' and int(course.split("_")[-1])
> 9:
                    timetable[str(k)]['4']['1'] = 'G'
                    timetable[str(k)]['4']['2'] = 'G'
                    timetable[str(k)]['3']['7'] = rooms[course]
                    # TODO: place G1 and G2 with un-collided English rooms
                    timetable[str(k)]['3']['8'] = rooms[course]
                    # add g2 extra classes
                    filled_rooms[-1][5].add(rooms[course])
            for course in student[4:]:  # for the elective classes
                for index, slot in enumerate(elect_slots):
                    if course in slot:
```

```
                        day, p1, p2 = self.glevel_set_map[elect_set][index][0]
.split(
                            "_")
                        timetable[str(k)][day][p1] = rooms[course]
                        timetable[str(k)][day][p2] = rooms[course]
                        day, p1, p2 = self.glevel_set_map[elect_set][index][1]
.split(
                            "_")
                        timetable[str(k)][day][p1] = rooms[course]
                        timetable[str(k)][day][p2] = rooms[course]

        self.filled_rooms[8][5].update(filled_rooms[8][5])
        self.time_table.update(timetable)

        return timetable

    def generate_as_al_level(self):
        """
        Generate AS-Level Classes
        1: Find the total number of classes for each of the categories: Extra(1F),
Bus(2F), Hum(3F), Enr(4F), Lan(5F), Sci(6F)
            1.1 assume that each student take 1 Eng and 4 Electives → 5*3=15 2-
periods per week for a student
            1.2 5*500/25=100 2-periods courses in total
        2: distribute 100 courses into Extra, Bus, Hum, Enr, Lan, Sci following
common-sense weightings (dict "as_freq")
            2.1 Extra (art/drama/pe/music): 4
            2.2 Bus (econ/acc): 10
            2.3 Hum (geo/his/psy): 10
            2.4 Mat (mat/further/cs): 23
            2.5 Eng (lit/lan/gc/3rd lan): 23
            2.6 Sci (bio/phy/chem): 30
        3: distribute timeslots for each course
            3.1 assign fixed scheduling
                - each slot accounts for 3 2-periods in AS
                - there are in total 6 slots for AS → don't need to fill in every
slot
            3.2 define schedule sets
                1. Mon12, Tue78, Thu34
                2. Mon34, Tue90, Thu78
                3. Mon78, Wed12, Thu90
                4. Mon90, Wed34, Fri12
                5. Tue12, Wed78, Fri34
                6. Tue34, Thu12, Fri56
        4: distribute the rooms for each course
            4.1 the rooms available for each floor is stored in the dict "
avail_rooms"
            4.2 limitations: courses at each scheduling set should not have the
same room
                - should check "filled_rooms" when handling this
                - hence, process "filled_rooms" into a timetable format
        5: distribute each student's list of courses taken
        6: project back to generate the timetable


        :return: timetable as a JSON
        """
        # dictionary for as course frequencies
        as_freq = {
            "Extra": 4,
```

```
            "Bus": 10,
            "Hum": 10,
            "Enr": 23,
            "Eng": 23,
            "Sci": 30
        }
        # dictionary for as course scheduling sets
        as_set_map = {
            0: ["1_1_2", "2_7_8", "4_3_4"],
            1: ["1_3_4", "2_9_10", "4_7_8"],
            2: ["1_7_8", "3_1_2", "4_9_10"],
            3: ["1_9_10", "3_3_4", "5_1_2"],
            4: ["2_1_2", "3_7_8", "5_3_4"],
            5: ["2_3_4", "4_1_2", "5_5_6"]
        }
        # Procedures #
        # 3 distribute timeslots for each course, each slot only needs 20 courses
        slots = [[] for _ in range(6)]
        avail_slots = list(range(6))
        for k, v in as_freq.items():
            for i in range(v):
                slot = random.choice(avail_slots)
                slots[slot].append(k + "_" + str(i))
                if len(slots[slot]) == 20:
                    avail_slots.remove(slot)

        # 4 distribute the rooms for each course
        # transform the g_level filled_rooms into a timetable format
        filled_timetable = {str(j): {str(k): set()
                                     for k in range(1, 11)} for j in range(1, 6)}
        glevel_set_map = list(self.glevel_set_map[0].values(
        )) + list(self.glevel_set_map[1].values())
        # don't include the extra two slots at last
        for index, slot in enumerate(self.filled_rooms[:8]):
            for v in slot.values():
                for room in v:
                    day, p1, p2 = glevel_set_map[index][0].split("_")
                    filled_timetable[day][p1].add(room)
                    filled_timetable[day][p2].add(room)
                    day, p1, p2 = glevel_set_map[index][1].split("_")
                    filled_timetable[day][p1].add(room)
                    filled_timetable[day][p2].add(room)
        # process the last two extra slots
        for index, slot in enumerate(self.filled_rooms[8:]):
            for v in slot.values():
                for room in v:
                    filled_timetable['3']['7'].add(int(room[1:]))
                    filled_timetable['3']['8'].add(int(room[1:]))
                    filled_timetable['4']['1'].add(int(room[1:]))
                    filled_timetable['4']['2'].add(int(room[1:]))

        # 4 distribute the rooms for each course
        rooms = {}
        filled_rooms = []  # track the filled rooms for each floor for each slot
        for index, slot in enumerate(slots):
            day1, p11, p12 = as_set_map[index][0].split("_")
            day2, p21, p22 = as_set_map[index][1].split("_")
            day3, p31, p32 = as_set_map[index][2].split("_")
```

```
            filled_room = dict(zip(range(2, 9), [set() for _ in range(2, 9)]))
            for course in slot:
                if course[:3] == "Sci":  # sci courses are distributed to three floors
                    # TODO: change to greedy floor
                    floor = random.choice([6, 7, 8])
                else:
                    floor = self.floor_map[course.split("_")[0]]
                    if floor == 1:  # if it is extra, just assign G
                        rooms[course] = "G"
                        continue
                # divide by the current filled rooms, use mod because the core slots for g1 are elective slots for g2
                room = random.choice(
                    list(self.avail_rooms[floor] - filled_timetable[day1][p11] - filled_timetable[day2][p21] - filled_timetable[day3][p31]))
                rooms[course] = self.map_building(room) + str(room)
                filled_room[floor].add(room)
            filled_rooms.append(filled_room)

        # 5 distribute each student's list of courses taken
        # generate the students' slots
        slots_count = [[i, len(slot)*25] for i, slot in enumerate(slots)]
        offset = 21000
        students_slots = {i: [] for i in range(offset+1, offset+501)}
        for student in students_slots:
            slots_count = sorted(slots_count, reverse=True, key=lambda x: x[1])
            slots_indices = [item[0] for item in slots_count[:5]]
            students_slots[student] = slots_indices
            for i in range(6):
                if slots_count[i][0] in slots_indices:
                    slots_count[i][1] -= 1

        # print(students_slots)

        # generate the students
        students = {i: [] for i in range(offset+1, offset+501)}
        slots_copy = copy.deepcopy(slots)
        # track the filled courses for each slot
        filled_courses = {
            i: dict(zip(slots_copy[i], [0] * len(slots_copy[i]))) for i in range(6)}
        for student in students:
            # print(student)
            for i in students_slots[student]:
                slot = slots_copy[i]
                course = random.choice(slot)
                students[student].append(course)
                filled_courses[i][course] += 1
                if filled_courses[i][course] == 25:
                    slots_copy[i].remove(course)

        # 5 generate the timetable
        timetable = {str(i): {str(j): {str(k): "G" for k in range(1, 11)}
                              for j in range(1, 6)} for i in range(offset+1, offset+501)}
        for k, student in students.items():
            for course in student:
                for index, slot in enumerate(slots):
                    if course in slot:
```

```
                        day, p1, p2 = as_set_map[index][0].split("_")
                        timetable[str(k)][day][p1] = rooms[course]
                        timetable[str(k)][day][p2] = rooms[course]
                        day, p1, p2 = as_set_map[index][1].split("_")
                        timetable[str(k)][day][p1] = rooms[course]
                        timetable[str(k)][day][p2] = rooms[course]
                        day, p1, p2 = as_set_map[index][2].split("_")
                        timetable[str(k)][day][p1] = rooms[course]
                        timetable[str(k)][day][p2] = rooms[course]

        """Generate AL Level, written in the same function to avoid passing filled_rooms which is hard for different set maps"""
        """
        Generate AL-Level Classes
        1: Find the total number of classes for each of the categories: Extra(1F), Bus(2F), Hum(3F), Enr(4F), Lan(5F), Sci(6F)
            1.1 assume that each student take 1 Eng and 3 Electives → 4*3=12 2-periods per week for a student
            1.2 4*500/25=80 2-periods courses in total
        2: distribute 80 courses into Extra, Bus, Hum, Enr, Lan, Sci following common-sense weightings (dict "al_freq")
            2.1 Extra (art/drama/pe/music): 4
            2.2 Bus (econ/acc): 12
            2.3 Hum (geo/his/psy): 10
            2.4 Mat (mat/further/cs): 15
            2.5 Eng (lit/lan/gc/3rd lan): 23
            2.6 Sci (bio/phy/chem): 20
        3: distribute timeslots for each course
            3.1 assign fixed scheduling
                - each slot accounts for 3 2-periods in AL
                - the set map for AL is the same as that for AS
            3.2 define schedule sets
                1. Mon12, Tue78, Thu34
                2. Mon34, Tue90, Thu78
                3. Mon78, Wed12, Thu90
                4. Mon90, Wed34, Fri12
                5. Tue12, Wed78, Fri34
                6. Tue34, Thu12, Fri56
        4: distribute the rooms for each course
            4.1 the rooms available for each floor is stored in the dict "avail_rooms"
            4.2 limitations: courses at each scheduling set should not have the same room
                - should check "filled_rooms" when handling this
                - hence, process "filled_rooms" into a timetable format
        5: distribute each student's list of courses taken
        6: project back to generate the timetable


        :return: timetable as a JSON
        """
        al_freq = {
            "Extra": 4,
            "Bus": 12,
            "Hum": 10,
            "Enr": 15,
            "Eng": 23,
            "Sci": 20
        }

```

```
        # PROCEDURES #
        # 3 distribute timeslots for each course, each slot only needs 20 courses
        slots = [[] for _ in range(6)]
        avail_slots = list(range(6))
        for k, v in al_freq.items():
            for i in range(v):
                slot = random.choice(avail_slots)
                slots[slot].append(k + "_" + str(i))
                if len(slots[slot]) == 20:
                    avail_slots.remove(slot)

        # 4 distribute the rooms for each course
        rooms = {}
        for index, slot in enumerate(slots):
            day1, p11, p12 = as_set_map[index][0].split("_")
            day2, p21, p22 = as_set_map[index][1].split("_")
            day3, p31, p32 = as_set_map[index][2].split("_")
            filled_room = dict(zip(range(2, 9), [set() for _ in range(2, 9)]))
            for course in slot:
                if course[:3] == "Sci":  # sci courses are distributed to three floors
                    # TODO: change to greedy floor
                    floor = random.choice([6, 7, 8])
                else:
                    floor = self.floor_map[course.split("_")[0]]
                    if floor == 1:  # if it is extra, just assign G
                        rooms[course] = "G"
                        continue
                # divide by the current filled rooms, use mod because the core slots for g1 are elective slots for g2
                room = random.choice(
                    list(self.avail_rooms[floor] - filled_timetable[day1][p11] - filled_timetable[day2][p21]
                         - filled_timetable[day3][p31] - filled_rooms[index][floor]))  # minus the as time slots as well
                rooms[course] = self.map_building(room) + str(room)
                filled_room[floor].add(room)

        # 5 distribute each student's list of courses taken
        # generate the students' slots
        slots_count = [[i, len(slot)*25] for i, slot in enumerate(slots)]
        offset = 20000
        students_slots = {i: [] for i in range(offset+1, offset+501)}
        for student in students_slots:
            slots_count = sorted(slots_count, reverse=True, key=lambda x: x[1])
            slots_indices = [item[0] for item in slots_count[:4]]
            students_slots[student] = slots_indices
            for i in range(6):
                if slots_count[i][0] in slots_indices:
                    slots_count[i][1] -= 1

        # print(students_slots)

        # generate the students
        students = {i: [] for i in range(offset+1, offset+501)}
        slots_copy = copy.deepcopy(slots)
        # track the filled courses for each slot
        filled_courses = {
            i: dict(zip(slots_copy[i], [0] * len(slots_copy[i]))) for i in range(6)}
```

```
        for student in students:
            # print(student)
            for i in students_slots[student]:
                slot = slots_copy[i]
                course = random.choice(slot)
                students[student].append(course)
                filled_courses[i][course] += 1
                if filled_courses[i][course] == 25:
                    slots_copy[i].remove(course)

        # 5 generate the timetable
        al_timetable = {str(i): {str(j): {str(k): "G" for k in range(1, 11)}
                                 for j in range(1, 6)} for i in range(offset+1,
offset+501)}
        for k, student in students.items():
            for course in student:
                for index, slot in enumerate(slots):
                    if course in slot:
                        day, p1, p2 = as_set_map[index][0].split("_")
                        al_timetable[str(k)][day][p1] = rooms[course]
                        al_timetable[str(k)][day][p2] = rooms[course]
                        day, p1, p2 = as_set_map[index][1].split("_")
                        al_timetable[str(k)][day][p1] = rooms[course]
                        al_timetable[str(k)][day][p2] = rooms[course]
                        day, p1, p2 = as_set_map[index][2].split("_")
                        al_timetable[str(k)][day][p1] = rooms[course]
                        al_timetable[str(k)][day][p2] = rooms[course]

        timetable.update(al_timetable)
        self.time_table.update(timetable)

        return timetable

    def add_pshe_gt(self):
        """
        Add PSHE session for each student.
        For G1s and G2s, this lesson is added at P5 on everyday to immitate the 1-
period classes.
        """
        g1_offset = 23000
        g2_offset = 22000
        student_list = list(self.time_table.keys())
        offset = 0
        flag = True
        for floor in self.avail_rooms.values():
            for room in floor:
                for i in range(offset, offset+25):
                    if i >= len(student_list):
                        flag = False
                        break
                    if int(student_list[i]) // 1000 == g1_offset // 1000 or
int(student_list[i]) // 1000 == g2_offset // 1000:
                        for day in range(1, 6):
                            self.time_table[student_list[i]][str(
                                day)]['5'] = self.map_building(room) + str(room)
                    else:
                        gt_day = random.choice(["1", "2", "4", "5"])
                        gt_floor = random.choice([6, 7, 8])
                        gt_room = random.choice(
```

```
                            list(self.avail_rooms[gt_floor]))
                    if gt_day != "5":
                        self.time_table[student_list[i]][gt_day]['5'] =
self.map_building(
                            gt_room) + str(gt_room)
                        self.time_table[student_list[i]][gt_day]['6'] =
self.map_building(
                            gt_room) + str(gt_room)
                    else:
                        self.time_table[student_list[i]][gt_day]['7'] =
self.map_building(
                            gt_room) + str(gt_room)
                        self.time_table[student_list[i]][gt_day]['8'] =
self.map_building(
                            gt_room) + str(gt_room)
                    self.time_table[student_list[i]]['3']['5'] =
self.map_building(
                            room) + str(room)
                if not flag:
                    break
                offset += 25
            if not flag:
                break

        return self.time_table


if __name__ == "__main__":
    # Need to generate multiple times so that the rooms not crash, therefore
creating no errors
    generator = TimetableGenerator()
    time_table1 = generator.generate_g_level("G1")
    time_table2 = generator.generate_g_level("G2")
    time_table3 = generator.generate_as_al_level()
    generator.add_pshe_gt()

    with open("timetable_with_gt_0.json", "w") as file:
        json.dump(generator.time_table, file, indent=4)

```