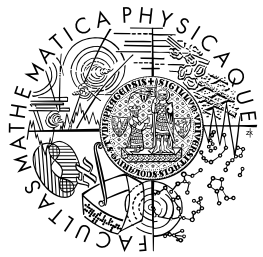


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Adam Polický

Scalable dynamic graph-based vehicular routing

Department of Software Engineering

Supervisor of the bachelor thesis: RNDr. Miroslav Kratochvíl, Ph.D.

Study programme: Computer Science

Study branch: General Computer Science

Prague 2021

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I thank my family and loved ones for supporting me and pushing me forward. Most of all, I thank my supervisor Miroslav Kratochvíl for his advice, patience and enthusiasm that motivated me to write this thesis.

Title: Scalable dynamic graph-based vehicular routing

Author: Adam Polický

Department: Department of Software Engineering

Supervisor: RNDr. Miroslav Kratochvíl, Ph.D., Department of Software Engineering

Abstract: Algorithms for finding shortest paths in large graphs form an essential part of many modern navigation and routing systems. In vehicular navigation, the problem is complicated by dynamic nature of the network caused by road closures and changes in traffic, preventing application of many common speed-up techniques. The aim of this thesis is to design an algorithm for finding paths in large graphs that gains efficiency and scalability by minimizing the number of visited graph objects in storage. This was achieved by iteratively simplifying the graph into a multi-layered approximative structure, and developing a modification of Dijkstra's algorithm that allow efficient navigation in the structure. The results show that the proposed method examines 4× less graph objects than A* and 14× less than Dijkstra, achieving better performance at the cost of slightly longer discovered paths. Additionally, the layered structure is able to accommodate changes in the base graph, allowing the algorithm to work on a changing network without costly recomputations.

Keywords: vehicle routing, navigation, graph algorithms, shortest-path finding

Contents

Introduction **3**

1 Pathfinding in large graphs **5**
1.1 Finding paths with Dijkstra's algorithm 5
1.2 Pathfinding optimizations 7
1.2.1 A* algorithm 8
1.2.2 Bidirectional search 8
1.2.3 Landmarks 10
1.2.4 Reach for A* 11
1.2.5 Contraction hierarchies 12
1.2.6 Highway dimensions 13

2 Vehicle routing and congestion avoidance **15**
2.1 Types of routing 15
2.2 Dynamic routing 16
2.3 Practical dynamic routing algorithms 17

3 Scalable algorithms on layered network **19**
3.1 Layered network graph 21
3.1.1 Aggregation algorithm 22
3.1.2 Handling dynamic changes 24
3.2 Pathfinding in the layered network 27
3.2.1 Dijkstra's algorithm with precision limit (DwPL) 28
3.3 Network updates 29

4 Benchmarks and results **33**
4.1 Preprocessing 33
4.2 Pathfinding in layered network 34
4.3 Complexity of path updates 38
4.4 Routing efficiency 38

Conclusion **41**

A Installation and use of the attached program **45**
A.1 How to build 45
A.2 Preparing data for benchmark 45
A.2.1 Note on the use of graphic functions 46
A.3 Starting the program 47
A.3.1 Graph loading 47
A.3.2 Pre-simulation phase 48
A.3.3 Post-simulation phase 48
A.4 Sample use of the program 48
A.5 Commands & Parameters 50
A.5.1 Commands 50
A.5.2 Parameters 50

Introduction

Ever since Dijkstra's algorithm was presented, it still holds its place in the field of pathfinding in graphs. A typical modern use-case for fast pathfinding is navigation in maps, used by millions of road users, which increases the need for a fast and reliable navigation system.

Vehicles should be able to react to real time events quickly. The network state should also be taken into consideration. Moreover routes should prevent other congestions along the line. Another problem is to spread the vehicles across the whole network in the shortest possible time.

To improve the performance of the search for best paths many speed-up techniques for the Dijkstra-style search algorithms have been invented. However, application of many of the invented speed-up techniques is complicated by the changing network conditions caused congestions, road closures and car accidents. Specialized routing and congestion avoidance algorithms are studied to overcome situations caused by these conditions.

This thesis proposes a pathfinding algorithm with quick response times working on dynamically simplified graph structure. The routes are computed with respect to the real time information and are based on the prediction of the future network state thanks to reservations vehicles make. This approach is supposed to improve the network throughput.

Finally, this thesis demonstrates the capabilities of the proposed algorithm in a simulated road network. The lengths of the approximate routes computed by the algorithm are only about 16% longer on average then the shortest ones. Furthermore, the number of scanned graph objects (e.g. vertices opened in Dijkstra's algorithm) required for the computation is significantly lower than in the case of A* and Dijkstra's algorithm, reducing the number of opened objects to less than 25% in case of A* and less than 8% in case of Dijkstra. The computation time is reduced to around 8%, resp. 3%, of computation time of A*, resp. Dijkstra's algorithm.

Layout of the thesis Chapter 1 (Pathfinding in large graphs) gives an overview on existing pathfinding algorithms and speed-up techniques that

improve query times.

Chapter 2 (Vehicle routing and congestion avoidance) lists the problems of traffic routing and shows categorization of routing approaches.

In chapter 3 (Scalable algorithms on layered network) the proposed pathfinding algorithm and its dependencies are presented.

Several variants of the algorithm are tested in chapter 4 (Benchmarks and results) where more results can be found.

Chapter 1

Pathfinding in large graphs

A reduction of road networks to graphs is commonly used for computer based pathfinding. Roads or streets are represented by edges and intersections are represented by vertices. This chapter gives an overview on pathfinding algorithms and speed-up techniques.

1.1 Finding paths with Dijkstra's algorithm

Definition 1 (Weighted graph)**.** *Weighted graph (V,E) is a graph in where each edge* $e \in E$ *is given a numerical weight [13]. Weight can be represented as a function:*

$$weight: \ E \to \mathbb{R}_+$$

Definition 2 (Shortest path)**.** *Path in a graph connecting two vertices is a sequence of edges such that following edges share a common vertex, the first edge of the sequence is adjacent to one of the vertices and the last edge is adjacent to the other vertex. The shortest path between two vertices is such a path that the sum of edge weights is minimal.*

Dijkstra's algorithm is well known in the field of pathfinding. It was published by Edsger Dijkstra in 1959[3]. It uses the basic idea of breadth first search but adapts it on edges with weights.

The algorithm can be used in multiple purposes. The original variant was focused on finding the shortest path between two vertices. Another variant computes shortest paths between a single fixed vertex and all others.

Dijkstra's algorithm works on a graph structure and uses a priority queue that determines the order of vertex exploration. The algorithm receives a starting vertex (*source*) and possibly a destination vertex (*target*). Vertices might be in one of three states in the following order: *not found*, *open*, *closed.*

The algorithm takes the source vertex, finds all adjacent vertices, changes their state to *open* and pushes them together with the distance from the source to the priority queue. The state of source vertex is set to *closed*. In the next iteration a vertex on the top of the queue, the one with the shortest distance from the source, gets popped out of the queue and all of its neighbors are explored. The computation follows until the target vertex is reached or there are no vertices left to explore, i.e. no open vertices. The pseudocode for the algorithm can be seen in algorithm 1.

Algorithm 1 Dijkstra's algorithm

```
procedure DIJKSTRA(vertices, source, target = null)
    for v ∈ vertices do
        state(v) ← not found
        distance(v) ← ∞
        predecessor(v) ← undefined
    end for
    distance(source) ← 0
    PQ.push(source)
    while PQ is not empty do
        v ← PQ.pop()                    ▷ v is the vertex with the smallest distance
        if v = target then
            return the path from source to target
        end if
        for n ∈ neighbors(v) do
            if distance(n) > distance(v) + weight(v, n) then
                distance(n) ← distance(v) + weight(v, n)
                predecessor(n) ← v
                state(n) ← open
                PQ.push(n)
            end if
            state(v) ← closed
        end for
    end while
end procedure
```

As already mentioned Dijkstra's algorithm explores vertices in order of their distance from the source vertex. Figure 1.1 shows the illustration of the computation of Dijkstra's algorithm.

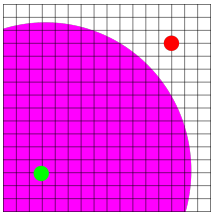


Figure 1.1: Illustration of the search of Dijkstra's algorithm. Purple corresponds to opened vertices, green is for the source and red is for the target.

Performance of Dijkstra's algorithm on large graphs

Dijkstra's algorithm can not be used on graphs with negative edge weights. In the pseudocode the vertex with the smallest distance from source is taken, expanded and closed. The assumption is that the reached vertex must have been reached using the shortest path possible. If there were edges with negative weights, there might be a shorter path leading towards a vertex that has already been closed. Closed vertices can not be reopened. Since roads do not have negative weights this is not an issue.

The main limitation of the Dijkstra's algorithm is its performance. In graph with V vertices each vertex is expanded at most once during the computation. The time complexity is $\mathcal{O}(V^2)$, resp. $\mathcal{O}(V \log V + E)$ if an efficient priority queue is used. To improve the performance, the number of scanned vertices must be reduced.

Other pathfinding algorithms

There are also another algorithms that can be used for finding the shortest path in a graph, such as Bellman-Ford or Floyd-Warshall algorithm. Bellman-Ford computes shortest paths from a single vertex to every other vertex. Floyd-Warshall computes shortest paths for all pairs of vertices. Both are not suitable in this case mainly because of their performance and difficulty of use on a changing graph structure which is why they are not addressed any further.

1.2 Pathfinding optimizations

Here, several common approaches for making the path finding more efficient are reviewed, namely: A* (1.2.1), bidirectional search (1.2.2), ALT (1.2.3), reach (1.2.4), and contraction hierarchies (1.2.5). In addition, the good performance on real road networks of the latter two is explained by the notion of highway dimension (1.2.6).

1.2.1 A* algorithm

A* algorithm is a variant of Dijkstra's algorithm that is modified to provide better performance when a heuristic is available for the graph. It was published in 1968 by Hart, Nilsson, and Raphael [7]. It is broadly used in the field of pathfinding algorithms mainly due to its efficiency. The difference between Dijkstra's algorithm and A* is that A* uses some additional information about the vertices that might have an impact on the order in which the vertices are expanded. It is called heuristics. The heuristic function should help to reduce the number of expanded vertices by directing the search towards the target. It must also be *admissible* for A* to guarantee the optimal result, which means that each vertex v gets a heuristic value that is less than or equal to the "true distance to the target".

The combination of the vertex's actual distance from source and the heuristic value is then used in the priority queue to determine the expansion order. If the heuristic function assigned only 0 to each vertex the computation would effectively be the same as of Dijkstra's algorithm.

A commonly used heuristic for A* is the Euclidean distance of two objects in Euclidean space, which are represented by vertices in the graph. This function favors vertices that are closer to the target. However Euclidean distance has its limitations. Figure 1.2 is one of the examples. In the following section another heuristic function for A* will be shown.

(a) (b)

Figure 1.2: Various situations that occur in A* algorithm. Part 1.2a shows that the search is directed towards the target. In part 1.2b it can be seen that even though some of the open vertices are not directly on the shortest path, the number of open vertices is still reduced.

1.2.2 Bidirectional search

Dijkstra's algorithm expands vertices by their distance from the source. It can be thought of as expanding in circles. Each iteration means setting the radius to the distance from source of the expanded vertex. The aim of bidirectional search is to reduce the number of expanded vertices by searching in two directions – from source to target and from target to source. For example in a graph where

Algorithm 2 A* algorithm

```
Ensure: h(x) a heuristic function assigning values to vertices
  procedure ASTAR(vertices,source,target = null)
    for v ∈ vertices do
      state(v) ← not found
      distance(v) ← ∞
      predecessor(v) ← undefined
    end for
    distance(source) ← 0
    PQ.push(source)
                ▷ queue of open vertices, sorted by f(x) = distance(x) + h(x)
    while PQ is not empty do
      v ← PQ.pop()                ▷ v has the smallest e = distance(v) + h(v)
      if v = target then
        return the path from source to target
      end if
      for n ∈ neighbours(v) do
        if distance(n) > distance(v) + h(v) then
          distance(n) ← distance(v) + h(v)
          predecessor(n) ← v
          state(n) ← open
          PQ.push(n)
        end if
        state(v) ← closed
      end for
    end while
  end procedure
```

the search tree branches with coefficient b and the distance of vertices from each other is d, the number of opened vertices is reduced from $\mathcal{O}(b^d)$ to $\mathcal{O}(b^{d/2})$. The two search branches should meet somewhere in the middle.

The computation of each branch works the same as Dijkstra's algorithm, or even A*. Each branch has its own priority queue. A vertex from top of priority queue is expanded, adjacent vertices opened, pushed to appropriate priority queue and the vertex gets closed. Since there are two priority queues, vertex with the smaller distance from both queues is expanded. The natural stopping condition is that a vertex is reached by both branches.

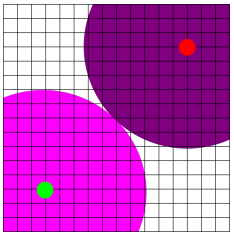


Figure 1.3: Illustration of bidirectional search.

1.2.3 Landmarks

In addition to heuristics, graph preprocessing might also help to speed up the search. This section describes a preprocessing technique that yields heuristic values for A*. Its name is ALT which is an acronym for A*, Landmarks and Triangle inequality[2].

Landmarks are a subset of vertices of a graph. The distance between every landmark and every other vertex of the graph is computed and stored. This information is then used to compute the heuristic values of vertices for A*. The value the heuristic function returns for a vertex v corresponds to an estimated lower bound of the distance between v and the target t. The heuristic value depends only on the distance between any landmark l and t is known, the distance between l and v is known. Then from the triangle inequality follows that:

$$\text{distance}(\texttt{v}, \texttt{t}) \leq \text{distance}(\texttt{l}, \texttt{t}) + \text{distance}(\texttt{l}, \texttt{v})$$

This is computed for each landmark and the maximum value is chosen as the estimated lower bound on the distance from v to t.

The number of landmarks depends on implementation. Delling and Wagner [2] measured how the number of landmarks affects response times and memory usage. The more landmarks, the more space is used. On the other hand, with more landmarks, the better the estimates can be. Hence, the response times are reduced.

Landmark section

The choice of optimal landmarks is a difficult problem to solve and it is not the aim of this thesis. There are more sophisticated heuristics such as *avoid* which iteratively starts a randomly rooted shortest path tree and selects the branch that avoids existing landmarks, or *maxCover* which selects the set of vertices that have the maximum coverage[2, 4]. Here are some of the more simple methods reviewed by Fuchs [4].

The farthest selection is a greedy method. Randomly choose a vertex v_0. From this vertex find the longest possible path. It end is in vertex v_1 which is now the first landmark. Then choose next vertex v_2 so that its geographic distance from all the landmarks is maximum.

In grid selection the graph is divided into the same number of cells as the number of coveted landmarks. In each cell the vertex with the greatest distance from imaginary center of the graph is selected. This should evenly spread the landmarks across the whole graph. Thus the coverage should be better and the possibility of having tighter distance estimates should be higher.

Landmarks and A*

The implementation of ALTs query phase is relatively simple. The A* remains the same, only the heuristic function changes from h(x) used in algorithm 2 to ALT Potential (see algorithm 3). It returns the upper bound on the distance between the vertex and the target derived from the triangle inequality.

Algorithm 3 ALT Potential

```
function altPotential(v)
    return max_{l∈landmarks}(|distance(l, v) − distance(l, target)|)
end function
```

1.2.4 Reach for A*

Goldberg, Kaplan, and Werneck [6] described another speed-up technique for pathfinding in graphs based on Dijkstra's algorithm that is called reach. The idea here is to consider only such vertices and edges that are likely to be on the shortest path, i.e. vertices that are able to "reach" the target.

Vertex v divides a path p ($v \in p$) in two parts p_1 and p_2. The reach of v is defined for every path that uses v as:

$$\text{reach}_p(v) = \min(\text{length}(p_1), \text{length}(p_2))$$

$$\text{reach}(v) = \max_{p \in \text{paths}}(\text{reach}_\text{p}(\text{v}))$$

In the query phase bidirectional Dijkstra's algorithm is used and vertices are pruned with respect to their reach. Whether in the forward search vertex v gets pruned or not, i.e. is inserted in the forward priority queue, depends on meeting the following condition:

$$\text{reach}(v) > \min(\text{distance}_\text{F}(v), \text{distance}_\text{B}(b))$$

where b is the vertex on top of the backward priority queue. For this condition to hold v must lie on such shortest path that is at least as long as the distance of b from the target. If the condition did not hold, v would be only on such shortest path that would end somewhere not reaching the target. Goldberg, Kaplan, and Werneck [6] showed that ALT is competitive with reach for smaller graphs (< 1M edges), but for large graphs (> 20M edges) reach is more than 20 times faster.

1.2.5 Contraction hierarchies

Geisberger et al. [5] described the last speed-up technique to be mentioned in this thesis called contraction hierarchies. Its goal is again to reduce the number of scanned vertices. A shortcut between two vertices of the length of the shortest path between these vertices is added to the graph.

Preprocessing is iterative. Choose a vertex v. Compute shortest paths between all pairs of v's neighbors. If such a path uses v, a new shortcut between this pair of vertices is added. Its weight is the weight of the path. Vertex v is labelled as processed, is removed from the vertices of preprocessing phase and is numbered in the order it was contracted. This vertex is ignored in the following iterations because all edges adjacent to it were either contracted or were not part of shortest path.

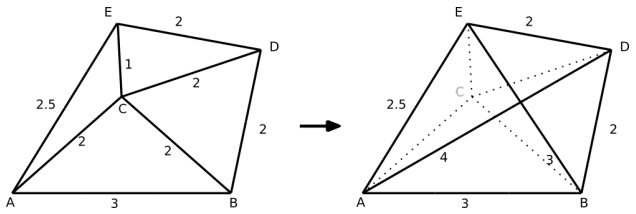


Figure 1.4: A contraction of vertex C

There is a number of heuristics to choose from that specify the order of contraction. Each contraction should reduce the number of edges as much as possible. Intuitively, the vertex with the highest degree should be contracted first. Geisberger et al. [5] state that it would be best to contract the vertices evenly across the graph. Doing so yields a structure where shortcuts do not overlap which is the desired result. Unpacking a shortcut is simple with an additional information containing the two shortcuts or edges that formed the new one.

This speed-up is used with bidirectional Dijkstra's algorithm on a graph structure where the vertices are the same and the edges are the original edges and shortcuts. When scanning a vertex v, only vertices that were contracted later will be considered. The computation ends when there is no such vertex left. For each vertex scanned v, this computes an upper bound on its distance

from s and t – distance$_s(v)$ and distance$_t(v)$. Among these vertices, the one that minimizes the sum of these distances is selected. Its name is u and is on the shortest path. Next, the paths from s to u and from u to t are computed in the same way.

1.2.6 Highway dimensions

The above speed-up techniques compute shortest paths very well on real road networks. Abraham et al. [1] state it can be explained using the notion of highway dimension. Graph has highway dimension of h if for every r and for every ball of radius of $4r$ and all paths of length greater than r inside this ball there exist a set of vertices S that covers all these paths and $|S| \leq h$. Meaning that there is a small set of vertices that occur on all paths longer than r.

Then there is defined shortest-path cover (SPC), respectively (r,k)-SPC, which is a set S of vertices that covers all shortest paths with length between r and $2r$ and $S \leq k$.

This notion is used to explain the performance of both contraction hierarchies and reach. Having small highway dimension guarantees that fewer objects are scanned by the aforementioned speed-ups. During the preprocessing phase of both techniques the vertices are ordered into layers by their SPC. Vertices from lower layers will not be used for shortest paths that have a certain length. This means that reach of a vertex is bounded by the mentioned length.

Chapter 2

Vehicle routing and congestion avoidance

The previous chapter discussed speed-up techniques for pathfinding in graphs. If all networks, whether road, rail or footpaths, had unlimited capacity, this chapter would be sufficient, but that cannot be assumed in the real world. The vehicle cruising speed depends on the speed limit and the occupation of a road. For example, a vehicle may not travel faster than a speed-limited vehicle that travels before it on the same road, and the road may have capacity for only a limited amount of vehicles in a given time interval. To be able to represent this, the edge weights will represent the expected time of traversing the edge by the vehicle. It is supposed that vehicles receive route that is optimal, i.e. takes the shortest amount of time. This must also hold for multiple vehicles, hence the current network state must be taken into account.

Definition 3 (Route schedule)**.** *A route schedule as a set of edges and corresponding times when these edges are planned to be used.*

Definition 4 (Graph with capacities)**.** *Graph with capacities is a graph where each edge has a numeric value (capacity) corresponding to the maximum amount of vehicles present on it in a given time interval.*

Definition 5 (Traffic flow)**.** *Traffic flow is a measure defined as:*

$$traffic\ flow = \frac{number\ of\ passing\ vehicles}{|time\ segment|}$$

2.1 Types of routing

Optimal routing The goal the vehicle routing problem is to determine the optimal set of routes for delivering goods to a set of customers. The problem is

related to scheduling problems and is known to be NP-hard[11]. In the context of this thesis, NP-hardness of exact routing problems suggests that, for practical deployments and problem sizes, we should aim for implementing an approximate algorithm.

Independent routing Independent routing is a model where vehicles use a dynamically aggregated information about the road network, i.e. road occupation or congestions. Based on this information, the vehicles are given best routes and set on their way. The model responds well to events on the roads in real time. This allows vehicles to be rerouted and reach their destination earlier. The rerouting overhead is small due to the lack of interaction with other vehicles. Since vehicles in this model do not consider others, the network information is most of the time delayed. Meaning, it takes some time for the routing system to process information from vehicles stuck in traffic and send it to other vehicles. There are no warranties that new routes will be faster or not.

This type of routing fails in cases where vehicles need to cooperate to avoid causing another problem by simply avoiding the problem on the road. There is a textbook example situation where independent routing fails. There happens a congestion on a highway. The navigation system decides that it is quicker to exit the highway and continue on a country road. Suppose the vehicle is not the first one and someone had the same idea even sooner. By definition 4, each road has its capacity and with more vehicles coming here, the country road is likely to be congested too.

Traffic routing Traffic routing is the process of improving traffic flow by distributing vehicles evenly across the network[8]. This may be a setback for some vehicles as their route may not be optimal. The difference between traffic and independent routing is that traffic routing is dedicated to improving traffic flow throughout the network while making some vehicles use sub-optimal routes.

Dynamic routing Dynamic traffic routing handles real time changes of the road network in order to improve the traffic flow.

2.2 Dynamic routing

Dynamic traffic routing handles real time changes of the road network in order to improve the traffic flow. Changes are either recurrent or non-recurrent. Recurrent being rush hours and non-recurrent traffic jams. Isa, Mohamed, and

Yusoff [8] summarized that there are two criteria by which routing algorithms are categorized: environment and routing strategy.

Traffic routing can be implemented in two types of environment: deterministic and stochastic.

Deterministic environment is fully observable and the state it is in determines the next state.

Stochastic environment is partially observable and the next state is nondeterministic. Changes in the network are saved and used for routing purposes such as computing new routes.

A routing algorithm should be able to react to changes during the routing process. It should be able to reroute vehicles whose routes are affected by the changes.

Offline rerouting strategy takes into consideration the current state of the network. With respect to this states new routes are planned and are followed by the vehicles. The routing process stops when vehicles leave their starting points.

Online rerouting strategy pre-planes routes in the same way as the offline strategy. After the vehicles leave towards their destinations the routing algorithm keeps monitoring the network state. Should there be faster routes, the vehicles get rerouted. The routing process stops when the vehicles arrive to their destinations.

The type of rerouting can be also divided into other two categories: reactive and proactive.

Reactive approach to routing determines next steps based on the current state.

Proactive approach uses the current state and historical data to predict the next state. The proactive type of routing tries prevent congestions.

2.3 Practical dynamic routing algorithms

This section summarizes two routing strategies where the categorization is visible.

Online, stochastic

Wang, Djahel, and McManis [12] suggested the following routing strategy. Vehicles are given the best route computed by Dijkstra's algorithm based on metrics such as: fuel consumption, travel time, travelled distance, ...The vehicles then start driving.

A vehicle follows its initial route until a real time event report is received, the route should be updated. New route is computed and the traversal continues. This routing strategy works as the computer interruption. The event information is pushed only to related vehicles so they do not have to check network status. New routes are computed by Dijkstra's or a heuristic algorithm based on the time it takes for the vehicle to get to the next road junction. Dijkstra gives the exact route, heuristic algorithm gives an estimate.

Online and offline, stochastic

Taniguchi and Shimamoto [10] focused on solving the vehicle routing problem (VRP) with time windows and real time road network information. The goal of their work was to create a routing and scheduling model for a delivery service that delivered goods for customers in a certain time window. Two types of models were compared: forecasted and dynamic. Both models used aggregated network information.

The forecasted model plans routes for each carrier beforehand. The routes are planned using historical data and information about the current state of the network. This routing strategy is considered to be offline because the routing process stops when the carrier leaves the depot.

The dynamic model uses the same principles as the forecasted model but the routing process continues even after the carrier departure. Each time the vehicle arrives to a customer, its route is recalculated using the real time information. The order of visiting customers and route may be change if there was a congestion or different problem with the previous route. The routing strategy of this model is online since the routes are being updated.

Chapter 3

Scalable algorithms on layered network

In section 1.2 (Pathfinding optimizations) some speed-up techniques were mentioned. They work well on graphs where the weights of edges do not change at all or change by an insignificant amount. If they changed, all three of these—ALT, reach and contraction hierarchies—might work on a graph structure that is no longer valid. Hence, the results might not be entirely correct and the preprocessing might have to be redone.

Finding paths with plain A* might be a solution to the invalidated preprocessed data. However, the number of scanned vertices and edges is quite high for longer distances.

In this section, we propose a structure that supports dynamic updates that are sufficiently frequent to support the dynamics of the real world, while allowing for fast pathfinding that visits fewer vertices. The structure will be constructed by gradually contracting/aggregating edges into different layers according to the area they occupy. In the resulting structure, the vehicles will be able to search for exact paths in close proximity, further away they will be able to use contracted/aggregated hyperedges, which will speed up the search.

In addition, if vehicles cooperate with the navigation system and upload information about the edges they intend to use, they will not have to put reservations on all the edges they plan to be on, but will be able to report the approximate position in higher layers. This will reduce the number of reservations they make.

To navigate on this structure, we propose a modification of the Dijkstra algorithm—Dijkstra algorithm with precision limit (section 3.2.1)—the performance of which is measured in chapter 4 (Benchmarks and results).

Layers

Exact routes can be found in the original graph. The proposed structure should contain both precise and imprecise information about the network. This is done by creating a layered structure of the graph. The original graph forms the base layer of the structure. Each higher layer is formed from the previous by its simplification which is done by grouping suitable adjacent edges. These groups are called hyperedges.

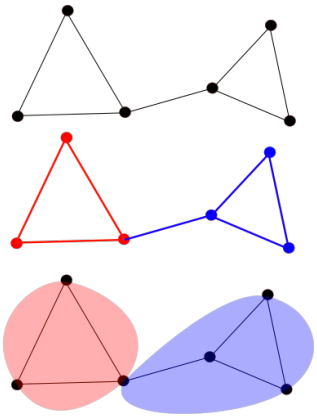


Figure 3.1: Edge grouping

The connection between vertices on the base layer is exact. An edge belongs to, or makes contact with exactly two vertices. The hyperedges on the upper layers can make contact with 0 or more hyperedges.

The difference in structure between the base and all the other layers might be an inconvenience while pathfinding in such a structure. For simplicity, the base layer—consisting of vertices and edges—will be transformed into the hyperedge representation. The edges will be viewed as hyperedges and vertices will be viewed as their contact points from which the neighborhood relationship is derived. The number of scanned objects will be reduced since the scanning will be performed only on hyperedges. This customization makes the graph structure undirected. The ability to distinguish between one-way and two-way streets disappears.

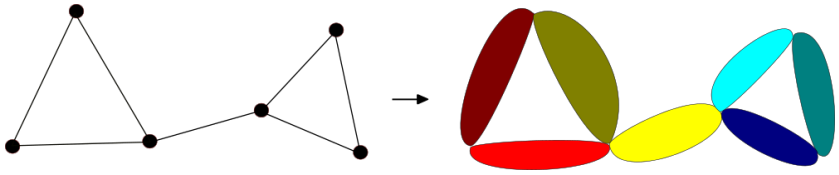


Figure 3.2: Transformation of vertex-edge based graph to hyperedge based graph.

3.1 Layered network graph

We define the previously mentioned terms more formally:

Neighborhood Edge e_1 is the neighbor of edge e_2 if there exists a path connecting e_1 and e_2 that consists only these two edges. The same holds for hyperedges.

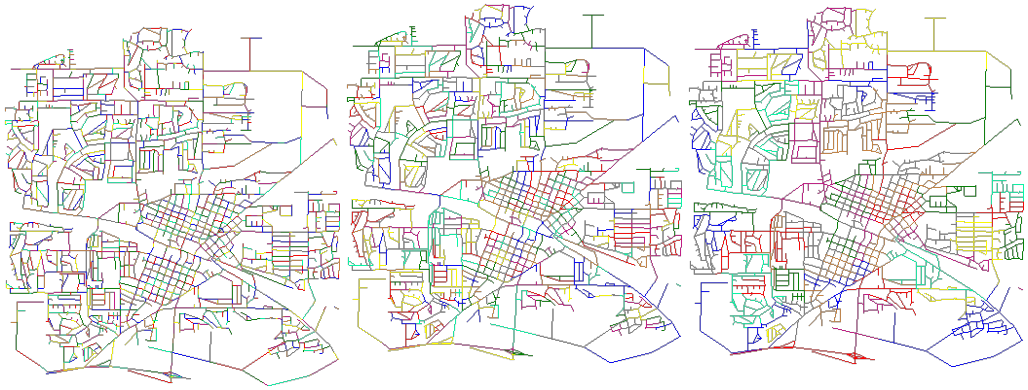
Hyperedge An edge is a hyperedge. Recursively hyperedge is a set of hyperedges.

Parent-child relationship Hyperedge p is the parent of hyperedge c and c is the child of p if p was formed from a set of hyperedges s such that $c \in s$ and layer(p) = layer(c) + 1.

Weight The weight of hyperedge is the sum of weights of its children. The base layer hyperedge does not have any children. Its weight is the weight of the edge it was formed from.

Layer Layer is a graph where the elements are hyperedges. There is a neighborhood relation between the hyperedges. The hyperedges have the same depth of the successor tree.

Layered graph Layered graph is a hierarchical graph structure where the leaves are formed of the edges of the original graph. Each layer was created from the previous by an aggregation of its hyperedges.



(a) Layer 0 (b) Layer $\frac{n}{2}$ (c) Layer n

Figure 3.3: Illustration of layers in graph with 2,000 hyperedges.

3.1.1 Aggregation algorithm

As mentioned above the hyperedges of new layer are constructed by aggregating suitable hyperedges from the highest existing layer. Hyperedges should be as compact as possible. Whereas stretched out ones may cause an error during the pathfinding. Needless to say the hyperedges must be connected.

Here is described an algorithm that created one layer. The algorithm is run over again as long as there are still more layers that can be created or until the desired number of hyperedges at the top level is reached.

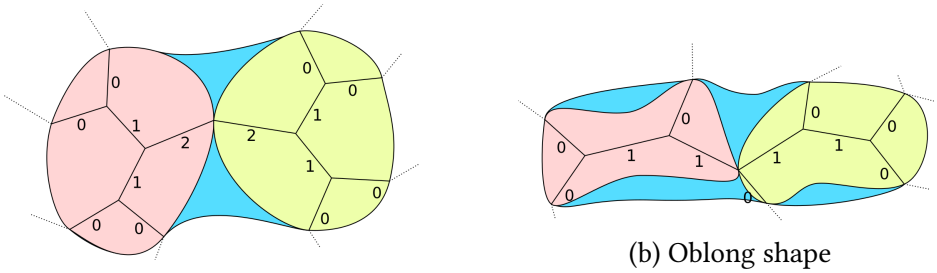
The algorithm works with a certain layer of the graph. The first time it is run, it is the original graph, respectively its hyperedge version. In one iteration, it finds the best pair of hyperedges to merge, merges it, and makes it a hyperedge at the next layer. The pairs of hyperedges are rated by a potential function.

Choosing a viable shape for the new hyperedge

The potential function used to construct new layers should yield higher values for hyperedges with round shape as opposed to oblong shape to ensure that the time required to traverse a hyperedge is approximately the same in all directions.

The potential for pair of hyperedges is computed as follows. For a pair of hyperedges the underlying structure is considered, i.e. the base layer hyperedges that are successors of the hyperedges of this pair. These hyperedges are considered to be a graph. On this graph a modified Dijkstra's algorithm is executed. In the initial phase all border hyperedges are pushed to the priority queue. Then the process is the same as usual. The computation stops when there are no hyperedges to be expanded. The potential of the newly considered hyperedge—the

pair of hyperedges—is the highest value assigned to an hyperedge. It is called *result* in equation (3.1).



(a) Round shape has higher label.

(b) Oblong shape

Figure 3.4: Dijkstra's algorithm from the outside in.

Using the *result* as the potential might be a disadvantage for hyperedges with small weights. Setting

$$\text{potential} = \frac{\text{result}}{\text{hyperedgeWeight}^k} \tag{3.1}$$

and tuning the parameter $k \in [1, \infty)$ makes it possible to control the order of hyperedge merging.

Creating new layers

The aggregation algorithm processes one layer at a time. Each hyperedge of the processed layer starts as a single component. In each iteration of the aggregation algorithm the pair with the highest potential is selected. A new hyperedge, whose children are hyperedges of the pair, is created in the next layer. This continues until there are no suitable pairs of hyperedges left. For each hyperedge h, that remained as single component, a new parent hyperedge is created. Its only child is h.

Algorithm 4 Aggregation algorithm

```
procedure createNewLayer(currentLayer)
    for h ∈ hyperedges of currentLayer do
        c_i ← h                    ▷ c_i is a new component whose only element is h
    end for
    while number of components > c || there are suitable pairs do
                                   ▷ c is the desired number of hyperedges on the top layer
        c_1, c_2 ← the pair with the best potential
        c_3 ← merge(c_1, c_2)
        newLayer.add(c_3)          ▷ new component c – parent of c_1 and c_2
    end while
    for h ∈ single element components do
        newLayer.add(h)
    end for
end procedure
```

Minimum Spanning Tree approach

A variant of Sollin's or Borůvka's algorithm for minimum spanning tree was also considered. Starting with components each of size 1 corresponding $1 : 1$ to hyperedges. Each component was merged with neighboring component with the smallest distance between them. The computation is very fast but the whole structure was out of balance and the components were not of a nice shape.

3.1.2 Handling dynamic changes

Dynamic networks can change their structures. Edges of the graph can be deleted and new edges can be added. This can be used to simulate various common real-world phenomena, such as road closures that are long term, and temporary road closures due to an accident that happen dynamically. In a single layered graph this can be handled with no issues. Both operations can be divided into two categories. One that affects the parent hyperedge—its connectivity or its neighborhood–and one that does not. Neighbors are either given or taken a link to the new/old neighbor. On graphs with multiple layers these operations are more complex.

Adding edges

This subsection describes and illustrates situations that can occur when adding a new edge to a graph. A more precise description of the algorithm is described in algorithm 5. The later described situations are illustrated in figures 3.5 and 3.6.

- Edge addition that connects elements in a single component does not affect parent's neigbors. Parent hyperedge is still connected, only its weight changes.
- Edge addition that connects two or more components affects parent's neighbors. Parent hyperedge must be informed about its new neighbors. The change is propagated towards all ancestors whose neighbors have changed.

Removing edges

This subsection describes and illustrates situations that can occur when removing an edge from a graph. A more precise description of the algorithm is described in algorithm 6. The later described situations are illustrated in figures 3.7 and 3.8.

- The removal of an edge does not change the connectivity of the parent hyperedge. The connections of parent hyperedge to its neighbors are still present. The structure of upper layers does not have to be changed.
- The removal of an edge splits the parent hyperedge into two parts or changes its neighbors. In figure 3.8 only a single hyperedge is split into two parts. The neighbors of these parts are subsets of the neighbors of the former hyperedge. The change is propagated towards all ancestors whose connectivity or neighbors have changed.

Algorithm 5 Edge addition

```
procedure ADDEDGE(edgeID)
    for n ∈ neighbors(edgeID) do
        neighbors(n).insert(edgeID)
    end for
    p ← parent of the most fitting hyperedge
    parent(edgeID) ← p
    children(p).insert(edgeID)
    while neighbors(p) changed do
        for n ∈ neigbours(p) do
            updateConnections(p, n)
        end for
        p ← parent(p)
    end while
end procedure
```

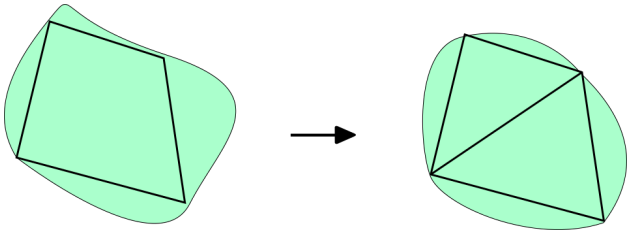


Figure 3.5: Edge addition not affecting parent's neighbors

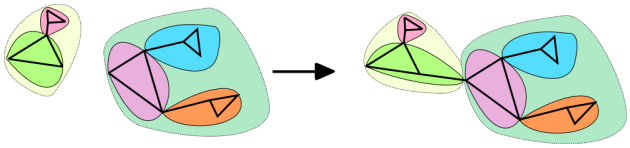


Figure 3.6: Edge addition affecting parent's neighbors

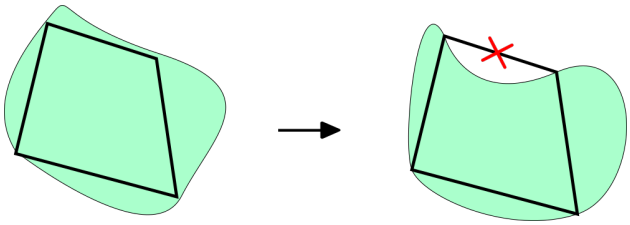


Figure 3.7: Edge removal not affecting parent's neighbors

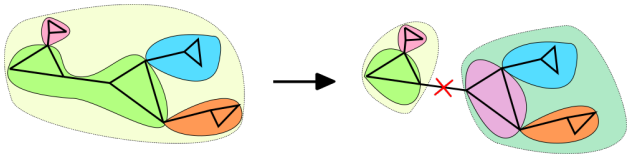


Figure 3.8: Edge removal that splits the parent hyperedge into two parts

Algorithm 6 Edge removal

```
procedure REMOVEEDGE(edgeID)
    parent(edgeID).remove(edgeID)
    currentEdge ← edgeID
    while parent(currentEdge) is not connected do
        split(parent(currentEdge))
        for n ∈ neigbours(parent(currentEdge)) do
            updateConnections(parent(currentEdge), n)
        end for
        currentEdge ← parent(currentEdge)
    end while
end procedure
```

Balancing the dynamic structure

The operations of edge addition and removal are fast, but not optimized in terms of balance. Since the structure is not self-balancing, a large number of changes causes an imbalance of the layer hierarchy.

In a realistic situation, this should not matter, because the roads change relatively slowly and the imbalance arises gradually. In the worst case, the whole structure can be rebuilt. If the structure were to be dynamic, it could be check after each addition if there is any suitable neighboring hyperedge to merge with.

3.2 Pathfinding in the layered network

Vehicles need exact driving instructions for the few following junctions but for junctions close to the destination, less precise information is sufficient.

Layer where the search is executed is determined by the distance travelled so far. Exact information must be used near the vehicle's current position, hence the search must be executed in the base layer. However, the computation on this layer is time consuming. The further the search gets from the current position. the less precise it needs to be. This is called a precision trade-off and it will be discussed in the next subsection.

Definition 6 (Jump-based precision limit)**.** *For algorithm calculation constants j representing the number of hyperedges on current path from the source(jumps), l representation the minimum number of hyperedges any path must use on each layer, and c representing the additional number of jumps performed on the base*

layer, the jump-based precision limit PRECISIONLIMIT *is defined by the equation*

$$\text{PRECISIONLIMIT} = \frac{j}{l} + c \tag{3.2}$$

3.2.1 Dijkstra's algorithm with precision limit (DwPL)

We propose a modification of Dijkstra's algorithm that can be used on layered graph network. The precision limit is used to determine whether a transition to an upper layer is possible or not. A priority queue is used to determine the order of hyperedge expansion. Each hyperedge is assigned the number of steps it takes to get there. The transition happens only when the precision limit is met. Another criterion for transition is that the parent of the scanned hyperedge is distinct from the parent of the expanded hyperedge. This ensures that already expanded hyperedges are not counted more than once. The DwPL pseudocode can be seen in algorithm 7 and the function testing the precision limit can be seen in algorithm 8.

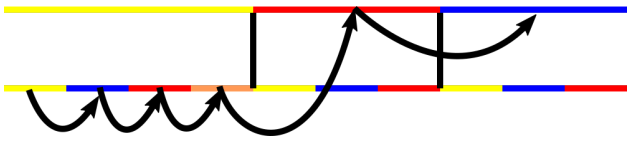


Figure 3.9: Direct jump to the parent hyperedge

Theorem 1 (Complexity of Dijkstra's algorithm with precision limit)**.** *On a graph with n edges that describes a realistic road network, Dijkstra's algorithm with precision limit explores at most* $\mathcal{O}(\log n)$ *hyperedges.*

Proof. On each layer l, DwPL extends all paths from the starting hyperedge by s steps, where s is bounded by:

PRECISIONLIMIT $\leq s \leq$ PRECISIONLIMIT + PARENTSTEPS.

and PRECISIONLIMIT is $\in \mathcal{O}(1)$ and PARENTSTEPS corresponds to the number of steps it takes to reach an hyperedge whose parent is distinct from the parent of the expanded hyperedge.

There must be at least PRECISIONLIMIT steps on each path on a single layer. Then the number of additional steps depends on when a hyperedge with a distinct parent gets found. Hence at most PARENTSTEPS more steps have to be done. Since each parent has either one or two children (algorithm 4), PARENTSTEPS $\in \mathcal{O}(1)$.

The number of paths depends on the number of neighbors a hyperedge has,

more specifically on the degree of appropriate vertices in the original—vertex-edge based—graph. For road networks this number is small [1].

Hence the number of paths from each starting point is $\mathcal{O}(1)^{\mathcal{O}(1)}$ and there are $\mathcal{O}(1)$ steps per path. On each layer $\mathcal{O}(1)$ hyperedges are opened. By the nature of algorithm 4 there are $\mathcal{O}(\log n)$ layers of the graph. Since $\mathcal{O}(1)$ hyperedges are opened on $\mathcal{O}(\log n)$ layers the total number of hyperedges opened is $\mathcal{O}(\log n)$.

□

The design of DwPL is not suitable for offline navigation. The vehicles are located on the base layer only. Routes are made out of hyperedges on upper layers too but the vehicles are not able to get there. DwPL is supposed to be used as a online routing navigation system. New search must be performed at each junction where the instructions contain hyperedges that are not on the base layer. Search in the aggregated structure is fast, hence the time consumption needed to perform search at such junction is minor. Moreover, the route can be more specific on the base layer by a fixed number of steps (c in definition 6).

3.3 Network updates

The proposed navigation system is able to work with the current hyperedge occupancy. In addition, reservations are added to help determine hyperedge occupancy in the future. Predictions are therefore not based on historical data but rather on serious reservations made by the vehicles. The vehicles create reservations on hyperedges they plan to use. The time of expected entrance is registered. The prediction of the time expected to traverse a road section can be more specific.

Aggregation of reservations

Reservations in a layered graph made by vehicles are registered in relevant hyperedges. Those do not have to be on the base layer. Since vehicles receive driving instructions that include hyperedges from upper layers, the reservations are registered there. The information about a new reservation is propagated to all ancestors of the reserved hyperedge. Moreover each vehicle can have at most one reservation for each hyperedge. On a graph with H hyperedges in the base layer, there are at most $\mathcal{O}(\log H)$ layers above a certain hyperedge, hence $\mathcal{O}(\log H)$ records must be done, one for each ancestor.

[1] Edges in graph used for the experiment have 3 neighbors on average and the maximum number of neighbors is 12(8 on one and 4 on the other end).

Algorithm 7 Dijkstra's algorithm with precision limit

```
procedure DwPL(source,target)
    for h ∈ hyperedges do
        state(h) ← not found
        distance(h) ← ∞
        predecessor(h) ← undefined
    end for

    distance(h_0) ← 0
    PQ.push(source, 0)          ▷ PQ queue of pairs (open hyperedge, #jumps)

    while PQ is not empty do
        h, currJumps ← PQ.pop()
        if h = target or h = ancestor(target) then
            return the path from source to h
        end if
        for n ∈ neighbors(h) do
            if transitionAllowed(h, n, currJumps) then
                n ← parent(n)
            end if
            if distance(n) > distance(h) + weight(h, n) then
                distance(n) ← distance(h) + weight(h, n)
                predecessor(n) ← h
                state(n) ← open
                PQ.push(n, curr_jumps + 1)
            end if
            state(h) ← closed
        end for
    end while
end procedure
```

Algorithm 8 Function that determines if a transition to an upper layer for certain hyperedge and number of jumps can be done. `precisionLimit` is the number of jumps needed to be done on each layer.

```
function transitionAllowed(hyperedge, neighbor, jumps)
    l ← layer(hyperedge)
    ph ← parent(hyperedge)
    pn ← parent(neighbor)
    if jumps > precisionLimit * l & ph ≠ pn then
        return true
    end if
    return false
end function
```

Use of aggregated reservations

The routing system computes optimal routes with respect to the current and also the expected state of the network. As already mentioned, the expected state is not predicted from historical data but rather from reservations made by vehicles. Hyperedge occupancy depends on the number of vehicles planning to be there at a certain point of time.

The expected number of vehicles present on a hyperedge at a time segment is determined by the number of reservations on this hyperedge and all of its ancestors. That is computed as the number of vehicles with direct reservations on the particular hyperedge in addition with occupation contributions from hyperedge above.

Chapter 4

Benchmarks and results

The pathfinding and routing system was tested on a dataset by Li et al. [9][1] that has been customized to support the hyperedge based approach.

The benchmarking program was implemented in C++ and uses SFML[2] library for simple dynamic graphical output. It works as a discrete simulation that navigates and reroutes vehicles in a layered graph. A python script was used to transform the vertex-edge based graph into an edge based graph.

Experiments have been performed on one core of a single Intel Core i5-7200U processor clocked at 2.50 GHz with 8 GB main memory running Ubuntu 20.04 (kernel 5.4.0). The program was compiled by the g++ compiler 9.3.0 using optimization level 3. More details can be seen in appendix A.

4.1 Preprocessing

First, the time needed to construct the hyperedge hierarchy is measured. There are $\mathcal{O}(\log n)$ layers of the graph, hence the preprocessed data takes $\mathcal{O}(n \log n)$ space. (n is the number of edges of the original graph.) We do not aim to prove this bound rigorously, instead we have observed that realistic graphs did not produce any pathological cases and the hyperedge hierarchy contractions were sufficiently regular and balanced. More rigorous evidence (and precise conditions or required structure modifications) are a subject of future work. Table 4.1 shows the empirical observations of the space usage of the multi-layered structure.

[1] Data was downloaded from `https://www.cs.utah.edu/~lifeifei/SpatialDataset.htm`, more specifically `https://www.cs.utah.edu/~lifeifei/research/tpq/SF.cnode` and `https://www.cs.utah.edu/~lifeifei/research/tpq/SF.cedge`.

[2] `https://www.sfml-dev.org/`

Graph Size [#edges]	Preprocessing Time [s]	Used Space [kB]	
		Original graph	DwPL w/ hyperedges
2k	1	58	153
30k	44	790	2060
220k	625	5359	13763

Table 4.1: Listing of preprocessing times needed to construct hyperedge hierarchy for DwPL and comparison of the space used to store the original graph and graph with hyperedge hierarchy.

4.2 Pathfinding in layered network

Clustering parameter k

In subsection 3.1.1 (Aggregation algorithm) tuning parameter k was defined, see equation 3.1. Its purpose is to control the order of hyperedge merging. Since k is used as a power constant in the denominator, the higher the k, the more the smaller hyperedges are preferred, more precisely hyperedges with smaller weights. Figure 4.1 shows a comparison of the results from navigating vehicles in a graph structure created with different values of the parameter k, here $k \in \{1.25, 1.5, 2\}$. The results show how many times longer the routes computed on the layered version of the graph were compared to the ideal time spend driving. The following experiments are performed with $k = 2$.

Precision limit

Figure 4.2 shows the dependency of route times on the precision limit. As expected the precision limit corresponds to the route precision. Meaning more time is spent in each layer so the routes are more precise.

However, the precision limit is also related to computational overhead. Table 4.2 shows the dependency of time, number of scanned objects and number of reroutes on the precision limit. The simulation was performed on a graph with 220 thousand edges with 100 vehicles having different endpoints.

Comparison with A* and Dijkstra's algorithm

In this subsection the performances of Dijkstra, A* and Dijkstra wPL are compared. There are 100 vehicles driving in the simulation. Each of them has different endpoints generated. A version of the map of San Francisco with 220,000 edges from the aforementioned dataset is used. The precision limit for DwPL is set to 4.

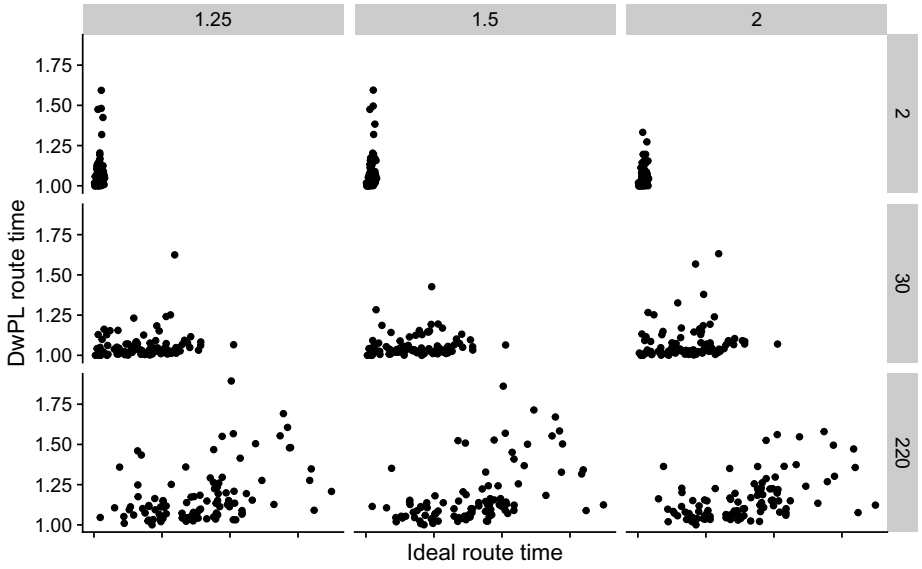


Figure 4.1: Comparison of route times computed by DwPL compared to the ideal route times. Column labels correspond to k, row labels correspond to graph size in thousands of edges.

Precision limit [#jumps]	Scanned objects [M]	Simulation time [s]	Reroutes [K]
2	94	72	7.8
3	112	90	6.0
4	123	103	4.7
5	134	121	4.0
10	159	158	2.3

Table 4.2: The dependence of time, number of scanned objects and number of reroutes on the precision limit in simulation on a graph with 220K and 100 vehicles.

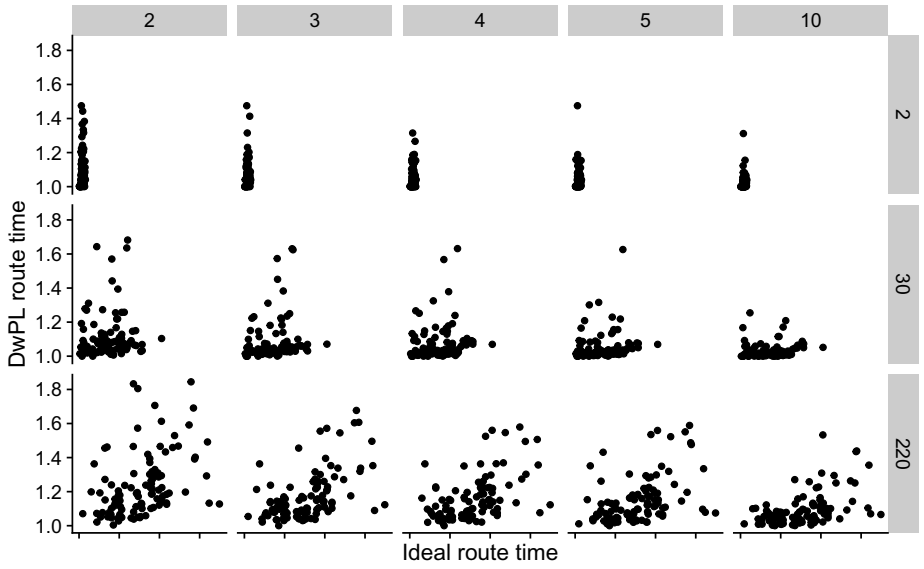


Figure 4.2: Comparison of route times by DwPL compared to the ideal route times. Column labels correspond to the precision limit(#jumps), row labels correspond to graph size in thousands of edges.

Dijkstra and A* are used in the same way as DwPL, i.e. as an online routing system. Experiments were performed for $\alpha = 5$ and $\alpha = 10$, which means that both will recompute the route every 5 or 10 hyperedges. Dijkstra with precision limit is forced to do so because it has no more instructions on the base layer, Dijkstra and A* are forced to do it artificially. The results are shown in table 4.3.

Even though the vehicles are rerouted 2.5 times more often compared to A* with $\alpha = 10$ the number of scanned objects is only 43% of what is scanned by A*. And the computation time is 15% of the A* value at the same α. The routes are 16% longer than ideal on average.

Moreover, if DwPL is compared to A* for $\alpha = 5$ (which is an even more

Algorithm	Scanned objects [M]	Simulation time [s]	Precision limit / α
Dijkstra	2314	4070	5
Dijkstra	1176	2110	10
A*	675	1563	5
A*	388	879	10
Dijkstra wPL	**169**	**131**	4

Table 4.3: Comparison of the number of scanned objects and simulation time between Dijkstra's algorithm, A* and Dijkstra's algorithm with precision limit.

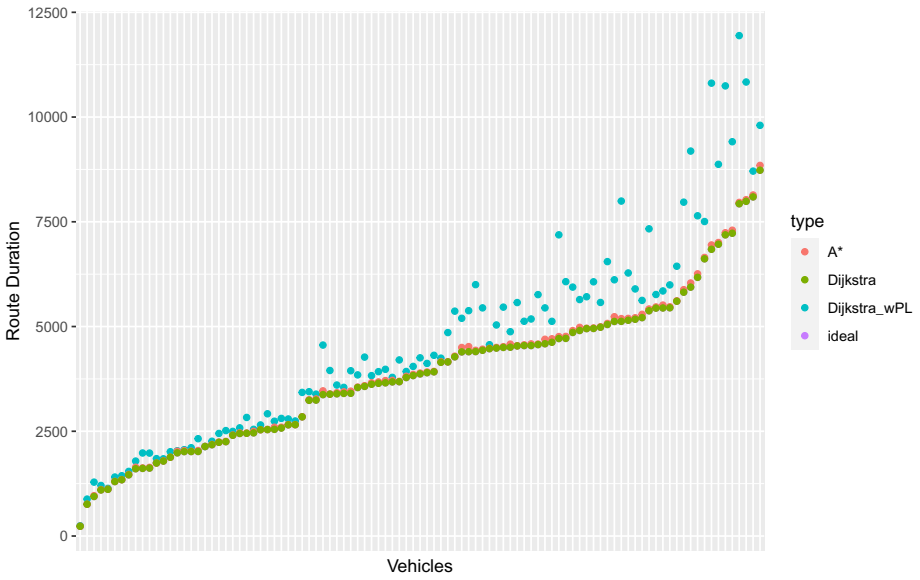


Figure 4.3: Comparison of route times of path-finding algorithms.

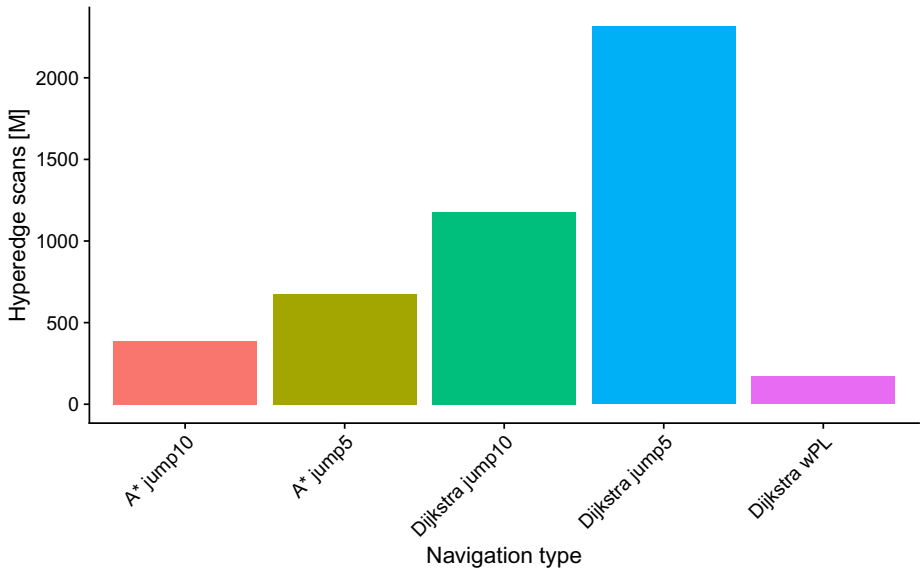


Figure 4.4: Comparison of scans of path-finding algorithms.

appropriate comparison), the number of scanned objects is approximately 25% of what is scanned by A*, and the simulation time is less than 9% of the A*'s.

4.3 Complexity of path updates

Regarding graph updates, there are two situations that need to be distinguished: congestions and hyperedge removals. Vehicles that are to use the affected hyperedges are informed by the system. The information is not propagated into upper layers because its impact there is considered to be negligible. *To simplify the implementation, it is assumed that each congested hyperedge causes traffic delays only for vehicles that traverse that precise hyperedge. In the real world, adjacent hyperedges would likely be congested as well.*

In case of hyperedge removal, multiple layers may be affected. The parent hyperedge can be divided into two parts. As the change is propagated into upper layers and hyperedges are split, the vehicles with direct reservations on affected hyperedges are informed. If a hyperedge remains connected even after the process, the vehicles with direct reservation in it do not need to be informed. Their destinations are still reachable via this hyperedge.

In a graph with H hyperedges in the base layer, a network update involves only directly affected vehicles. There are at most $\mathcal{O}(|\text{vehicles}|)$ of them. Each of these vehicles has only one direct reservation in one of the $\mathcal{O}(\log H)$ affected hyperedges. The change causes at most $\mathcal{O}(|\text{vehicles}|)$ reroutes. Since each of the $\mathcal{O}(\log H)$ hyperedges has $\mathcal{O}(1)$ neighbors, the network transformation takes $\mathcal{O}(\log H)$.

4.4 Routing efficiency

In this section the reservation system is tested. The goal is to see if vehicles choose the best route based on the real time state of the network.

In real world networks the road capacity depends on the number of lanes, shape, etc. *To simplify the implementation the maximum capacity is the same for all edges of the network.* The simulation model has a tunable parameter corresponding to the capacity. It is set very low compared to the real value[3] to enhance the edge occupancy.

In this experiment, 100 vehicles with the same start and end point are running on San Francisco graph with 2k edges. They set off moments after each other to make sure that the edges fill up and the vehicles are forced to use different

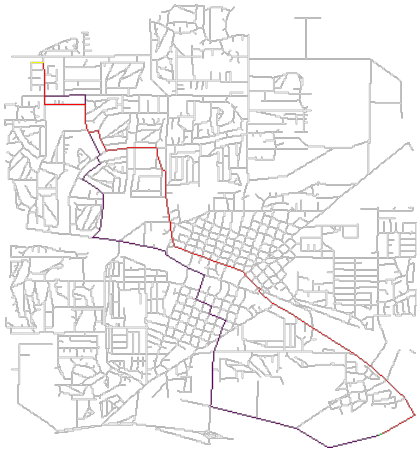
[3]In this case, it was set to 5 as opposed to 30. The parameter can be found in table A.2 as 'vehs_per_unit' in appendix A.

Algorithm	Precision limit / α	Avg route duration		Computation time [ms]	
		Without	With	Without	With
Ideal (no traffic)		535.829		—	
A*	5	657.890	643.812	3752	5826
Dijkstra wPL	4	714.543	660.827	965	2690

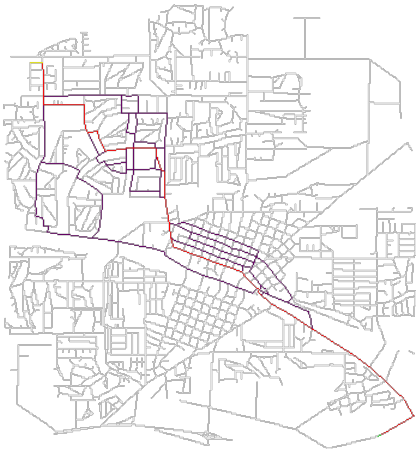
Table 4.4: Comparison of ideal route time and average route times of A* and DwPL with and without real-time network state information.

edges. The endpoints are chosen to be on opposite sides of the graph so that the differences in the paths used are more visible. The goal of this experiment is to show that vehicles reach their destination faster if they work with the predicted network state based on reservations. The results can be seen in table 4.4 and the routes vehicles used are illustrated in figure 4.5.

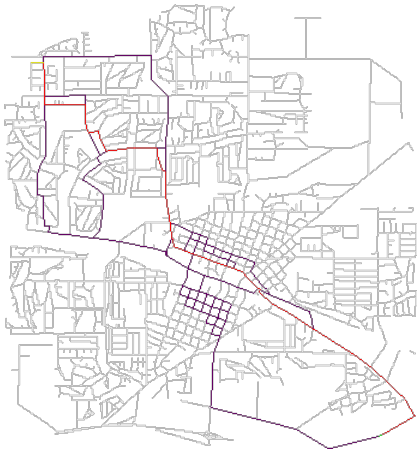
Figure 4.5 shows routes the vehicles used in this experiment. Part 4.5a shows the ideal route computed by A* (red) and the ideal route computed by DwPL (violet). This also demonstrates one of the cases where DwPL chooses a slightly suboptimal path. Parts 4.5b and 4.5c show that vehicles are spread across the road network when using the real-time information. The simulation time is longer in the second case for both A* and DwPL because of the way occupancy is computed. With each query there is a simulation of the occupation of the edges from the very beginning of the simulation. The computation time could be reduced by possibly using a cache to store snapshots of the network state and simulate future states using these snapshots.



(a) Ideal routes by A* and DwPL



(b) Routes used by A*



(c) Routes used by DwPL

Figure 4.5: Comparison of route utilization with and without real time information. Yellow - source, green - target, red - ideal route, violet - in part 4.5a the ideal route by DwPL and in parts 4.5b and 4.5c routes used by the vehicles.

Conclusion

The goal of this thesis was to propose an algorithm which could be used as part of a routing system. The algorithm should be usable on a structure that represents a road network, even including its utilization, and the system should disperse vehicles across the network to make the best use of its throughput. The algorithm should use the knowledge of routing strategies and pathfinding speed-up techniques and explore as few objects as possible in order to be fast.

We have proposed an algorithm achieves this by using a layered graph network that makes it possible to trade off the precision of pathfinding computation for speed gains from the approximation. This network is created by aggregating edges into successively larger hyperedges, which create a smaller, quickly searchable representation of the same graph.

To navigate vehicles in such a structure, we proposed a modified version of Dijkstra's algorithm—Dijkstra's algorithm with precision limit (DwPL)—which approximates the best path, with precision–performance trade-off determined by tunable constants. In particular, it is faster compared to Dijkstra's or A* algorithm but the computed paths are sub-optimal. The computation process is also affected by the precision limit. The higher it is, the better the calculated paths are, the longer the calculation time. As opposed to A*, DwPL computes paths effectively even without heuristics.

The number of scanned objects by DwPL in the previously mentioned simulation on a graph with 220k edges is reduced to 25% of what A* scans and 7% of what Dijkstra scans. The simulation time is reduced to 8% of what A* takes and 3% of what Dijkstra takes. In other words, DwPL scans 4× and 14× less objects than A* and Dijkstra. The simulation time is more than 12× and 33× faster than in case of A* and Dijkstra. The routes computed by the said algorithm are only 16% longer on average than the optimal ones. Moreover, the information a vehicle must upload to the system to make a reservation is reduced to size logarithmic in the length of the route, which in a realistic scenario means significant bandwidth savings.

Future work

In the future, it might be interesting to add heuristics as used by A* (section 1.2.1) or to use the reach (section 1.2.4) to prune some of the hyperedges during the search.

The layered hyperedge structure works empirically well, but it would be great to design an algorithm that keeps the structure always in a provably good shape. We expect that such structure might be similar to the tree-balancing algorithms.

Bibliography

[1] Ittai Abraham et al. "Highway dimension, shortest paths, and provably efficient algorithms". In: *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics. 2010, pp. 782–793.

[2] Daniel Delling and Dorothea Wagner. "Landmark-based routing in dynamic graphs". In: *International Workshop on Experimental and Efficient Algorithms*. Springer. 2007, pp. 52–65.

[3] Edsger W Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[4] Fabian Fuchs. "On Preprocessing the ALT-Algorithm". In: *Student thesis, Faculty of Computer Science, Institut for Theoretical Informatics (ITI), Karlsruhe Institute of Technology (KIT)* (2010).

[5] Robert Geisberger et al. "Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks". In: *Experimental Algorithms*. Ed. by Catherine C. McGeoch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 319–333. ISBN: 978-3-540-68552-4.

[6] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. "Reach for A*: Efficient Point-to-Point Shortest Path Algorithms". In: *2006 Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, Jan. 2006. DOI: 10.1137/1.9781611972863.13. URL: https://doi.org/10.1137/1.9781611972863.13.

[7] Peter Hart, Nils Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/tssc.1968.300136. URL: https://doi.org/10.1109/tssc.1968.300136.

[8] Norulhidayah Isa, Azlinah Mohamed, and Marina Yusoff. "Implementation of dynamic traffic routing for traffic congestion: A review". In: *International Conference on Soft Computing in Data Science*. Springer. 2015, pp. 174–186.

[9] Feifei Li et al. "On trip planning queries in spatial databases". In: *International symposium on spatial and temporal databases*. Springer. 2005, pp. 273–290.

[10] Eiichi Taniguchi and Hiroshi Shimamoto. "Intelligent transportation system based dynamic vehicle routing and scheduling with variable travel times". In: *Transportation Research Part C: Emerging Technologies* 12.3-4 (2004), pp. 235–250.

[11] Paolo Toth and Daniele Vigo. *The vehicle routing problem*. SIAM, 2002.

[12] Shen Wang, Soufiene Djahel, and Jennifer McManis. "A hybrid vehicular re-routing strategy with dynamic time constraints for road traffic congestion avoidance". In: (2013).

[13] Eric W. Weisstein. *Weighted Graph*. URL: `http://mathworld.wolfram.com/WeightedGraph.html`.

Appendix A

Installation and use of the attached program

A.1 How to build

The attached program uses the SFML[1] library for graphical purposes. On Debian-based Linux systems (such as Ubuntu), it may be installed with APT:

```
apt-get install libsfml-dev
```

The program is build with `make` command and uses g++ compiler with c++17 standard.

To unpack and compile the program do the following steps:

```
unzip dwpl.zip
cd dwpl/src
make
```

A.2 Preparing data for benchmark

After the program is compiled, it can be run. The program works with an edge-based graph, i.e. edges that are in a neighborhood relation with other edges. There are a few such graphs in the `data` folder with names ending with the suffix `.edgs`.

However, the user is not limited to this data. Any dataset following the schema mentioned bellow, and also described in the file `data/templates/template.edgs`.

[1] `https://www.sfml-dev.org/`

```
[edgeId] [x1] [y1] [x2] [y2] [neighborId ...]
```

The data in the mentioned folder are in the said schema. Its original format, as downloaded from [9][2], can be converted to the edge-based format by a simple python script located in `data/processing/shave_data.py`, more specifically its function `to_edge_based()` that takes two input files, one with vertices and one with edges.

There are also files ending with `.layers` in the `data` folder. These are the pre-processed layers for the above layered graph(3.1) with corresponding name. Their primary use is to get the program up and running quickly. If the `*.layers` files for the corresponding graphs did not exist, the layers for the graph would have to be created. For a graph with 220k edges it takes about 2 minutes, so it is a nice way to speed it up.

If any `.edgs` file does get edited, the corresponding `.layers` file needs to be deleted to avoid using invalid information. A new `.layers` file gets created and saved for the following use during the start up.

A.2.1 Note on the use of graphic functions

For the graphical output to work correctly, a folder with name `img` must be present. The programs executable accesses the folder with path `../img/`. If the user compiles the program, the resulting structure should look like so:

```
/data
    /processing
        ...
    xxx.edgs
    yyy.edgs
    xxx.layers
    yyy.layers
/img
/src
    *.hpp
    *.cpp
    main
README.md
```

[2] `https://www.cs.utah.edu/~lifeifei/research/tpq/SF.cnode` and `https://www.cs.utah.edu/~lifeifei/research/tpq/SF.cedge`.

A.3 Starting the program

The program can be run from the command line in two ways:

```
./main
```

or

```
./main -f input.edgs
```

In the second way, the user chooses the graph on which the program will run. If the user runs program in the first way, the `data/sf2k.edgs` file is used by default.

The program operates in three phases: Graph loading, Pre-simulation phase, and Post-simulation phase.

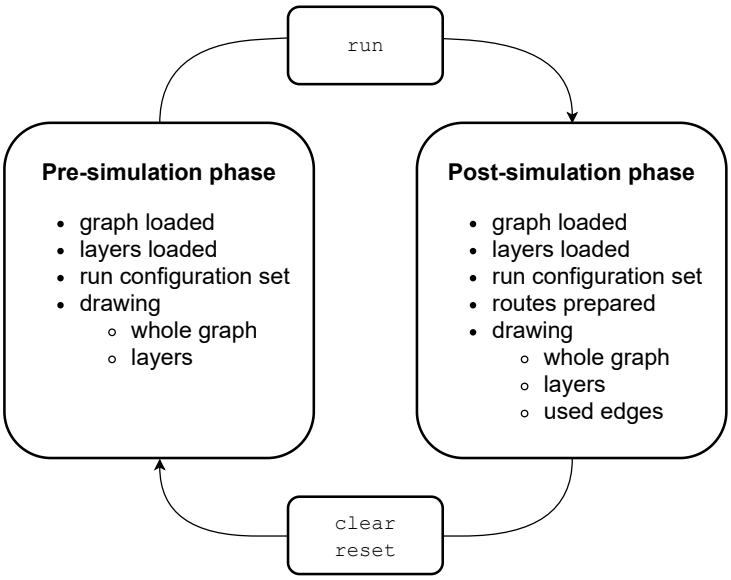


Figure A.1: Program workflow diagram.

A.3.1 Graph loading

The file with input graph must either be given as a parameter on the command line, or `data/sf2k.edgs` is chosen by default. Or the filename inside `main.cpp(38)` can be changed and the program must be recompiled.

The graph (from a `.edgs` file) is loaded into memory and is the base layer of the graph structure used by DwPL. Next, the layers of the graph must be created. If the layers already existed and were stored in `data/.layers` file, they would be used to save some time. If the layers for the graph have not been created yet, they need to be created. After some time it moves to the Pre-simulation phase.

A.3.2 Pre-simulation phase

In this phase the graph and all of its layers are loaded in the memory. The simulation can be run with the predefined configuration. The configuration can be changed by commands listed in section A.5.1 (Commands).

The graph with its layers can also be drawn by the `draw` command. Its paremeters are mentioned in the previously mentioned section.

The command `run` moves the program in the Post-simulation phase. At this moment the routes on which the vehicles will travel are created(depending on the configuration), the vehicles are created and sent on their way.

A.3.3 Post-simulation phase

At this phase all vehicles have completed their routes, whether successfully or unsuccessfully. Information about their routes are printed to the console. Records of their routes still remain in memory, more specifically the routes they have used.

With this, the `routes` parameter for the `draw` command makes the program draw the edges the vehicles used.

The previous phase can be accessed by commands `clear` and `reset`. The `clear` command clears the data about vehicles and routes. It also deletes the created routes (pairs of endpoints), the seed of the random generator remains the same. So the command `run` will perform a new simulation where the configuration remains the same and the vehicles will follow different routes. The `reset` command also clears the data about vehicles and routes. The random generator is given the same seed as last time. The `run` command will run the same simulation as last time, thus having the same result.

A.4 Sample use of the program

Suppose the user wants to run a simulation on the San Francisco graph with 2k edges, with 50 vehicles that drive between 5 pairs of endpoints and are navigated by DwPL. The user should do the following steps.

1. Run the program with the command line parameter `-f` to specify the input file. Make sure you are in the `bin` folder where `main` is located.

```
./main -f ../data/sf2k.edgs
```

2. To see how the fifth layer of the graph looks like, command `draw` with parameter specifying the layer must be run.

```
draw 4
```

3. The current configuration can be listed using command `conf`. The command outputs the navigation type, the number of vehicles, and the number of routes that will be used when the simulation is run.

```
conf
```

4. The default number of vehicles is 30. To set it to 50, command `v` must be run.

```
v 50
```

5. The default number of routes is 0 which means that each vehicle will get a random route. To set the number of routes to 5, command `r` must be run.

```
r 5
```

6. The user can check whether the configuration is to his expectations by running command `conf` again. Now the program should be ready to run the simulation. To do so, command `run` must be run.

```
run
```

7. The program prints stats when the simulation is finished. Now, if the user wants to see which routes were used, command `draw routes` must be run. The path to the rendered file is printed to the console.

```
draw routes
```

8. The simulation can be reset with the command `reset` which would make the program run the same simulation again.

```
reset
```

9. To run a new simulation with the same configuration but different routes, the command `clear` should be run.

```
clear
```

10. To exit the program, type in `ctrl + D`.

To run the experiment referenced in table 4.4, the user can define macro `SF2K_EFFICIENCY` by appending `-DSF2K_EFFICIENCY` to `CXXFLAGS` in `src/Makefile`. This macro sets the endpoints to the appropriate edges. The vehicles use only those two endpoints. To fully replicate the experiment, the number of vehicles must be set to 100, and the edge capacity (in table A.2 as 'vehs_per_unit') must be changed to 5. The changed source code needs to be recompiled. The easiest way to do it is to run:

```
make clean
make
```

A.5 Commands & Parameters

A.5.1 Commands

The program waits for commands in the console or executes the commands. Table A.1 contains a list of all commands of this program, their description, and possibly their parameters and default values.

A.5.2 Parameters

The program also works with various configuration parameters, such as precision limit or clustering parameter k. Table A.2 lists the parameter names, their description, and where to find them.

Name	Description	Parameters	Implicit
`run`	Runs the configured simulation.		
`v`	Sets the number of vehicles to new value. Expects one non-negative integer.	`uint`	30
`r`	Sets the number of routes. Vehicles will travel between the given number of pairs of locations. If 0 then #routes == #vehicles	`uint`	0
`n`	Sets the type of navigation to one of following	`d` (Dijkstra)	
		`a` (A*)	
		`dwpl` (DwPL)	•
`conf`	Lists the specification of simulation.		
`clear`	Deletes existing routes, deletes congestions, keeps the random generator the same. Performing `run` starts new simulation, different routes will be created.		
`reset`	Deletes existing routes, deletes congestions, resets the random generator. Performing `run` should end with the same result as last time.		
`help`	Prints help.		
`draw`	Draws the base layer of the loaded graph.		
	Draws the layer with the specified number. Must be between 0 and (#layers-1)	`uint`	
	Draws the routes vehicles drove, if there are any.	"routes"	

Table A.1: List of commands and their descriptions.

Name	Description	Location
dwpl_jumps	The precision limit for DwPL(definition 6). Specifies the number of jumps DwPL has to make on path of each layer.	main.cpp(33)
hca_k	The clustering parameter k from def 3.1	Preprocessing.hpp (18)
realtime	Indicates whether vehicles find their routes with respect to reservations and current network state or not.	main.cpp(32)
a_star_jumps	Equivalent of dwpl_jumps for A* and Dijkstra. Specifies after how many hyperedges the algorithm tries to foind better routes.	Navigation.hpp(39)
specify_more	Specifies how many more hyperedges need to be explored on the base layer.	Navigation.hpp(40)
vehs_per_unit	Capacity of edges	Navigation.hpp(41)
routes	How many routes will the vehicles use in the simulation	command 'r'
vehicles	How many vehicles will be part of the simulation	command 'v'
nav_type	What navigation algorithm will be used in the simulation.	command 'n'

Table A.2: List of simulation parameters and their descriptions.