Abstract

This project proposes an algorithm to find the shortest path between two points on a road network, considering real-time traffic data

Navigation System for the Boss

DAVYDOV DOMINIK

Dd22072

2023

# Requirements and Assumptions.

The goal of this project is to develop a traffic control system for high-ranking official (the Boss) in Riga, Latvia, to ensure his travel time is as fast as possible. The system will use data gathered by public services about the traffic on different streets in the city to provide the car of the Boss with computer-assisted route planning.

## 1.1 Requirements:

1. The system should read a simplified street plan from an input file that divides the streets into segments stretching from one intersection to another.
2. The system should receive data about each street segment from the public services, including:
   1. The maximum number of cars Ci that can drive through the street segment Si without experiencing any delay. The value Ci=0 means that the segment is currently closed.
   2. The time Ti required to travel the segment Si, which is the minimum time when there are no obstacles.
   3. The number of cars Ni currently driving through the segment Si.
   4. The maximum number of cars Mi that can be squeezed on the street segment Si during the peak traffic jam.
   5. The delay time Di that is additionally required on the segment Si for every car that is above the number Ci. This lets us compute the actual time required for the segment Si.
3. The system should receive current coordinates of the Boss's car in Si and the direction of travel.
4. The Boss can send to the system his destination segment Si at any time, and the system will return the segment numbers in the order that would allow reaching the destination as early as possible.
5. The values of Ci, Ti, Di, and Ni can change at any time, and the system should be able to receive new values.
6. The Boss can ask to recompute the chosen route at any time if he thinks that the speed is insufficient.

## 1.2 Assumptions:

1. The implementation of this system has full support from the Riga City Council and the government, and the public services will provide the required data without any issues.
2. The system does not consider any traffic rules, regulations, or signals. It only considers the data provided by the public services.
3. The system is designed only for the Boss's car, and not for any other car.
4. The system is designed to optimize the Boss's travel time, and not to optimize traffic flow or reduce congestion.
5. The GPS system accurately determines the Boss's current coordinates and provides them to the central server.
6. The program that receives the Boss's coordinates and returns the segment and direction exists and is functional.
7. The values for the segment information can be provided either on schedule or on demand by the public services.
8. The central server can handle multiple requests from the Boss and other users simultaneously.
9. The company Insatiable Ltd will add the street names to the segment numbers.
10. Each segment represents a one-way road, and if there is a two-way road, there should be both directions represented as separate segments.
11. If any cars are stopped on intersections of the roads, they are counted in one of the aligned segments.
12. The destination segment should be passed by the Boss’s car in any way, and consequently, the destination node has only one edge and this edge’s value is not equal to 0.
13. There is only one way to ride from the current location of the Boss’s car, meaning that the road is open, and in any case, it will be ridden through.
14. Circular motion roads should be divided into segments.

# Input File.

The input file should contain a simplified street plan that divides the streets into segments stretching from one intersection to another. Each line in the file should contain the following information about a single segment:

The first line should contain the Segment ID of the Boss’s current location.

The second line should contain the Segment ID of the Boss’s destination.

Subsequent pairs of lines should have the following structure: Line one contains information with Segment ID (Si) (Segment ID cannot be equal to 0), Maximum number of cars can drive without any delays (Ci), Time required to travel through the segment (Ti) in seconds, Number of cars currently driving (Ni), Maximum number of cars during peak traffic (Mi), Delay time in seconds for every car above Ci (Di); Line two contains Segment IDs of the predecessor segments. Segments with no predecessor nodes in this line have 0.

Example of input file:

1

2

1 10 900 2 20 60

0

2 15 1020 5 30 120

1

4 0 240 0 9 60

1

In this example, the program gets the Segment ID 1, the Boss’s current location, the Segment ID 2, the Boss’s destination, and information about 3 segments. The first segment’s properties: maximum 10 cars can drive through this segment without experiencing any delay, time required to travel through the segment is 900 seconds, number of cars currently driving through the segment is 2, maximum number of cars during peak traffic jam is 20, Delay time for every car above 10 is 1 minute. The second segment’s properties: maximum 15 cars can drive through this segment without experiencing any delay, time required to travel through the segment is 1020 seconds, number of cars currently driving through the segment is 5, maximum number of cars during peak traffic jam is 30, Delay time for every car above 15 is 2 minutes. The fourth segment’s properties: maximum 0 cars can drive through this segment without experiencing any delay, that means the segment is currently closed and the Boss cannot drive through it. And the first segment connected to the second segment and to the fourth segment.

# Output File:

The output file should contain the ordered list of segment numbers that the Boss needs to travel through to reach his destination as quickly as possible, along with the estimated time of arrival. The estimated time of arrival can be calculated by adding up current time and the minimum time required for each segment, as well as any additional delay time due to the number of cars currently driving on the segment.

The output file format should be as follows:

The first line will contain the total number of segments in the route. If the number is 0 then ‘No path can be found’ and the next lines should not be read.

The second line will contain the estimated travel time to travel to the destination, in seconds.

The third line will contain a sequence of segment numbers in the order that would allow reaching the destination as early as possible.

Example of output file:

3

930

2 6 9

In this example, the Boss needs to travel through 3 segments to reach his destination. The estimated travel time is 930 seconds. And 2, 6 and 9 are the segments the Boss needs to travel, where 9 is the destination segment.

# Solution alternatives and theoretical justification.

Here are several approaches with their theoretical justifications, advantages, and disadvantages:

## 4.1 Dijkstra's algorithm:

Dijkstra's algorithm is a widely used algorithm for finding the shortest path in a graph with non-negative edge weights. It maintains a priority queue of unvisited nodes and selects the next node with the smallest distance from the source node, making it suitable for finding the shortest path from a current location to a destination in a network of streets. The program can represent the streets as nodes and the intersections as edges, with each node having a list of edges representing the possible paths.

The algorithm will start at the Boss's current location and calculate the shortest path to the destination, considering factors such as the maximum number of cars allowed on each street segment, the current number of cars on each segment, the maximum number of cars during peak traffic, and the delay time for each additional car. It will return the sequence of street segment IDs that the Boss's car should take to reach the destination as quickly as possible.

One advantage of Dijkstra's algorithm is that it is guaranteed to find the shortest path in a graph with non-negative edge weights. It is also relatively simple to implement and has a time complexity of O((E+V) logV), where E is the number of edges and V is the number of vertices in the graph. However, it requires storing all the edges in the graph, which can be impractical for large graphs. Additionally, if there are negative edge weights, the algorithm can produce incorrect results.

## 4.2 A\* algorithm:

A\* is a heuristic search algorithm that improves upon Dijkstra's algorithm by incorporating a heuristic function that estimates the cost from a node to the destination. It works by maintaining a priority queue of unvisited nodes and selecting the next node with the smallest f-score, where the f-score is the sum of the actual cost from the source to the current node and the estimated cost from the current node to the destination. A\* is suitable for this problem, as a good heuristic function can estimate the time required to reach the destination. The time complexity of A\* is O((E+V) logV), where E is the number of edges and V is the number of vertices in the graph.

One advantage of A\* is that it can prune parts of the search space that are unlikely to lead to the goal, making it faster than Dijkstra's algorithm. It is also guaranteed to find the shortest path if the heuristic is admissible, meaning it never overestimates the true distance to the goal. However, one disadvantage of A\* is that it requires a good heuristic function to be effective. If the heuristic is poor, the algorithm may take longer to converge than Dijkstra's algorithm. Additionally, like Dijkstra's algorithm, it requires storing all the edges in the graph. A\* can be especially useful in large graphs, as it reduces the number of nodes that need to be explored.

## 4.3 Reinforcement learning:

Reinforcement learning is an artificial intelligence algorithm that can be used for traffic management problems, where an agent, such as the Boss's car, interacts with an environment, like city streets. The agent's goal is to learn a policy that maximizes a reward function, such as the Boss's arrival time, without requiring a complete model of the environment. Reinforcement learning is advantageous as it can adapt to changing environments and learn complex, non-linear policies that are difficult to design by hand. However, it can be computationally expensive and requires careful design of the reward function to avoid unintended behaviour. In contrast, graph search algorithms like Dijkstra's and A\* algorithms fall under the category of search and optimization algorithms, which are used to find the shortest path in a graph. While reinforcement learning is an example of a machine learning algorithm that falls under the category of artificial intelligence algorithms.

# The selected approach.

For the given problem of finding the shortest path between the Boss's current location and his destination, we will use the A\* algorithm. A\* is a heuristic search algorithm that estimates the cost from a node to the destination. It maintains a priority queue of unvisited nodes and selects the next node with the smallest f-score. A\* algorithm uses a good heuristic function to estimate the time required to reach the destination, making it suitable for this problem.

Algorithm:

The algorithm begins with the Boss's current location as the starting point and his destination segment as the goal. The starting point is added as an extra node, which acts as a processor for the current segment node Si of the Boss's car. Each street segment is represented as a node in the algorithm, and the edges between a node Si and its predecessors are weighted by the actual time required for the segment Si. The actual time is calculated using the formula: Actual time = T + ((N > C) ? (N – C)\*D : 0), which takes into account any delay time caused by the number of cars on the segment. If a node's C value exceeds a certain threshold, the program removes the connection.

The A\* algorithm uses a priority queue to maintain the nodes' order, with the priority being the total estimated cost to reach the goal through that node. The estimated cost is the sum of the cost of reaching the current node plus the estimated cost of reaching the goal from that node. The algorithm terminates when it reaches the destination node or there are no more unvisited nodes.

Data structures used:

The system will use a graph data structure to represent the street plan, with each street segment represented as a node in the graph. The edges will be represented as weighted edges between the nodes. The priority queue data structure is used to maintain the nodes in the order of their f-score, and a map data structure is used to store the segments and their information.

Advantages:

A\* algorithm is better than other algorithms such as reinforcement learning and Dijkstra's algorithm because it can prune parts of the search space that are unlikely to lead to the goal, making it faster than Dijkstra's algorithm. It is also guaranteed to find the shortest path if the heuristic is admissible, meaning it never overestimates the true distance to the goal. Additionally, it reduces the number of nodes that need to be explored, making it especially useful in large graphs.

Disadvantages:

One disadvantage of A\* is that it requires a good heuristic function to be effective. If the heuristic is poor, the algorithm may take longer to converge than Dijkstra's algorithm. And A\* algorithm requires additional data structures to store the heuristic information, which can increase the memory usage compared to Dijkstra's algorithm. Specifically, A\* algorithm stores a priority queue that contains nodes to be expanded based on their estimated distance to the goal, while Dijkstra's algorithm stores a priority queue that contains nodes to be expanded based on their actual distance from the start node. Additionally, like Dijkstra's algorithm, it requires storing all the edges in the graph.

To ensure a good heuristic, next points should be considered:

The heuristic should never overestimate the actual cost to reach the destination. This means that the heuristic function should be lower or equal to the actual cost. If the heuristic overestimates the actual cost, then the algorithm may explore unnecessary nodes, which can slow down the search.

The heuristic should be consistent with the edge costs. This means that for any edge (u, v) with cost c(u, v), the estimated cost from u to the destination should be less than or equal to the estimated cost from v to the destination plus the cost of the edge c(u, v). Mathematically, this can be expressed as h(u) <= h(v) + c(u, v). A consistent heuristic ensures that the algorithm will always choose the optimal path.

A good heuristic should incorporate domain knowledge about the problem. For example, in this case, the heuristic could consider factors such as the traffic patterns, the speed limits on the roads, the number of lanes on each road, etc.

And it's important to experiment with different heuristics to find the best one for the problem. You can try different heuristics and compare their performance on a set of test cases. This will help you identify which heuristic works best for this specific problem.

# Estimates of Complexity.

Time Complexity:

The time complexity of the A\* algorithm depends on the quality of the heuristic function and the size of the graph. In the worst case, where all nodes need to be explored, the time complexity will be O(b^d), where b is the branching factor of the graph, and d is the depth of the solution. However, in practice, A\* algorithm performs much better than this worst-case scenario. The algorithm explores fewer nodes than Dijkstra's algorithm because it uses the heuristic to prioritize the nodes to visit. Therefore, the time complexity of A\* is usually much lower than O(b^d) and is often close to O(b^(d/2)).

The space complexity of A\* algorithm is proportional to the size of the graph, as it needs to store all the nodes and their connections. Therefore, the space complexity of the algorithm is O(V+E), where V is the number of nodes in the graph and E is the number of edges.

Justification:

A\* algorithm has a lower time complexity than Dijkstra's algorithm in most cases because it uses a heuristic function to guide the search, reducing the number of nodes that need to be explored. In the worst-case scenario, the time complexity is the same as Dijkstra's algorithm, but in practice, the algorithm performs much better due to the use of the heuristic function. The space complexity is proportional to the size of the graph and is therefore O(V+E). This is reasonable as the algorithm needs to store all the nodes and their connections to perform the search. The space complexity can be reduced by using a sparse graph representation or by storing only the necessary information about each node. However, in this problem, the size of the graph is not expected to be very large, so the space complexity should not be a significant concern.

# Initial Assessment of the Possible Implementation:

## 7.1 Challenges for another programmer:

1. Understanding the A\* algorithm and implementing it efficiently.
2. Creating a graph data structure that represents the street plan and ensuring that it is accurate and up to date.
3. Developing a good heuristic function that considers all relevant factors and is consistent with the edge costs.
4. Debugging the program in case of any errors or bugs.

## 7.2 Ease and Speed of Implementation:

1. The implementation of the A\* algorithm should not be very difficult for a programmer who has experience with graph-based algorithms and data structures.
2. Creating a graph data structure and populating it with street plan data may be time-consuming and require some research.
3. Developing a good heuristic function may require some trial and error and experimentation to find the most efficient one.
4. The hardware requirements for the program should not be very high, and it should be able to run efficiently on most modern computers.
5. The real-world response time of the program will depend on various factors such as the size of the graph and the efficiency of the heuristic function. However, since the A\* algorithm is known to be efficient and effective, the response time should be reasonable for most cases.

## 7.3 Hardware Requirements:

The hardware requirements for this program should not be very high. A modern computer with a standard processor and memory should be sufficient to run the program efficiently. However, if the street plan is very large, the program may require more memory and processing power to handle the graph and perform the necessary calculations.

## 7.4 Real-World Response Time:

The real-world response time of the program will depend on various factors such as the size of the graph, the efficiency of the heuristic function, and the hardware resources available. However, since the A\* algorithm is known to be efficient and effective, the response time should be reasonable for most cases. In general, the response time should be in the order of milliseconds to a few seconds for most street plans.

# 8. Test samples.

## 8.1 One segment example:

Input:  
20

20

20 7 210 5 18 45

Output:

0

Image of the graph for example 8.1:

Diagram

Description automatically generated with low confidence

## 8.2 Example with 2 existing travel routes:

Input:

20

1

20 5 120 2 10 45

0

1 3 30 2 5 10

14 3

2 5 180 3 7 60

20

3 8 240 4 17 45

2

15 10 300 6 17 35

20

14 8 180 2 17 35

15

Output:

4

570

20 2 3 1

Image of the graph for example 8.2:  
A picture containing necklet, accessory

Description automatically generated

8.3 Example with no existing travel routes:  
Input:

20

1

20 5 120 2 10 45

0

1 3 30 2 5 10

14 3

2 0 180 0 7 60

20

3 8 240 4 17 45

2

15 10 300 6 17 35

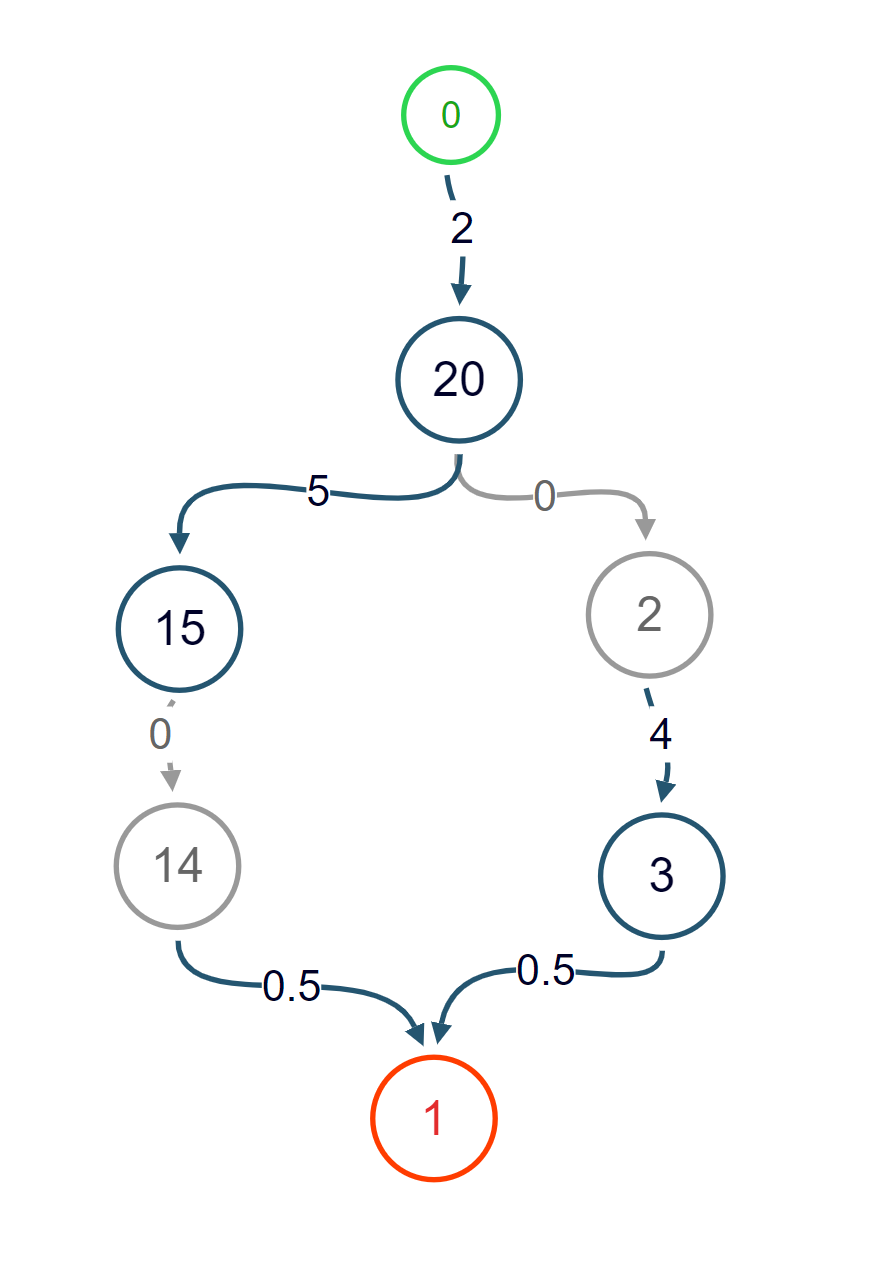
20

14 0 180 0 17 35

15

Output:

0

Image of the graph for example 8.3:  


## 8.4 Real-life mini example:

Input:  
20

22

20 5 120 2 10 45

0

22 3 30 1 5 30

9 10 12

2 5 180 3 7 60

20

3 8 240 4 17 45

2

15 8 300 3 17 40

20

5 8 180 4 17 45

3

14 8 180 2 17 35

15

30 0 200 0 17 55

33

6 8 180 1 17 30

5 8

10 8 180 4 17 45

7 14 4

13 8 200 16 17 50

11 30

1 0 30 0 5 35

15

31 3 30 0 6 25

1 34

32 3 30 0 6 25

31

33 3 30 0 6 25

32

34 3 30 0 6 25

33

8 5 120 4 10 45

7 14 4

7 5 60 5 10 45

8 5

11 5 120 3 10 45

7 4 14

4 5 180 4 10 45

30 11

9 5 180 3 10 45

6

12 5 180 2 10 45

13

Output:

5

810

20 15 14 10 22

Images of the real-world map and the graph for example 8.4:  
Map

Description automatically generated

Diagram, shape

Description automatically generated

## 8.5 Relevant tests with large data volume

To ensure the program's reliability and ability to handle any real-world scenario, we can use automated testing frameworks to run a suite of acceptance tests for large data volumes. We can also generate random test cases, including edge cases such as no valid paths between the starting and destination segments, and use them to test the program's performance. It is important to test the program on various input sizes and data types.

For testing with large data volumes, we can generate large input files with random segment information and run the program on them to check that it can complete execution in a reasonable amount of time. We can increase the number of segments and their length in the input file to create more relevant tests. To check that the code passes the tests, we can compare the output of the program with the expected output for each input file.

# Conclusions.

In this project, the goal is to develop a computer-assisted route planning system that provides real-time information about traffic conditions on various street segments in Riga, Latvia, to enable a high-ranking individual, the Boss, to travel faster than other drivers. The proposed system should take into account various factors such as the maximum number of cars that can drive through a street segment without experiencing any delay, the number of cars currently driving through the segment, the maximum number of cars that can be squeezed on the segment during peak traffic, and the delay time that is required for every car above the number of cars that can drive through the segment without experiencing any delay.

The project proposes using the A\* algorithm as a solution alternative, which is a heuristic search algorithm that combines the best features of uniform-cost search and greedy best-first search. The algorithm requires a good heuristic function to be effective, and it can be especially useful in large graphs as it reduces the number of nodes that need to be explored. However, like Dijkstra's algorithm, it requires storing all the edges in the graph.

The project outlines the algorithmic details of the A\* algorithm, including the key parts of initialization, selection, update, termination, and path reconstruction. The data structures used in the project include a graph data structure to represent the street plan, a priority queue to maintain the nodes in the order of their f-score, and a map data structure to store the segments and their information.

And the proposed solution using the A\* algorithm is appropriate, given the specific requirements of the problem. However, the system's effectiveness and potential impact will depend on its assumptions, limitations, and challenges, such as maintaining accurate and up-to-date segment information, its focus on optimizing travel time for a single user rather than reducing congestion or optimizing traffic flow, and the need for efficient processing of large volumes of data. Therefore, careful consideration must be given to these factors when assessing the system's overall effectiveness and potential impact.