DSA Project Report
Segfault Squad

24B0929, 24B1010, 24B1012

November 2025

Contents

1 Directory Structure

```
.
|Include
|   `-- nlohmann
|       `-- json.hpp
|-- Phase-1
|   |-- include
|   |   `-- nlohmann
|   |       `-- json.hpp
|   |-- Graph.cpp
|   |-- Graph.hpp
|   |-- makefile
|   |-- SampleDriver.cpp
|   `-- script.sh
|-- Phase-2
|   |-- include
|   |   `-- nlohmann
|   |       `-- json.hpp
|   |-- Graph.cpp
|   |-- Graph.hpp
|   |-- makefile
|   |-- SampleDriver.cpp
|   `-- script.sh
|-- results
|   |-- testcase0
|   |-- testcase1
|   |-- testcase2
|   |-- testcase3
|   |-- testcase4
|   |-- testcase5
|   |-- testcase6
|   |-- testcase7
|   |-- testcase8
|   `-- testcase9
|-- tests
|   |-- testcase0
|   |-- testcase1
|   |-- testcase2
|   |-- testcase3
|   |-- testcase4
|   |-- testcase5
|   |-- testcase6
|   |-- testcase7
|   |-- testcase8
|   `-- testcase9
```

```
|-- SampleDriver.cpp
|-- makefile
`-- report.pdf
```

1.1 Phase 1

This directory contains the implementation of Phase 1 queries. The include directory contains the header files we have used in the project. This includes the nlohmann folder that houses the single include json.hpp file to help us parse json files in C++. Next we have the Graph.cpp and the Graph.hpp files which contain the implementation and declaration of the Graph class respectively. The makefile is used to compile the code. phase1 is the executable file that is generated after compilation. SampleDriver.cpp is the driver code that is given to parse the json queries and call the appropriate C++ functions. script.sh is the bash script that we use to run our code on multiple test cases.

1.2 Phase 2

These directories follow the same structure as Phase 1, with the implementation of Phase 2 queries. The executable is named phase2.

1.3 report.pdf

This pdf contains the project report.

1.4 results

This directory contains the output files generated after running our code on the test cases in the tests directory. Each testcase$_i$ directiory contains 2 files - output1.json and output2.json respectively for Phase 1 and Phase 2 query outputs.

1.5 tests

Each testcase$_i$ directory contains the testcases we run our input on. Each directory contains the graph.json file which contains the graph data and the queries1.json and queries2.json files which contain the queries for Phase 1 and Phase 2 queries respectively.

2 Makefile and Targets

all: This is the default target. It compiles the code and generates the executable file.

run: runs the executable file on the graph.json and queries.json files present in the current directory and generates output.json file.

clean: removes all object files.

cleanx: removes the executable file.

test: executes the script.sh file to run the code on all test cases.

phase1: compiles the code in Phase-1 directory and generates the phase1 executable in root directory.

phase2: compiles the code in Phase-2 directory and generates the phase2 executable in root directory.

clean: removes all object files and executables in both Phase-1 and Phase-2 directories.

The first 4 targets are for makerfile in each phase directory while the last 3 targets are for the makefile in the root directory.

3 Assumptions

3.1 K shortest path heuristic

- We assume that the penalty factor and stretch factor used in the heuristic approach are appropriate for generating diverse candidate paths. These factors may need to be tuned based on the specific characteristics of the input graphs to achieve optimal performance.
- We assume that we can find atleast K candidate paths within the maximum number of attempts specified (250 in our case) because otherwise there might exist paths which we should output but are not outputting them due to the attempt limit.
- We assume that the input graphs are connected and that there exists at least one path between the source and target nodes for the K shortest paths query else no heuristic paths are found.

4 Test Case and Analysis

We have not used any script to generate test cases. We have borrowed the test case and results from another group. We have run our code on the test cases and compared our output with the expected output(their output). We have compared our output with expected output using the script.sh script which has been described in the next section.

5 Python Scripts and Libraries

We have used a bash script to run out code on multiple test cases. This bash script creates a python script to compare our output with the expected output and prints the differences accordingly(A more intelligent comparison was needed rather than just comparing using diffs since processing times were bound to be different).
The python script uses the json library to parse the json files and compare the outputs and the sys library to take command line arguments.
After this the bash script runs the python script on each test case and prints the differences accordingly. Finally the bash script removes the python script created.

6 Time and Space Complexity

Let V be the number of vertices and E be the number of edges in the graph. Let K be the number of shortest paths to be found.

6.1 Remove Edge

- **Time Complexity:** $\mathcal{O}(1)$ - We use a hash map to store edges for $\mathcal{O}(1)$ access.
- **Space Complexity:** $\mathcal{O}(E)$ - We store all edges in a hash map.

6.2 Modify Edge

- **Time Complexity:** $\mathcal{O}(1)$ - We use a hash map to store edges for $\mathcal{O}(1)$ access.
- **Space Complexity:** $\mathcal{O}(E)$ - We store all edges in a hash map.

6.3 Shortest Path

- **Time Complexity:** $\mathcal{O}((V+E)\log V)$ - We use Dijkstra's algorithm with a priority queue.
- **Space Complexity:** $\mathcal{O}(V+E)$ - We store the graph using adjacency lists.

6.4 KNN

- **Time Complexity:** $\mathcal{O}(V\log K + (V+E)\log V)$ - For Euclidean KNN, we compute distances to all nodes and maintain a max-heap of size K. For Shortest Path KNN, we run Dijkstra's algorithm.
- **Space Complexity:** $\mathcal{O}(V+E+K)$ - We store the graph using adjacency lists and maintain a max-heap of size K.

6.5 K shortest paths (exact)

- **Time Complexity:** We need to find the time complexity of Yen's algorithm here. The time complexity of Dijsktra's algorithm using a min-heap is $\mathcal{O}((V+E)\log V)$. In the worst case, we may need to find k shortest paths, and for each path, we may need to run Dijkstra's algorithm up to V times (once for each node in the path). Therefore, the overall time complexity of Yen's algorithm is $\mathcal{O}(kV(V+E)\log V)$.
- **Space Complexity:** The space complexity is $\mathcal{O}(V+E)$ to store the graph and $\mathcal{O}(kV)$ to store the k shortest paths.

6.6 K Shortest Paths (heuristic)

- **Time Complexity:** In the first part of our function, we run Dijkstra algorithm max_attempts times to find candidate paths. Each Dijkstra run takes $\mathcal{O}((V+E)\log V)$ time using a priority queue. Therefore, the time complexity for finding candidate paths is $\mathcal{O}(\text{max\_attempts} \cdot (V+E)\log V)$ where max_attempts is a constant (250 in our case). In the second part, we evaluate all combinations of candidate paths to select the best K paths. If we have C candidate paths, evaluating all combinations of size K takes $\mathcal{O}\left(\binom{C}{K} \cdot K^2 \cdot L\right)$ time, where L is the average length of path. We get this since the time complexity of calculate_group_score() is K^2 and it is called once for each subset of size K. Since K is small (between 2 and 7), this part is manageable as maximum value of C is $2 \cdot K$. Thus, the overall time complexity is dominated by the Dijkstra runs, resulting in an overall time complexity of $\mathcal{O}(A(V+E)\log V)$ where A is maximum attempts. This is well within acceptable limits for our input size which is capped at 5000 nodes.

- **Space Complexity:** The space complexity is $\mathcal{O}(V+E)$, which accounts for storing the graph representation (adjacency list) and additional data structures used during the execution of Dijkstra's algorithm, such as distance and parent arrays. The candidate paths and their associated data also contribute to the space complexity, but since K is small, this does not significantly affect the overall space complexity. All the distance and parent arrays used in Dijkstra's algorithm are of size $\mathcal{O}(V)$, and the graph representation takes $\mathcal{O}(V+E)$ space. Therefore, the overall space complexity remains $\mathcal{O}(V+E)$.

6.7 K shortest paths (approx)

- **Time Complexity:** The worse-case time complexity of the Weighted A* algorithm is the same as that of the standard A* algorithm i.e $\mathcal{O}(E \log V)$. In the worst case, the algorithm may visit every node and relax eery edge. We use a min-heap to store the nodes to be explored. Each edge is processed once. For each edge, we might perfor a heap operation (insert or update) which takes $\mathcal{O}(logV)$ time. Therefore, the overall time complexity is $\mathcal{O}(ElogV)$.

  The effective time complexity is lower in practice due to the heuristic pruning. By inflating the $h(n)$ value, the algorithm prioritizes nodes closer to the target more aggressively. This reduces the number of nodes explored, leading to faster execution in larger graphs.

- **Space Complexity:** The space complexity is $\mathcal{O}(V+E)$ to store the graph and $\mathcal{O}(V)$ for the priority queue and other data structures used in the algorithm.

7 Approach for Each Query

7.1 Phase 1 Queries

7.1.1 Remove Edge

This is pretty straightforward. We just make the disabled flag of that edge true in case the edge exists and is not already disabled. We return true in this case. If the edge does not exist or is already disabled, we return false.

7.1.2 Modify Edge

We first check if the query has a patch. If it does, we check which attributes need to be modified. If no modification is needed, we return false. Otherwise we modify the required attributes and return true. Before modification we check if the edge exists and is not disabled. If it does not exist we return false directly. If it exists and is disabled we enable it and automatically return true since modification is guaranteed. Modification of attributes is done as per the patch provided in the query.

7.1.3 Shortest path

For shortest path query, we use Dijkstra's algorithm. We first parse through the query to get required parameters like source, target, forbidden nodes and road types. We then run modified Dijkstra's algorithm on the graph considering the forbidden nodes and road types. If we reach the target node, we return the shortest distance. If we exhaust all possibilities without reaching the target, we return false in the "possible" attribute of answer indicating no path exists. The notion of minimum is either distance of time based on the query parameter. If it is distance, we consider edge weights as distances. If it is time, time cost is calculated using the speed profile given in the edge attributes which is done by finding the speed corresponding to the given time of day and calculating time cost as distance/speed.

7.1.4 KNN

There are two types of KNN queries: one based on euclidean distance and the other based on shortest path distance. For euclidean distance based KNN query, we simply calculate the euclidean distance from the given coordinates to all other nodes and maintain a max-heap which we pop until size is K to get the K nearest nodes which are then returned. For shortest path distance based KNN query, first we find the nearest node from the given coordinates using euclidean distance. We then run modified Dijkstra's algorithm from that node to find shortest path distances to all other nodes considering forbidden nodes and road types. We maintain a max-heap of size K to get the K nearest nodes based on shortest path distance which are then returned.

7.2 Phase 2 Queries

7.2.1 K shortest paths (exact)

To find the k shortest paths between a source and target node, we implement Yen's algorithm. The algorithm works as follows:

We begin by running a standard Dijsktra's search to find the optimal path from the source to the target. This path is stored as the first entry in our paths vector.

To find the next shortest path, we look at the previous shortest path and try to deviate from it at every possible node. For each node in the previous path (except the destination), we do the following:

- Define the spur node as the current node that is being processed. This acts as the "deviation point" where the new path will branch off.
- Define the rooth path as the path form the source to the spur node. This path remains fixed for this iteration.

To ensure the algorithm finds a new path that is different from previous ones, it temporarily modifies the graph. It looks at all previoiusly disovered paths. If a previous path shares the same root path, the edge immediately from the spur node in the previous path is disabled (marked disabled = true). This forces the new path to deviate from the previous paths. To ensure the paths are loopless, all nodes appearing in the root path are also forbidden temporarily. With these constraints, we run Dijkstra's from the spur node to the target. This results in a spur path. We combine it with the root path to form a total path. This path is added to a min-heap of candidate paths. After checking all deviation points, the graph edges are re-enabled for the next iteration. The algorithm returns the best path from the candidate paths heap.

7.2.2 K shortest paths (heuristic)

Unlike before where we used Yen's algorithm to find K shortest paths, we now use a heuristic approach to find candidate paths and then select the best one based on certain criteria. For finding the candidate paths, we first find the shortest path using Dijkstra's algorithm. We then multiply the edge weights of the edges in the shortest path by a penalty factor which needs to be tuned appropriately and run Dijkstra's algorithm again to find another path. We repeat this process until we have adequate number of candidate paths. Penalty is also implemented for paths that are unique so that running Dijkstra again does not return same path which will lead to infinite loop. The candidate paths is valid if we do not get a previously found path again and if the length of the path is within a certain stretch factor of the original shortest path length after the size of candidate set is k in order to ensure k outputs.

Once we have the candidate paths, we need to choose the K paths among them. To do this, we calculate 2 types of penalties for each candidate path: distance penalty and overlap penalty. Since K is between 2 and 7, we have calculated to total penalty of all sets of size k with shortest path present in it. The set which minimizes total penalty is out set of best candidates. These k paths are then returned but if there does not exist k paths then we return all possible paths. This is not a greedy algorithm as we check for everything to minimize the total penalty as much as possible.

7.2.3 K shortest paths (Approx)

We are asked to prioritse speed over accuracy in this query. This means that there will exist a trade off between speed and accuracy. We use a modified version of the A* algorithm called the weighted A* algorithm.

We define the heuristic function as the straight line distance between the current node and the target. We applied a conversion factor (approximately 111km per degree) to cnovert the latitude and longitude differences into approximate distances. We scale the heuristic function by a weight factor w > 1. This makes the algorithm more greedy and faster but less accurate.

Define $f(n) = g(n) + w \cdot h(n)$ where $g(n)$ is the cost from the source to node n, $h(n)$ is the heuristic function (straight line distance from n to target), and w is the weight factor.

We initialize a min-heap that stores nodes and the value computed by our function. We add the source node to queue with initial f-score based entirely on the heuristic (since g = 0). We run a loop over the priority queue until it is empty. In each iteration we pop the node and check whether it has been visited. If not we mark it as visited. If the node is the target node, we have found the shortest path and we can return. We iterate through all the neighbours of the current node and calculate the tentative g-score. If this score is better than the previously recorded g-score for that neigbour, we update the g-score and calculate its new weighted f-score and push it into the priority queue.

8 AI Logs

Following are the links to AI chats that were used during the project:

- `https://chatgpt.com/share/6921f560-ec18-800e-8b56-f3261fb217fa`
- `https://gemini.google.com/share/8048a10707bc`
- `https://chatgpt.com/share/6921f4bc-cd70-800e-9681-c1987ae62140`
- `https://gemini.google.com/share/e13a328ce343`
- `https://gemini.google.com/share/973a4576fdcc`
- `https://gemini.google.com/share/4ee9834abdc9`