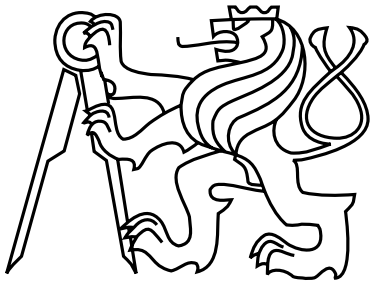
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science



Master's Thesis

Implementation of the Transit Node Routing Algorithm in the AgentPolis Simulation Framework

Bc. Michal Cvach

Supervisor: Ing. David Fiedler

Study Programme: Open Informatics

Field of Study: Artificial Intelligence

May 22, 2020

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Cvach Michal** Personal ID number: **457898**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence**

II. Master’s thesis details

Master’s thesis title in English:

Implementation of the Transit Node Routing Algorithm in the AgentPolis Simulation Framework

Master’s thesis title in Czech:

Implementace algoritmu Transit Node Routing v simulačním frameworku AgentPolis

Guidelines:

Transit node routing (TNR) with arc-flags is a very promising shortest path finding technique, possibly a hundred times faster than Contraction Hierarchies and a million times faster than the Dijkstra algorithm. Your task will be to implement this algorithm and integrate it with the AgentPolis framework.
1) Study articles about TNR and arc flags and understand the techniques.
2) Design and implement TNR with arc flags
3) Test the solution‘s correctness and performance.
4) Integrate your solution with AgentPolis and test it as a traveltime provider.

Bibliography / sources:

[1] H. Bast et al., “Route Planning in Transportation Networks,” in Algorithm Engineering, 2015.
[2] J. Arz, D. Luxen, and P. Sanders, “Transit Node Routing Reconsidered,” in Experimental Algorithms, Berlin, Heidelberg, 2013, pp. 55–66.
[3] C. Demetrescu, A. V. Goldberg, and D. S. Johnson, Eds., The shortest path problem: ninth DIMACS implementation challenge. Providence, R.I: American Mathematical Society, 2009.
[4] H. Bast, S. Funke, P. Sanders, and D. Schultes, “Fast Routing in Road Networks with Transit Nodes,” Science, vol. 316, no. 5824, pp. 566–566, Apr. 2007.
[5] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner, “Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra’s Algorithm,” in Experimental Algorithms, Berlin, Heidelberg, 2008, pp. 303–318.

Name and workplace of master’s thesis supervisor:

Ing. David Fiedler, Artificial Intelligence Center, FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **04.02.2020** Deadline for master's thesis submission: **22.05.2020**

Assignment valid until: **30.09.2021**

Ing. David Fiedler Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

. _____ _____

Date of assignment receipt | Student's signature

Aknowledgements

I would like to thank my supervisor Ing. David Fiedler for his patience and commitment in supervising this thesis. I would also like to thank my family, my girlfriend, and my friends for their support and understanding. Special thanks go to my fellow student Vu Huy Hoang who helped me greatly during my studies.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague on May 22, 2020 .....

Abstract

Computation of the shortest distance from one node to another in a graph is a common problem in computer science. Many applications contain a component for the shortest distance computation. This thesis focuses on the problem of shortest distance computation in directed weighted graphs. The thesis explores methods that precompute some auxiliary structures from the input graph to speed up the queries. This approach is suitable in cases where a large amount of queries needs to be answered and the graph does not change between the queries. The result of this thesis is a library for shortest distance computation using three such methods called Contraction Hierarchies, Transit Node Routing, and Transit Node Routing with Arc Flags. The library can preprocess directed weighted graphs to obtain the auxiliary structures needed by the three methods and then answer shortest distance queries quickly by utilizing those structures. The implementation is evaluated on graphs obtained from real road networks. The implementation of the Transit Node Routing with Arc Flags method based on Contraction Hierarchies present in the library answers random shortest distance queries more than 10 000 times faster than Dijkstra's Algorithm on the largest test graph, while only needing about 50 times the amount of memory.

Keywords: Shortest distance problem, Contraction Hierarchies, Transit Node Routing, Arc Flags, Road networks, Travel time computation

Abstrakt

Výpočet nejkratší vzdálenosti mezi dvěma vrcholy v grafu je častým problémem v informatice. Spousta aplikací obsahuje nějakou komponentu pro výpočet nejkratších vzdáleností. Tato práce se zaměřuje na problém výpočtu nejkratší vzdálenosti ve vážených orientovaných grafech. Práce prozkoumává metody, které ze vstupního grafu předpočítají nějaké pomocné struktury, které poté urychlují samotné dotazy. Tento přístup je vhodný v situacích, kdy je potřeba zodpovědět velké množství dotazů a graf se nemění mezi jednotlivými dotazy. Výsledkem této práce je knihovna pro výpočet nejkratších vzdáleností pomocí tří metod nazvaných Contraction Hierarchies, Transit Node Routing a Transit Node Routing s využitím Arc Flags. Výsledná knihovna dokáže pro libovolný vážený orientovaný graf předpočítat pomocné struktury potřebné pro dané tři metody a následně je využít pro rychlý výpočet nejkratších vzdáleností. Výkon implementace byl vyhodnocen na grafech získaných z reálných silničních sítí. Implementace metody Transit Node Routing založené na Contraction Hierarchies s využitím Arc Flags obsažená ve výsledné knihovně dokáže vypočíst nejkratší vzdálenost mezi dvěma náhodnými vrcholy grafu 10 000krát rychleji než Dijkstrův Algoritmus pro největší z testovaných grafů, a přitom potřebuje pouze 50krát více paměti.

Klíčová slova: Problém nejkratší vzdálenosti, Contraction Hierarchies, Transit Node Routing, Arc Flags, Silniční sítě, Výpočet doby jízdy

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Finding the shortest distance between two points is, without a doubt, one of the most iconic problems in computer science. An algorithm for computing the shortest distance between two points is often a part of a larger system and can greatly influence the performance of the whole system.

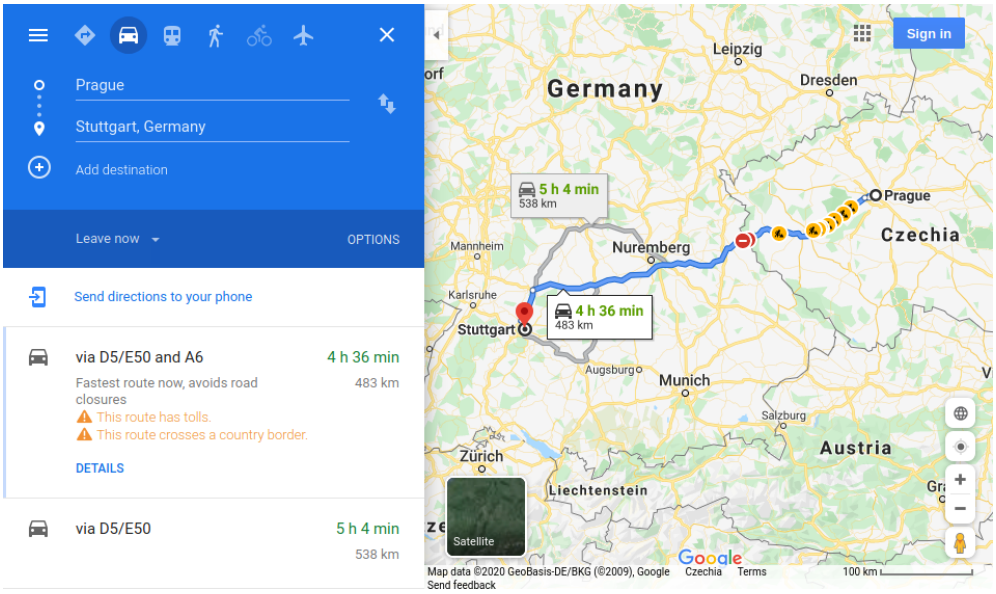


Figure 1.1: A typical application that needs a component that computes shortest distances - Google Maps.

One domain where the shortest distance problem is often tackled is transportation. Applications from the transportation domain that require shortest distance computation are for example GPS navigation applications (one example is given in figure 1.1 [1]), systems for logistics companies that may need to transport products efficiently, applications for simulating transport systems, and many more. One thing that is specific for those applications is the fact that real road networks have a specific structure. When we think of road networks

[1]The picture was taken from Google Maps on 05/08/2020. <https://www.google.com/maps/>

as graphs, they are almost planar graphs (only tunnels or overpasses cause edge crossings), and the average node degree in those graphs is fairly low (there are no intersections of 20 or more roads in real road networks). Due to this, it makes sense to develop specific algorithms for those use cases.

Many of the use cases need to answer large amounts of shortest distance queries in a short time. If the graph does not change between the queries, it makes sense to precompute some auxiliary structures that speed up the queries. This thesis focuses on some of the methods that work well for real road networks. For our implementation, we have decided to combine the *Transit Node Routing* framework with *Contraction Hierarchies* and *Arc Flags* to achieve significant speedup while keeping the memory overhead caused by the precomputed structures reasonable.

The result of this thesis is a library that can produce auxiliary structures for directed weighted graphs and then use those structures to quickly answer shortest distance queries. The library provides multiple precomputation methods that have different performance and memory requirements. While this library was developed mainly for usage in the *AgentPolis* framework into which it was integrated during this thesis, it can also be used in other projects. Since the number of use cases for which this library might be beneficial is large, we believe it might be useful for other researches and developers as well.

1.1 Goals

The main goal of this thesis is to implement *Transit Node Routing* based on *Contraction Hierarchies* extended by *Arc Flags*. This is a state-of-the-art approach for precomputing auxiliary structures to speed up shortest distance queries. At the time of writing this thesis, there does not exist any public implementation of this approach to our knowledge. The implementation should be usable from an existing *Java* simulation framework for modeling transportation systems called *AgentPolis* that is being developed by the Smart Urban Mobility cluster at the Czech Technical University in Prague.

We can, therefore, divide the thesis into four goals. The first goal is to study the relevant literature and understand the described methods. The second goal is to efficiently implement the algorithms. This implementation must be able to precompute the necessary auxiliary structures and then use them to speed up shortest distance queries. The third goal is to test the correctness of the implementation and evaluate its performance. We need to ensure that the distances computed by the implementation are correct and we also need to compare the performance of the implementation with some baseline algorithms. The fourth and final goal is to integrate the implementation into AgentPolis so that the framework can switch between existing methods for shortest distance computation and the new methods implemented during this thesis.

1.2 Thesis Outline

Chapter 2 formally defines the problem we are dealing with in the rest of this thesis and we establish a theoretical background. Chapter 3 describes the existing methods that can be

used to compute shortest distances in a road network. We describe the basic algorithms as well as the state-of-the-art approaches.

In chapter 4 we describe the methods used in our implementation in more detail. The methods we use are *Contraction Hierarchies*, *Transit Node Routing* and *Arc Flags*. We explain how to precompute the structures needed by those methods and how those structures can be utilized by the query algorithm to answer queries faster.

Chapter 5 then describes the implementation. We discuss the structure of our implementation and we describe the capabilities of our implementation. The integration into the AgentPolis framework is then described in chapter 6.

In chapter 7 we describe how we ensured the correctness of the implementation and how we evaluated the implementation to determine its benefit. We provide tests conducted using only our implementation independently as well as tests performed in the AgentPolis framework. We report the results and we draw some conclusions from them.

Chapter 2

Problem Description

Our goal is to quickly answer distance queries in a given graph. We use real road networks in our experiments, but the described principles, as well as the library, work for any correct directed weighted graph. We assume that the user will issue multiple queries for the same graph. Therefore it makes sense to precompute some structures that will help us answer subsequent queries faster. The more queries we need to answer the more sense it makes to spend some time precomputing, as the overhead required for the precomputation will be smaller than the cumulative time saved due to the faster query answer time.

In section 2.1 we describe all the terminology used in the rest of this work. In section 2.2 we then describe the problem we are dealing with itself.

2.1 Terminology

In this work, we will mostly be working with *directed weighted graphs*. A directed graph $G = (V, E)$ consists of a set of nodes V and a set of edges E. We denote the number of nodes by $n = |V|$ and the number of edges $m = |E|$. For directed graphs, each edge is an ordered pair of nodes, $\forall e \in E : e = (s, t), s \in V, t \in V$. For each edge $e = (s, t)$, we will call the node s *source node* of that edge, and the node t *target node* of that edge. In an undirected graph, each edge is an unordered pair of nodes. In such cases, we can transform the undirected graph $G' = (V', E')$ into a directed graph $G'' = (V'', E'')$ by replacing each edge $e' = \{s', t'\}$ by a pair of edges $e''_1 = (s', t')$, $e''_2 = (t', s')$.

For each node $v \in V$, we define an *outdegree* of v denoted $deg^+(v)$ as the number of edges for which v is the source node: $deg^+(v) = |\{e' : e' \in E, e' = (v, x), x \in V\}|$. Analogically, we define an *indegree* of v denoted $deg^-(v)$ as the number of edges for which v is the target node $deg^-(v) = |\{e' : e' \in E, e' = (x, v), x \in V\}|$. The *degree* of the node v denoted $deg(v)$ is then defined as the sum of its outdegree and indegree $deg(v) = deg^+(v) + deg^-(v)$. The *average degree* for a graph $G = (V, E)$ is the arithmetic mean of degrees of its nodes.

$$\frac{\sum_{v \in V} deg(v)}{|V|} \tag{2.1}$$

When talking about *directed weighted graphs*, we add a weight function $W : E \to \mathbb{N}_0$ assigning each edge a non-negative integer. Since we will be computing shortest distances

between points, we will be using those weights to compare possible candidate paths. When working with road networks, which is the case for this thesis, the most straightforward option is to use actual lengths of edges (roads) as weights. A better metric for road networks is the travel time, because usually when working with road networks we care more about the time than the actual distance (imagine for example a delivery service that needs to deliver packages as quickly as possible). Travel time can be estimated by dividing the length of the edge by the speed limit for the corresponding road. Those are, however, only examples of possible weight functions that work well on real road networks. For specific use cases or use cases from other domains than transportation, other weight functions might make more sense.

We want our graphs to be free of *parallel edges* (sometimes called multiple edges) and *loops*. Parallel edges mean that two or more edges have the same source and target node. Loops are edges for which the source and the target node are the same node. Graphs containing parallel edges or loops are usually called multigraphs. If our input graph contains loops, we can easily get rid of them by simply removing them as they will never occur in any shortest path (remember that we require our weights to be non-negative). Analogously, if our input graph contains parallel edges, we can only keep the edge with the smallest weight for each pair of source and target nodes. Graphs without parallel edges and loops are called *simple graphs*. Many algorithms expect simple graphs as input. When using algorithms that can handle multigraphs, parallel edges and loops only decrease performance, while not changing the results.

Real road networks can be transformed into weighted directed graphs rather easily. The junctions are transformed into nodes, and the roads connecting those junctions to edges. One useful property of real road networks is that the graphs obtained from them are almost *planar*. Planar graphs are graphs that can be drawn on a plane in such a way that their edges intersect only at their endpoints. There are structures in road networks that prevent them from being planar such as bridges and tunnels, but still, the average degree of road networks is fairly low. This property of road networks is beneficial for a lot of the methods described further. Most of those methods work on arbitrary graphs, but their performance is better on graphs with a lower average degree such as road networks.

Some methods that were developed with road networks in mind also need geographical positions of nodes to work. In such cases, each node also has a geographical position specified by its *latitude* and *longitude*. An obvious downside of such methods is the fact that they can not work with graphs that do not have spatial coordinates. Contraction Hierarchies and Transit Node Routing used in our implementation do not work with any geographical positions, and can, therefore, be used even for graphs where we do not have this information.

2.2 Problem Formulation

Assume we have a weighted directed graph $G = (V, E)$ and a pair of nodes s and g. Our problem is to find the shortest distance from s to g in G. We will call node s the *start node* and node g the *goal node*.

The shortest distance from s to g is the minimum weight of a path from s to g. If there exists a path p from s to g, then this path consists of k nodes: $p = (v_1, \ldots, v_k)$.

Path p is a valid path from s to g if $\forall i = 0, \ldots, k-1 : (v_i, v_{i+1}) \in E$, $v_0 = s$, $v_k = g$ and additionally no node occurs twice in the path. The weight of path p denoted $w(p)$ is computed as $w(p) = \sum_{i=0}^{k-1} W((v_i, v_{i+1}))$. The shortest distance from s to g denoted $d(s, g)$ then corresponds to the minimum weight of a path from s to g. We define P to be the set of all valid paths from s to g, so $p \in P \Leftrightarrow p$ is a valid path from s to g. The shortest distance from s to g is then defined as $d(s, g) = \min_{\forall p \in P} w(p)$. If there exists no valid path from s to g in the graph ($P = \varnothing$), then $d(s, g) = \infty$. There can be multiple shortest paths for a pair of start-goal nodes s and g.

In our case, we assume that there are multiple pairs of s and g for which we need to compute the shortest distance. We assume that the graph does not change in between those queries. Because of this, we can allow our algorithms to precompute some structures for the given graph in order to answer subsequent queries faster. To obtain those structures, the algorithm will need some time, which we denote as the *preprocessing overhead* of the method. When comparing query times, we do not consider this overhead. We simply assume, that if the amount of queries gets large enough, the time saved by the query speedup will outweigh the preprocessing overhead significantly and the proportion of the overhead for one query converges to zero.

Note that in this work, we focus on distance queries, meaning we want to return the distance $d(s, g)$ as quickly as possible. We do not want to obtain actual paths, only distances. The performance of various methods can change significantly if we also require to obtain actual paths and not just distances.

Chapter 3

Survey

Since the computation of the shortest distances between two places is a very common problem in computer science, there are many articles regarding this topic. In this part, we will present various possible approaches for computing the shortest distances. We will limit ourselves to approaches that work well for real road networks. As mentioned previously, one useful attribute of road networks is the fact, that they are almost planar graphs. This means that the average degree of those graphs is fairly low. This structural property of road networks is beneficial for many of the mentioned methods and their performance would diminish on graphs with large average degrees.

The methods can be distinguished into two basic categories. One category consists of methods that do not use any preprocessing of the input graph and only answer queries with the information contained in it. The other category consists of methods that use some sort of preprocessing of the input graph to obtain a different representation of the input graph, which is then used to answer queries. Those methods differ in what is precomputed and how.

The most basic approach for computing the shortest distance between two points s and g in a graph is the well-known *Dijkstra's algorithm* [1]. This algorithm incrementally builds a shortest-path tree from the start node s. When the goal node g is settled, the distance to it from the node s in the tree is the shortest distance $d(s, g)$. The downside of this algorithm is the fact, that each query is processed individually and each time the graph must be searched again. Dijkstra's algorithm is, therefore, a representative of the first category, as it only uses information contained in the original graph.

A trivial improvement of the Dijkstra's algorithm is the Bidirectional Dijkstra [2]. We can run two Dijkstra searches, one from s expanding forward (outgoing) edges and one from g expanding backward (incoming) edges. When the two searches meet, we get a candidate for the shortest distance. This reduces the search space approximately by a factor of 2. Modified variants of Bidirectional Dijkstra are used in some of the more complex methods.

If we know that we will need to answer multiple queries and the graph will not change between the subsequent queries, it makes sense to precompute some auxiliary structures that can be used to answer the queries faster. The precomputation will take some time, but we only need to precompute the structures once. The more queries we then answer, the more we benefit from the precomputation and it becomes more meaningful to actually do the precomputation.

The easiest precomputation method is precomputing a *distance matrix*. One can easily precompute a matrix of all distances between all pairs of points in time $\mathcal{O}(n^3)$ (where n is the number of nodes in the graph) using the *Floyd–Warshall algorithm* or even faster using n runs of the Dijkstra's algorithm with a good heap implementation of the priority queue. Queries can then be answered by a simple table lookup so in time $\mathcal{O}(1)$.

The downside of this approach is the fact, that the resulting distance matrix gets large quite quickly, as the memory complexity for the distance matrix is $\Theta(n^2)$. For example, a road network of Prague with the closest surroundings consists of 28 686 nodes. A full distance matrix (when representing each distance as a 4-byte integer) already needs more than 3 GB of space (for reference, the graph representation of the same network required for the Dijkstra's algorithm only needs around 2 MB of memory). If we want to precompute a full distance matrix for a whole country, we quickly run into problems. For example, a complete road network for Florida used in the 9th DIMACS Implementation Challenge [1] consists of over one million nodes. This means that we would need over 4 terabytes of memory for the full distance matrix for such network. This makes this approach inapplicable for large road networks in most of the cases.

On the other hand *Dijkstra's algorithm* does not have any memory overhead. Only the actual graph needs to be stored in memory. The downside is then the time required to answer queries because an actual search has to be done for each query. For our example road network of Prague, one random query took over 6 milliseconds using this algorithm. For applications that need to answer billions of such queries, we would have to wait multiple months for the results. Ideally, we would like to decrease the time needed to answer large query sets to at most units of hours, but without the huge memory overhead required for the distance matrix.

There are a lot of methods that lie somewhere between those two extremes. Most of those methods rely on some sort of preprocessing of the input road network. Usually, those methods add new edges to the graph, precompute a distance matrix for some subset of nodes (not the full distance matrix though), or they add some additional attributes to the nodes or edges in the graph to speed up the queries. When choosing a method, there is a certain trade-off we need to consider. Most of the time, methods with faster query times also require a bigger memory overhead. The size of the available memory and the size of the input graph should be therefore considered where choosing a method.

3.1 Best-First Search Based Approaches

Some methods can answer random queries faster than the basic Dijkstra's algorithm without requiring any preprocessing. Those methods are usually based on the best-first search idea. Those methods explore the graph by always expanding the most promising node at the given moment, usually based on some heuristic.

A typical representative is the A* search algorith [3]. This algorithm works similarly as the basic Dijkstra's algorithm, only when choosing a node to expand at the given moment, A* chooses the node v that minimizes $f(v) = g(v) + f(v)$ where $g(v)$ is the weight of the

[1] Information about the 9th DIMACS Implementation Challenge on shortest paths can be found on <http://users.diag.uniroma1.it/challenge9/>.

path from the start node to v, and $h(v)$ is an estimation of the weight from v to the goal node. This estimation is acquired using some heuristic. If the heuristic estimates the weights well enough, this can speed up the search significantly as nodes in the direction of the goal node are preferred. Many heuristics can be used in A* search with various properties. This is outside the scope of this work but we refer the interested readers to [4] where properties of heuristics and their design are discussed. For real road networks where more subsequent queries have to be answered, best-first search based approaches are generally outperformed by methods that preprocess the input graph in some way [3].

3.2 Highway Hierarchies

One of the first methods that uses preprocessing of the road network in order to speed up the queries is called Highway Hierarchies [5, 6]. This approach is based on the fact, that we can usually split the road network into multiple levels based on the importance of the individual roads. For example, highways will be mostly in the highest level, because they will be used in a lot of the queries when going far away, while small roads connecting some small villages will be in the lower levels, as those roads are only used in a small subset of very specific queries.

In Highway Hierarchies, we first define a neighborhood for each node to consist of its H closest neighbors. Then an edge (u, v) is called a highway edge if there is some short path $\langle s, \ldots, u, v, \ldots, t\rangle$ such that neither u is in the neighborhood of t nor v is in the neighborhood of s. The unification of all highway edges then defines the first level of the highway hierarchy. We then contract the network, meaning we remove low degree nodes, and then find highway edges in the new network again, obtaining the next level of the hierarchy. This process is applied recursively (identifying the highway edges followed by contraction). This way we obtain the highway hierarchy.

For queries, the idea is that when we are far away from start or goal, only high-level edges need to be considered (when going from Paris to Barcelona, we will probably use some highways and not some small roads). The search algorithm is, therefore, a modified Bidirectional Dijkstra that starts in the lowest level of the hierarchy. When the search gets far enough from the start node (or goal node in the other direction), the algorithm switches to a higher level of the hierarchy. Edges in lower levels of the hierarchy are then not relaxed. The farther away s and g are from each other, the more edges are skipped during the search as the algorithm gets to higher levels of the hierarchy. Because of the structure of real road networks, this approach is very successful. [7]

3.3 Contraction Hierarchies

Contraction Hierarchies is a method that is similar to Highway Hierarchies but takes the idea even further. In Contraction Hierarchies [8, 9, 10], we also obtain a hierarchy for the network. The difference is that in this case, each node has its own level of the hierarchy called rank. Nodes with higher importance (such as highway entrances and exits) have high ranks while junctions on small roads have lower ranks.

During the preprocessing phase, all nodes in the graph are contracted. The contraction of a node is a process during which the node is removed, but new edges might have to be added into the graph to preserve the shortest distances between nodes that have not been contracted yet. Those edges are called shortcut edges. The edge set for the graph used for the queries is obtained as a union of the original edges and the shortcut edges. Additionally, the ranks of each node determined by their order during the contraction process are utilized in the query algorithm.

The queries work in a bidirectional fashion. A modified Dijkstra's algorithm is run both from the start node and the goal node. In each direction, only edges going from nodes with lower ranks to nodes with higher ranks are relaxed. Each time a node is settled in one direction that has already been settled in the other direction, we get a candidate for the shortest path. [8]

3.4 Transit Node Routing

The main idea of Transit Node Routing [7, 11, 12] is that for every road network, there is a fairly small subset of nodes that are used in a large portion of the queries. This subset usually consists of the large important junctions, highway entrances and exits. The nodes of this subset are called transit nodes. If we precompute pairwise distances for all pairs of transit nodes and also distances to closest transit nodes (called access nodes) for all nodes in the graph, we can then answer a large portion of the queries by combining those precomputed distances. Since the transit nodes set is fairly small, the memory required to store the precomputed pairwise distances for this set and the memory required to store the closest transit nodes for all nodes will be much smaller than the memory that would be required for the full distance matrix.

Queries, where the precomputed distances for the transit node-set can be utilized, can be answered very quickly. If we want to obtain the shortest distance from s to g, we have to obtain all access nodes for both s and g, then for each pair of a access node of s and b access node of g, we just compute obtain new candidate for the shortest distance as $d(s, a) + d(a, b) + d(b, g)$. We then choose the minimum from all the a and b combinations. Since we have all three terms in the sum already precomputed, obtaining those candidates means only three table lookups and two sum operations. The problem is that for queries where s and g are relatively close to each other, the precomputed distances can not be utilized, because the shortest path might not contain any transit nodes. When you are for example going from a hotel to a restaurant just a few blocks away, there is a big chance that you will not need to go through any large important junctions. Those queries are called local queries. For those, we have to fall back to some other method. We can either use Dijkstra or some other method such as previously mentioned Highway Hierarchies or Contraction Hierarchies.

3.5 Arc Flags

In Arc Flags [13, 14] the input graph $G = (V, E)$ if split into k regions. For each edge $e \in E$, we will store k boolean flags called arc flags. Flag $i \in 0, \ldots, k-1$ for edge e is set to true,

if there exists a shortest path to some node in region i that contains edge e, otherwise it is set to false. The query algorithm is then a slightly modified Dijkstra. When computing the shortest distance from s to g, we first obtain the region r for the node g, and then we run a Dijkstra search only expanding edges for which flag r is set to true. We can also obtain the region for s and then use a modified Bidirectional Dijkstra.

The performance of Arc Flags depends on the partitioning of the input graph into k regions. There are various approaches to obtaining those partitions. If we are working with a real road network, a simple option is to geometrically split the graph into k rectangular cells of equal size. More sophisticated approaches based on for example Quad-trees or kd-Trees tend to provide better speedups.

The trade-off between speedup and memory overhead can also be controlled by choosing the appropriate number of regions k. For a larger amount of regions, the query times are lower, because more edges can be discarded during the search speeding up the query. On the other hand, since we have to store k flags for each edge in the graph, by increasing the number of regions k, we are also increasing the memory overhead needed by the flags. [13]

3.6 Hub Labels

Hub Labels [15, 16, 17] is the fastest of the mentioned methods, only slower in terms of queries than the full precomputed distance matrix. For an input graph $G = (V, E)$, this method computes a forward label $L_f(v)$ and a backward label $L_b(v)$ for each node $v \in V$. The forward label $L_f(v)$ consists of a sequence of pairs $(w, d(v, w))$ where w is a node $w \in V$ called hub and $d(v, w)$ is the distance to the hub from the node v. The backward label is constructed in a similar fashion, except there are pairs $(w, d(w, v))$. The labels must obey a so called cover property, which says that for each pair of nodes s and g, the set $L_f(s) \cap L_b(g)$ must contain at least one hub v that is a part of some shortest path from s to g. The queries are then very straightforward, if we want to obtain the shortest distance from s to g, we just find the node $v \in L_f(s) \cap L_b(g)$ that minimizes $d(s, v) + d(v, g)$.

The main downside of Hub Labels is the fact, that its memory overhead is the largest of all the mentioned methods (again when not counting the full precomputed distance matrix). Storing all the labels requires up to two orders of magnitude more space than storing the graph itself. This is by orders worse than for the other methods mentioned so far making this approach impractical for large input graphs.

3.7 Hub Labels Compression

Since the main problem of Hub Labels is the large memory overhead, it is a good idea to try to compress the labels somehow to decrease the memory overhead of the method while still keeping the query times reasonably low. A great compression technique is described in [15]. It is based on the idea that hub labels can be represented as trees. We can represent a label $L(u)$ as a tree T_u rooted at u and having the hubs in $L(u)$ as vertices. For each two vertices $v, w \in L(u)$, there is an arc (v, w) in T_u if the shortest path from v to w in the input graph contains no other vertex from $L(u)$.

The compression then exploits the fact that trees representing labels of nodes that are close to each other in the graph are similar in the sense that they have a lot of subtrees in common. We can only store each distinct subtree only once with a unique id, and then in the label for u we can only store the ids of the subtrees forming the tree T_u. The authors also use lossless compression for the actual subtrees based on a recursive representation saving further space. [15]

The compression can significantly decrease the memory overhead, but the price for that is the fact that when answering a query, we must first recover (decompress) the labels for both nodes before we can answer the query as described in section 3.6.

3.8 Combinations

Some of the methods can be combined to achieve further performance improvements. The individual methods can be mostly divided into two groups. One is hierarchical approaches, where we impose some hierarchy on the input graph. Highway Hierarchies, Contraction Hierarchies and Transit Node Routing all fall in this category as all of those methods make use of the idea that we can split the network into roads and junctions of various importance. The other category is goal-directed approaches that try to direct the search towards the goal node as quickly as possible by preferring nodes or edges that are getting the search closer to the goal node. This can be viewed as a heuristic approach. Arc Flags represent this approach.

In practice, combining hierarchical and goal-directed methods usually yields great results [18]. Combinations of Highway Hierarchies with Arc Flags, Contraction Hierarchies with Arc Flags, and Transit Node Routing with Arc Flags all work very well in practice. Especially the combination of Transit Node Routing with Arc Flags is very practical because the query times get very close to the query times of Hub Labels (just by tens of percent worse) while having significantly smaller memory overhead. For further information about possible combinations and their performance, we refer the interested readers to [18].

Chapter 4

Solution Approach

In this thesis, we use Transit Node Routing based on Contraction Hierarchies extended by Arc Flags. We decided to go with this approach because this is the second-fastest currently known method only being slower than Hub Labels. Hub Labels require significantly larger memory overhead though and the performance gain is not that significant [19]. In this chapter we describe all three methods in more detail. We first describe Contraction Hierarchies in section 4.1, then Transit Node Routing in section 4.2 and then Arc Flags in section 4.3.

4.1 Contraction Hierarchies

Let us now focus on Contraction Hierarchies in more detail. Contraction Hierarchies take advantage of the hierarchical nature of real road networks. While Highway Hierarchies build a hierarchy consisting of multiple levels of roads of increasing importance as described in section 3.2, Contraction Hierarchies take this approach even further and each node has its own level in the hierarchy.

The main idea of the method is that we order the nodes in the graph by their importance. We then start contracting the nodes in order of ascending importance. When contracting a node, the node is removed from the graph, but to preserve the shortest distances between the remaining nodes, new edges might need to be added into the graph. Those edges are called *shortcut edges*. This method, therefore, alters the original input graph by adding new edges into the graph.

Additionally, the query algorithm for Contraction Hierarchies also uses the importance ordering of the nodes when finding the shortest distance. The algorithm works in a bidirectional fashion, so we start a forward search from the start node and at the same time a backward search from the goal node. In both directions, we only expand edges going from lower priority nodes to higher priority nodes. This allows the query algorithm to skip a lot of edges when processing each node, and due to the added shortcuts, the algorithm will still find the optimal solution. [8]

Let us now talk about the individual steps of the algorithm.

4.1.1 Contraction

The primary process of Contraction Hierarchies is the actual node contraction. We need to contract all nodes in the input graph in a certain order to obtain shortcuts in the graph. The order in which we contract the nodes can be arbitrary, the order does not affect the correctness of the algorithm. Some orders are better in terms of performance of the obtained hierarchies than others though. We will discuss how to obtain good orders in the next subsection. For now, let us assume that we already have an order and we can start contracting nodes using that order.

When contracting node v, we are dealing with a graph $G' = (V', E')$. This graph can already contain fewer nodes than the original graph because some nodes could already be contracted, and the edge set can be already expanded by new shortcut edges during those contractions. When contracting the node, we will obtain a new graph $G'' = (V'', E'')$ where $V'' = V' \setminus \{v\}$ and $E'' = E' \cup \{\forall e : \ e \text{ is a necessary shortcut}\}$. This means, that we actually remove the node from the graph, but to preserve the shortest distances, we might need to add some new shortcut edges into the graph.

Here we need to consider all nodes u such that $(u, v) \in E'$ and w such that $(v, w) \in E'$. For each pair u, w we need to check if there exists some shortest path in the graph that contains edges (u, v) and (v, w). Let us now assume that there exists such shortest path and it goes from some node a to some node b. If there does not exist a different shortest path in the graph from a to b that does not go through v, then after the removal of v, the shortest distance from a to b would not be preserved. Therefore we need to add a shortcut edge from u to w with the weight $c(u, v) + c(v, w)$. In the worst case, we might need to add a shortcut edge for each pair u, w. An example of the node contraction process is given in figure 4.1.

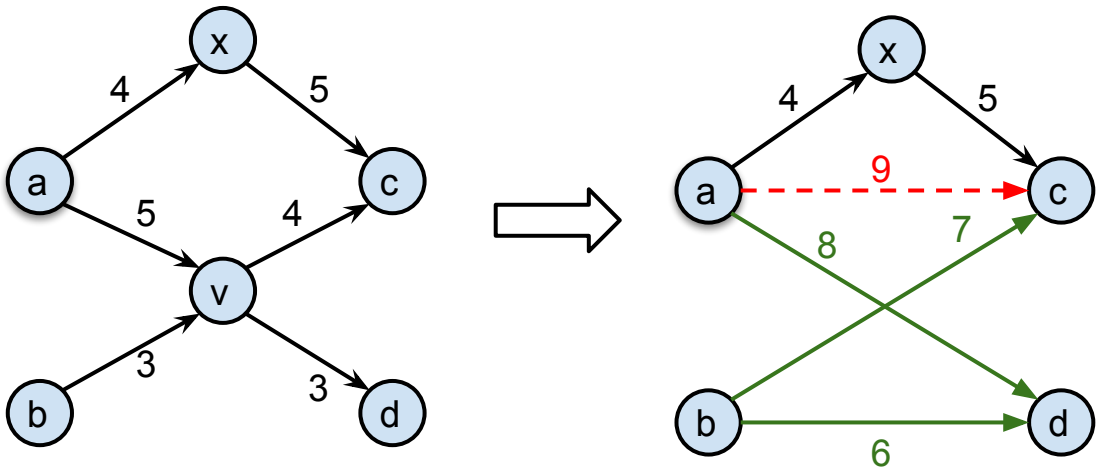


Figure 4.1: Node contraction example. We are contracting node v. In this case, we have to add all the green shortcut edges to preserve shortest distances among a, b, c and d (neighbors of v). The red shortcut from a to c does not have to be added. If we do not add it, there still exist a shortest path from a to c with the same length as the one going through v. Adding it does not affect the correctness of the query algorithm though.

To obtain all the shortcuts that need to be added when removing node v, we could perform a shortest-path search in G' from each possible start node, ignoring node v, until all forward neighbors of v have been found. From the computed distances, we can then decide which shortcuts are needed.

Since this would require an extensive amount of individual Dijkstra's algorithm runs, this could slow the whole preprocessing process a lot. Therefore an approximation approach inspired by [20] is often used. One can realize, that adding shortcuts even if they are not necessary does not harm the algorithm in a sense, that even if we add unnecessary shortcuts, the algorithm will still return the actual shortest distances. We could theoretically add a shortcut for each pair u, w. This would lead to an excessive amount of new edges in the graph though, so it would increase the memory overhead by a fair amount.

We want to add as little shortcuts as possible, but we also do not want to spend too much time determining which of the shortcuts are needed and which can be omitted. For this, we can perform a process called a single-hop backward search. For each edge $(x, w) \in E'$ we store a *bucket entry* $(W(x, w), w)$ with node x. We can then limit the forward search from u to distance

$$d(u, v) + \max_{w:(v,w)\in E'} W(v, w) - \min_{x:(x,w)\in E'} W(x, w). \tag{4.1}$$

When reaching a node x, we scan its bucket entries. Fr each entry (C, w), we can infer that there is a path from u to w of length $d(u, x) + C$.

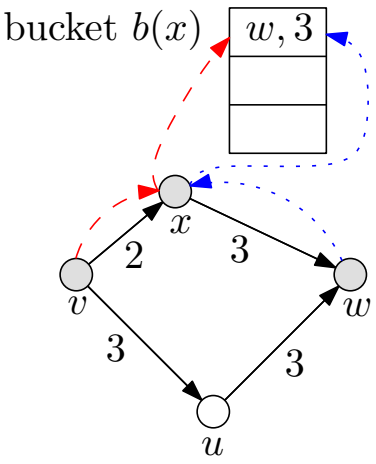


Figure 4.2: Example of the bucket entries utilization. When we are relaxing the edge from v to x, we can immediately scan the bucket entries of x. In this case, we immediately obtain the cost of the shortest path from v to w going through x without actually processing x. [21]

The buckets serve us as a sort of look ahead. Each time when scanning the bucket of some node, we can obtain some shortest path witnessing the unnecessity of some shortcut without actually processing the node. A schema showing how the buckets are utilized is shown in figure 4.2.

While using this bucket approach, we can also limit the number of hops (edges) used in any path $\langle u, \ldots, w \rangle$, and we can limit the total search space size of a forward search. The hop

limit denotes the maximum number of edges on paths that we will consider when deciding whether a shortcut is really necessary. Because each shortcut replaces exactly two edges, the alternative shortest path proving that the shortcut is unnecessary usually does not contain many more edges. Limiting the number of hops and the search space size allows us to speed up the contraction process. As long as we make sure to always add a shortcut (u, w) when we have not found a path from u to w witnessing that the shortcut is unnecessary, subsequent queries will still be answered correctly. [8]

Limiting the number of hops and search spaces can speed up the contraction process. The bigger the limit, the more shortcuts will get added into the graph, because we will possibly add shortcuts that are not necessary, only we were not able to find a witness. This means, that by limiting the number of hops and the search space size, we can speed up the preprocessing phase, but the performance of the obtained Contraction Hierarchy will be slightly worse because it will contain more unnecessary edges.

4.1.2 Node Ordering

As mentioned in section 4.1.1, the performance of the obtained Contraction Hierarchy is greatly influenced by the order in which we contract the nodes. We want to contract the nodes in order of ascending importance, meaning we want to contract nodes that will need to be expanded in more of the queries later. To obtain the actual importance of the nodes, we would need to know all the queries that our hierarchy will be used to answer in advance.

We instead try to approximate the importance of nodes by a linear combination of different importance terms. The most commonly used terms are:

- *Edge Difference.* The difference between the number of shortcuts added by removing the node and the original degree of the node. We prefer to contract nodes where not many new shortcuts will be added earlier.
- *Uniformity.* We do not want to contract nodes that are close to each other in the graph soon after each other. We should try to contract nodes in various parts of the graph uniformly often to obtain hierarchies with good performance for random queries.
- *Deleted neighbors.* We can also take the number of neighbors that have already been contracted into account. This heuristic turns out to be quite effective in practice.
- There are other possible terms that could be utilized. For example *Voronoi Regions* are mentioned in [8].

Since the described terms can change in time due to some nodes in the graph being contracted, we usually store the nodes along with their priorities in a priority queue. Each time when contracting a node we choose the node with the lowest priority currently in the queue, and we update priorities for its neighbors. [8]

The final order of the node contractions assigns a *rank* to each node. These ranks are then used in the query algorithm to speed up the search.

4.1.3 Query

We already know how to contract nodes and how to obtain good orderings of nodes for the contraction process. The only thing that is left to be answered is how to actually use the structures obtained by the preprocessing phase for the actual queries.

An output of the preprocessing phase is a graph G containing all the original nodes and edges from the input graph, but also additional shortcut edges added into the graph during contraction of some nodes. Additionally, each node has a rank assigned, determined by the order in which the nodes were contracted. Nodes contracted later have higher ranks.

Let us now split G into an *upward graph* $G_\uparrow$ and a *downward graph* $G_\downarrow$. Graph $G_\uparrow$ contains such edges (u, v) that the rank of u is lower than the rank of v, while $G_\downarrow$ contains such edges (u, v) that the rank of u is higher than the rank of v.

Assume that we want to answer a query which asks about the shortest distance from some start node s to some goal node g. The actual query algorithm is a modified bidirectional Dijkstra's algorithm that starts a search from s in $G_\uparrow$ and simultaneously a search from g in $G_\downarrow$. The algorithm alternates between a forward and a backward search. Whenever a node is settled in one direction that is already settled in the other direction, we get a new candidate for the shortest path. We can abort the search in one direction if the weight of the smallest element in the queue for that direction is at least as large as the best candidate path found so far. Two examples of queries answered by the Contraction Hierarchies query algorithm are given in figure 4.3. [8]

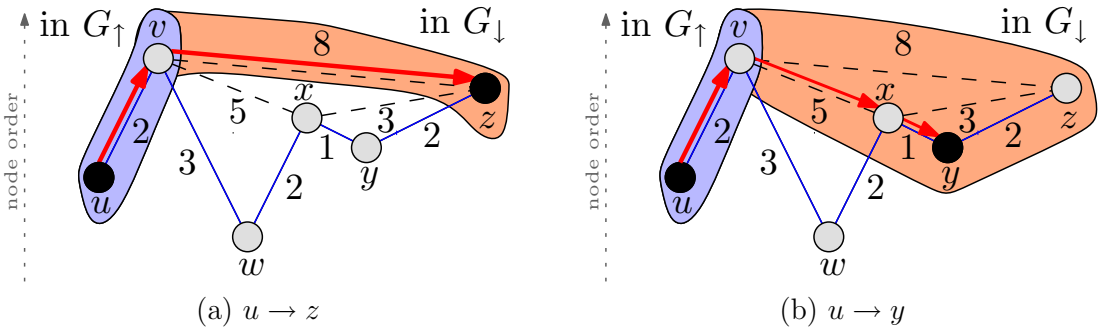


Figure 4.3: Two Contraction Hierarchy queries. Normal lines denote original edges, dashed lines denote shortcut edges, red thicker arrows correspond to the found path. The blue frame encapsulates all the nodes processed in the forward search from the start node and the red frame encapsulates all the nodes processed in the backward search from the goal node. [21]

4.1.4 Negative Memory Overhead When Using Contraction Hierarchies

One interesting property of the Contraction Hierarchies query algorithm is, that it always expands edges going from nodes with lower ranks to nodes with higher ranks. This property allows us to store each edge only once at the node with the lower rank because it will never be expanded when processing the node with the higher rank. This also holds for bidirectional

edges. Those edges will be expanded by the search in $G_{\uparrow}$ in one direction and by the search in $G_{\downarrow}$ in the other direction.

When performing a Dijkstra's algorithm search in the original graph, we need to actually store bidirectional edges at both of their endpoints, so we essentially need to store those edges twice. Due to this property, the memory overhead of Contraction Hierarchies data structures might actually be negative. In the ideal case, each road in the original graph is a bidirectional road, so the memory required for those edges in the Contraction Hierarchies data structure will be half of the memory required for the Dijkstra's algorithm. Additionally, Contraction Hierarchies will contain some shortcuts, but if there are fewer shortcuts than half of the edges in the original graph, then the complete memory overhead of Contraction Hierarchies is negative, meaning the structures required for the Contraction Hierarchies query algorithm need less memory than the actual graph representation for the Dijkstra's algorithm.

4.2 Transit Node Routing

Transit Node Routing is based on the idea, that when traveling far away, you usually join some of the larger roads close to your starting point, and when you get close to your destination, you also leave some larger road using some exit. Those access points and exits are called *transit nodes*. Usually, for each location, there is only a small amount of relevant access points and exits. When you, for example, want to travel from Paris to Barcelona, it does not matter where exactly in Paris you will start your journey, because you will probably try to get to some highway as soon as possible, and then you will travel along highways until you get close enough to your destination in Barcelona. Then you will exit the highway using some exit and navigate to your final destination along some smaller roads.

Transit Node Routing tries to somehow identify a set of access points and exits in the graph that will be used very often when traveling. The full distance matrix containing distances between all pairs of transit nodes is then computed, along with the forward and backward access nodes for each node in the graph. Forward access nodes for a node v is such a set of transit nodes that when going far enough, you will always pass through one of these nodes. Analogically, backward access nodes for a node w is such a set of transit nodes, that when arriving to w from far away, you will always pass through one of these nodes.

Distance from v to all of its access nodes (in both directions) is also precomputed and stored with the access nodes. When a query then asks for the shortest distance from a start node s to a goal node g, then assuming s and g are far enough from each other, the query can be answered by trying all forward access nodes a of node s and all backward access nodes b of node g. For each pair of a, b, we get a new candidate $d(s, a) + d(a, b) + d(b, g)$. Since all three of those distances are already precomputed, obtaining this candidate consists only of three table lookups and the addition of the values. If we try all possible combinations of a and b, the shortest obtained distance is the actual shortest distance in the graph.

One thing to note here is that we have been talking about the fact, that the nodes must be far enough from each other in order to answer the query as described previously. If we are only traveling a short distance, for example, if we want to get from a hotel in Paris to a close restaurant in Paris, it is very likely that we will not pass through any of the transit nodes in the graph. Those queries are usually called *local queries*. The pre-computed values do not

help us with those queries, so we need to answer those using some different fallback method. Using Dijkstra's algorithm as a fallback method already yields solid results, because for the local queries we already know that the start and target nodes will be close to each other so the algorithm will only expand a small part of the graph. Queries, where a big portion of the graph would need to be expanded, are answered using the transit nodes. A more sophisticated method for the local queries could be also used. We use Contraction Hierarchies to answer the local queries in our implementation. We also need a way of determining whether a query is a local query and we need to use the fallback algorithm, or if we can use transit nodes to answer the query. [11]

4.2.1 Transit Node Routing as a Framework

As the previous part suggests, Transit Node Routing is more a framework than a single algorithm. There are various approaches to the implementation of Transit Node Routing. Each of those implementations must tackle the following problems:

- Obtaining a set $\mathcal{T} \subseteq V$ of transit nodes.
- Computing a distance table $D_{\mathcal{T}} : \mathcal{T} \times \mathcal{T} \to \mathbb{R}_0^+$ of shortest distances between all pairs of transit nodes.
- Finding access nodes for each node. This means obtaining a forward (backward) access nodes mapping $A^{\uparrow} : V \to 2^{\mathcal{T}}$ ($A^{\downarrow} : V \to 2^{\mathcal{T}}$) such that for any shortest s-g-path P containing transit nodes, $A^{\uparrow}(s)$ ($A^{\downarrow}(s)$) must contain the first (last) transit node on P.
- Obtaining a locality filter $\mathcal{L} : V \times V \to \{true, false\}$ such that $\mathcal{L}(s, g)$ is always true when no shortest path from s to g is covered by a transit node. Note, that false positives are allowed, this means $\mathcal{L}(s, g)$ can be true even when the shortest path contains a transit node. This only means that we would use our fallback algorithm to answer a query that could have been answered using transit nodes. While this does not affect the correctness, it affects the performance negatively, since obtaining the distance using the fallback algorithm takes more time than obtaining it using transit nodes. [22]

There are many approaches to solving those problems. A grid-based implementation based on splitting the input into rectangular cells of equal size (here actual geographic positions of nodes have to be used) and an implementation based on Highway Hierarchies are presented in [11]. A more sophisticated (and also faster) implementation based on Contraction Hierarchies is then suggested in [22].

4.2.2 Transit Node Routing Based on Contraction Hierarchies

Let us now describe how Contraction Hierarchies can be used to solve the problems from the Transit Node Routing framework.

Obtaining a set of transit nodes using Contraction Hierarchies is really simple. We can just choose k nodes with the highest rank in the Contraction Hierarchy. Remember that

nodes with higher ranks should mean more important nodes, so usually for example highway exits will have higher ranks. One benefit of this approach is that we can choose any k we want. A lower amount of transit nodes means that the memory overhead will be smaller, as we will only need a distance matrix for fewer transit nodes, but the performance will be slightly worse because the average distance from nodes to their access nodes will be higher (as there are fewer transit nodes in the graph), which means more queries will be classified as local queries and those have to be answered by a slower fallback algorithm. Also consider the fact, that when $k = n$, Transit Node Routing degrades to a full distance matrix, because in that case every node is a transit node and we will precompute distances for all pairs of nodes in the graph.

There are multiple options for computing the distance matrix for transit nodes. A simple approach is to run the Dijkstra's algorithm k times from each transit node until it settles all the other transit nodes. The distance matrix can be computed faster by adjusting the method described in [20].

When computing forward access nodes for node s, all we have to do is run a forward Contraction Hierarchies query from s, that will not relax edges leaving transit nodes. When the search runs out of nodes to settle, the settled transit nodes form the set of forward access nodes for s. Computing backward access nodes works similarly, only we perform a backward Contraction Hierarchies query. Proof of correctness for this procedure along with some possible improvements is included in [22].

The last thing we need is the locality filter. The simplest locality filter is based on search spaces. When computing the access nodes, we can also save the forward and backward search spaces for each node. The forward search space of node s contains all nodes settled by the forward Contraction Hierarchies search from s that are not transit nodes. The backward search space is obtained analogically. When we want to obtain the shortest distance from some start node s to some goal node g, we can check forward search space of node s and the backward search space of node g. If these two search spaces are disjoint, all shortest paths from s to g must go through some transit nodes, and therefore we can set $\mathcal{L}(s, g) = false$. On the other hand, if the intersection of those two search spaces is non-empty, there might be a shortest path not going through any of the transit nodes, and therefore we set $\mathcal{L}(s, g) = true$.

4.2.3 Query

Once we have all the necessary structures ready, queries can be answered really quickly.

When given a start s and a goal g, we must first check the locality filter $\mathcal{L}(s, g)$. If the locality filter returns true, we have to use our fallback algorithm, because the shortest path might not contain any transit nodes. Dijkstra's algorithm or for example Contraction Hierarchies can be used in this case (we use Contraction Hierarchies).

If the locality filter returned false, then we can consider all nodes a that are forward access nodes of s and all nodes b that are backward access nodes of g. For each such pair a, b, we obtain a new shortest distance candidate obtained as $d(s, a) + d(a, b) + d(b, g)$. Since all those three values are already precomputed ($d(a, b)$ in the transit nodes distance table and $d(s, a)$ and $d(b, t)$ are stored together with the access nodes for the respective nodes), we can obtain those distances very quickly. We then just have to check all possible pairs and return the best candidate. A schematic representation of this process is shown in figure 4.4. [22]

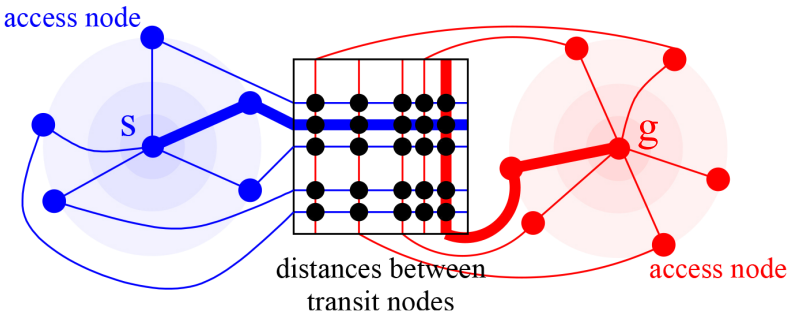


Figure 4.4: Schematic representation of Transit Node Routing. [7]

4.3 Arc Flags

We have already described the main idea of Arc Flags in section 3.5. This idea can be slightly modified and then used on top of Transit Node Routing to speed up the queries even further. The main difference when using Arc Flags together with Transit Node Routing is the fact that when using Arc Flags alone, we compute flags for each individual edge of the graph, while for the combination with Transit Node Routing we compute flags for pairs (v, w) where v is a node from the graph and w is its access node. This makes sense because in Transit Node Routing we do not relax individual edges, but instead, we only compare distances for pairs of access nodes.

4.3.1 Partitioning

When combining Arc Flags with Transit Node Routing, we first split the graph into k arbitrary regions. There are no requirements for this partitioning, except for the fact that each node must be assigned to exactly one region. Partitionings that are in some way uniform and try to gather nodes that are close to each other in one region will perform the best, but the approach works with any partitioning.

We use a simple partitioning that in the beginning randomly chooses k nodes as the representatives for the regions. Each region is then expanded by gradually adding nodes closest to it. We use *single-linkage*, meaning the distance from a node to a region equals the distance from the node to the closest node in the cluster. We expand the regions in a round-robin manner (we expand the regions one after another) to ensure that the regions will contain an equal number of nodes. With this approach, it might happen that one region will consist of multiple unconnected clusters (if the region we are expanding is unreachable from all the nodes, we simply expand it by adding some other unassigned node into the region, starting another cluster). This simple clustering method already improves the performance significantly.

4.3.2 Computing Arc Flags for the Access Nodes

Assume each node has a region assigned. We then precompute arc flags for each pair of v, w where v is a node in our graph, and w is an access node of v. For each such pair, we will

have k flags. Flag i ($0 \leq i < k$) is set to $true$ if there exists a shortest path from v to some node in region i going through the node w, otherwise it is set to $false$.

We can compute the distances from both v and w to all the other nodes in the graph using for example Dijkstra's Algorithm. By comparing those distances, we can then obtain the arc flags. If there exists a node x from a region i such that $d(v, w) + d(w, u) = d(v, u)$, then for the access node w we set the flag for region i to $true$, because a shortest path from v to u goes through w (in theory, there can be another shortest path with the same weight not going through w, but setting the flag to $true$ in this case is a conservative approach that ensures the results will be correct). If there exists no such node x from a region i, we set the flag to $false$.

4.3.3 Utilizing Arc Flags for Queries

The precomputed arc flags can be used during the Transit Node Routing query algorithm to reduce the number of access nodes we have to check in both directions. Let us assume that we want to answer a query about a distance from a start node s to a goal node g. Let us further assume that the region of s is i and the region of g is j. Now when considering forward access nodes of s, we only need to try such nodes for which the arc flag j is set to true (meaning there exists a shortest path through it to some node in the region of the target node). Analogically, when considering backward access nodes of g, we only need to try access nodes with arc flag i set to true. This obviously speeds things up, as fewer combinations of access nodes have to be tried. One example of such query is given in figure 4.5.

Arc flags need additional memory for the region numbers for all nodes and then for the flags stored with each access node. For each node in the graph, we need to store one integer denoting the region of the node. This is linear with respect to the number of nodes in the graph. As for the flags, in this thesis we use 32 regions, so for every access node, we need to store an additional 4 Bytes of information (32 boolean flags). Since for large transit node-sets the memory required for the transit node-set distance table outweighs the memory required for the access nodes significantly, the arc flags that are attached to the access nodes do not make a big difference in terms of memory overhead when working with large transit node-sets.

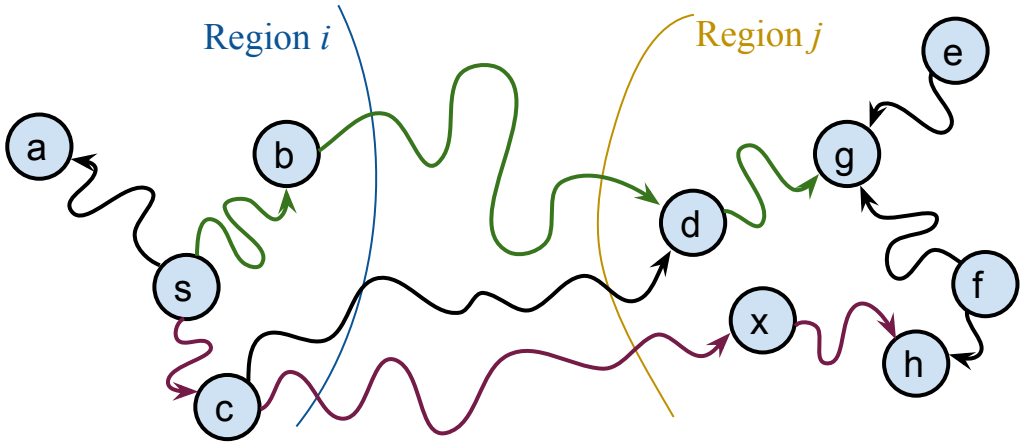


Figure 4.5: Example Transit Node Routing query using Arc Flags. We need to compute the shortest distance from s to g. The start node s belongs to region i and has three access nodes a, b and c. The goal node g belongs to region j and has three access nodes d, e and f. In this example, for access node a the arc flag j can be set to $false$, as there is no node in region j for which the shortest path from s goes through a. For both b and c the flag j must be set to true. For b there exists a shortest path going from s through b to some node in region j, in this case the node in region j is g, the goal of the query (the path is indicated by dark green). The path from s through c to g is not the shortest path from s to g, but there exists another node in region j for that the shortest path goes through c and this node is h (the shortest path is indicated by dark purple). The curves in this picture show paths, that might consist of multiple individual edges.

Chapter 5

Implementation

To achieve the best possible performance, we have decided to use the *C++* language for our implementation. The implementation is split into two major components. Those components are the *preprocessor* and the *library*. The *preprocessor* is an executable application that allows the user to precompute data structures that can be used to answer queries quickly. The *preprocessor* also allows benchmarking of the computed data structures on a given query set. The *library* is implemented as a shared library, that can be invoked from *Java* since one of our main goals is to integrate our implementation into AgentPolis. The preprocessor is described in more detail in section 5.1 and the library is described in section 5.2. The main components of our implementation and their belonging to either the preprocessor or the library (or both) are illustrated in figure 5.1.

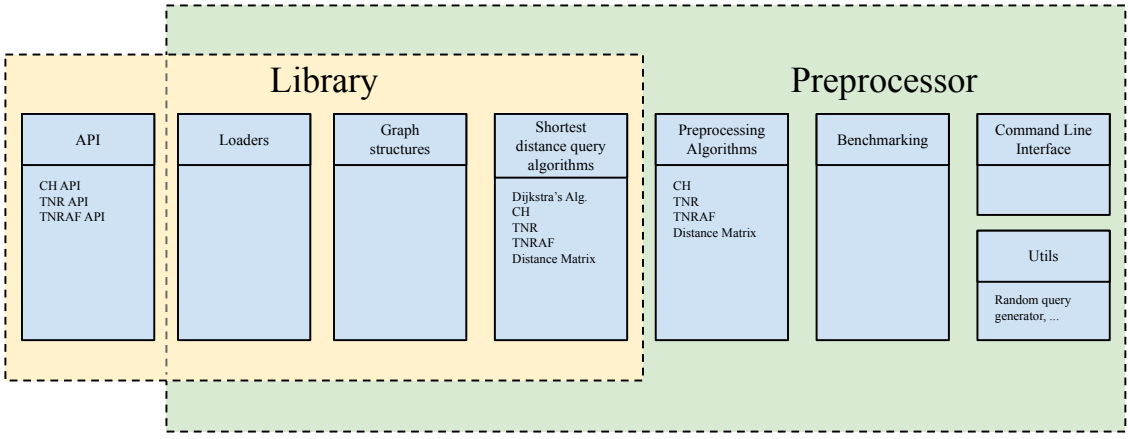


Figure 5.1: Schematic representation of the main components of our implementation.

Our implementation has a minimal number of dependencies. The preprocessor is self-contained and does not require any additional libraries apart from the standard C++ library. The library depends on *Java Native Interface* (JNI) [1] to connect the C++ code with the

[1] Java Native Interface on Wikipedia: <https://en.wikipedia.org/wiki/Java_Native_Interface>.

Java code. This is included in the *Java Development Kit*. We use *CMake* [2] to build the project. This combination allows the implementation to run on a variety of platforms. The implementation includes a documentation generated using Doxygen [3].

The final implementation can be downloaded from GitHub [4]. The commit corresponding to this thesis `cbacc39cf000f4c5af3ddb360d945a655600f446`. The project might be further developed in the future.

5.1 Preprocessor

The preprocessor is an executable application that provides a simple command-line interface for precomputing the structures required by the shortest distance algorithms of Contraction Hierarchies, Transit Node Routing and Transit Node Routing with Arc Flags. Additionally, the preprocessor allows the user to benchmark the data structures using a set of queries. The preprocessor will answer the queries and report the time needed to answer them along with the returned distances. Those distances can be used to verify the correctness of the structures.

The input of the preprocessor must be a directed weighted graph represented in a supported format (currently, there are two supported formats, one is our proprietary format called *XenGraph*, and the other format is called *DIMACS* and was used for the 9th DIMACS Implementation Challenge on Shortest Paths). The formats are described on the GitHub page of the project [5].

Section 5.1.1 describes what structures can be precomputed by the preprocessor, section 5.1.2 describes the benchmarking capabilities of the preprocessor. The command-line interface is briefly described in section 5.1.3 (it is described in more detail on GitHub).

5.1.1 Preprocessing Capabilities

The preprocessor allows the user to precompute structures for Contraction Hierarchies, Transit Node Routing and Transit Node Routing with Arc Flags. The preprocessing for each method is a straightforward implementation of the principles described in chapter 4.

For Contraction Hierarchies the nodes in the input graph are contracted in an order based on their edge difference and the number of their already contracted neighbors. During this process, new shortcut edges are added into the graph. In the end, all the data necessary for the query algorithm are output into a binary file with a `.ch` suffix that can be then used by the library to answer queries or by the preprocessor for benchmarking. Both original edges and shortcut edges need to bestored in the output file. Additionally, node ranks for all the nodes are stored since those are also used by the query algorithm.

[2] CMake is a family of tools that simplify the process or building C++ projects. More information on <https://cmake.org/>.

[3] Doxygen is a standard documentation generation tool for C++ projects. More information on <http://www.doxygen.nl/>.

[4] The implementation on GitHub: <https://github.com/aicenter/shortest-distances>

[5] Description of the formats used in our implementation can be found on <https://github.com/aicenter/shortest-distances/blob/master/FORMATS.md>.

When precomputing data structures for *Transit Node Routing*, the user can determine the transit node-set size k that will be used. As described in section 4.2.2, more transit nodes usually mean better performance, but at the cost of larger memory requirements. The input graph is then first preprocessed in the same way as for Contraction Hierarchies, but after that, the preprocessing continues by obtaining the transit node-set, which means finding out the k nodes with the highest rank assigned by the Contraction Hierarchies precomputation. A shortest distance table containing distances for all pairs of transit nodes is then computed before access nodes for all nodes and search spaces used for the locality filter can be computed. In the end, all the data required by the TNR query algorithm are again output into a binary file with a `.tnrg` suffix. This binary file contains all the data the file for Contraction Hierarchies would contain, and additionally contains the transit node-set, its distance table, the access nodes for all nodes, and the data for the locality filter.

For *Transit Node Routing with Arc Flags*, the user can again determine the transit node-set size. After that, we need to obtain all the data required by ordinary Transit Node Routing, but when finding access nodes, we additionally compute arc flags for them. In the end, all the data required by the TNRAF query algorithm are output into a binary file with a `.tgaf` suffix. This file contains all the data the file for ordinary Transit Node Routing would contain, and additionally, it also contains all the arc flags data.

The formats of the individual binary files used to store the data structures are described on the GitHub [6] page of the project. The binary format was chosen to allow fast loading of the structures when we want to answer queries.

5.1.2 Benchmarking Capabilities

The preprocessor application also allows the user to easily benchmark the precomputed data structures to either ensure the distances the query algorithms will return using those data structures are correct, or to evaluate the performance of the methods. The preprocessor allows the user to benchmark Contraction Hierarchies, Transit Node Routing, Transit Node Routing with Arc Flags, and additionally Dijkstra's Algorithm. Dijkstra's Algorithm serves as a baseline, we can use the distances computed by Dijkstra's Algorithm as ground truth, and we can compare the running times of the other methods with Dijkstra's Algorithm.

```
5
0 1
1 3
2 0
4 0
2 4
```

Figure 5.2: An example of a query set file that can be used to benchmark a method using the preprocessor. This query set contains 5 queries, each line represents one query (for example the first line corresponds to a query asking the shortest distance from node 0 to node 1).

[6] Description of the formats used in our implementation can be found on <https://github.com/aicenter/shortest-distances/blob/master/FORMATS.md>.

If we want to benchmark any of the four methods, we need to give the preprocessor the corresponding data structure (For example for Contraction Hierarchies it will be a file with the *.ch* suffix. For Dijkstra's Algorithm, the input is the original graph in the *XenGraph* format) and a set of queries that will be used for the benchmark. The queries must be in a plain text file that starts with a line that contains exactly one number q that denotes the number of queries in the set, this line is then followed by exactly q lines each representing one query. Each of those lines contains two integers separated by a space *"s g"* where s is the start node for the query and g is the goal node of the query. An example of such a file is given in figure 5.2. The preprocessor will run all the queries in the query set and report the complete time required to answer all the queries in seconds and the average time needed for one query in milliseconds. The user can also provide an optional output file argument with a file path where the preprocessor should output the computed distances. If this argument is provided, the preprocessor will output a simple plain text file that starts with a line that contains the file path to the query set used for the benchmark followed by q lines, each containing the result of one query. This output file can then be used to for example verify the correctness of the distances computed by the structures (for example using the `diff` tool in Linux or the `FC` tool in Windows).

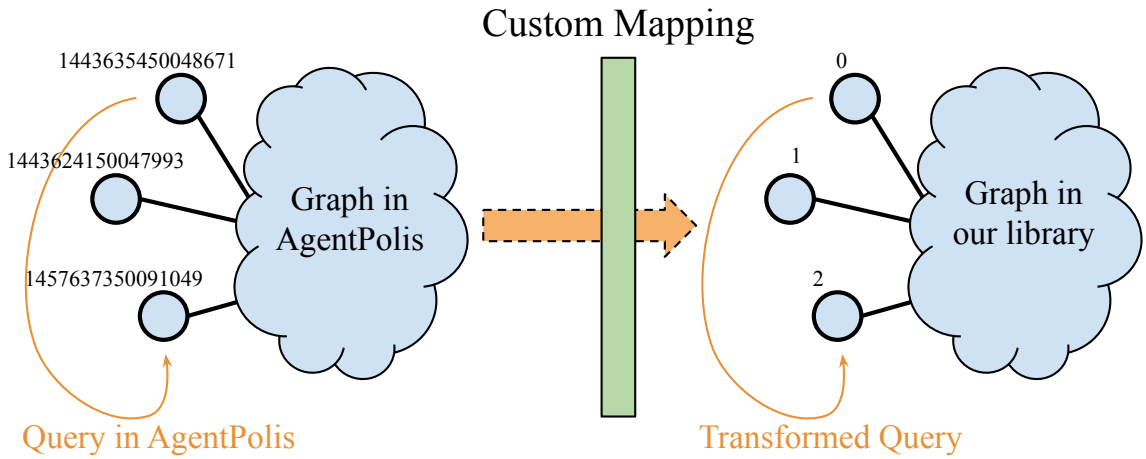


Figure 5.3: An example showing how a custom mapping for the node indices is used in AgentPolis. AgentPolis internally uses sixteen digits indices obtained from GeoJSON. Since our implementation internally works with identifiers in the range from 0 to $n-1$, we must use a custom mapping to transform the start and goal nodes of the queries from the indices used in AgentPolis to indices used in our library.

It can happen that the indices that will be internally used in our implementation do not correspond to the indices the user uses for his input graph. In this case, the preprocessor allows the user to specify a node mapping that will be used to transform indices in the query set to indices used internally in our implementation. The user can then use arbitrary positive integers (assuming they fit into the long long unsigned int data type) as indices. The user can, therefore, use a mapping to transfer those indices. An example showing how

the mapping is utilized for AgentPolis is given in figure 5.3. The mapping file format is again pretty simple:

- The file starts with a line in the format "*XID n*" where n denotes the number of nodes in the graph. *XID* is a fixed string constant that serves as a magic constant.
- The first line is followed by exactly n lines each only containing one positive number i. The j-th line represents the original ID of the *(j-2)*-th node. So the second line contains the original ID of the node with the ID *0* in our application. The third line contains the original ID of the node with the ID *1*, and so on up to the line *n+1* contains the original ID for the node with the ID *n-1* in our application.

When benchmarking, the user can decide if he wants to use a custom mapping or not. If a custom mapping is used, then the file containing the query set must contain the queries represented using the custom indices. For the AgentPolis framework, we use a mapping from indices obtained from the GeoJSON representation of the nodes to indices used by our implementation.

5.1.3 Interface

The preprocessor has a simple command-line interface (CLI) that allows the user to precompute the structures and then benchmark their performance. This CLI is supports two basic commands *create* and *benchmark*. After the command, the next argument must be the chosen method. Here, the user can choose between Contraction Hierarchies, Transit Node Routing, and Transit Node Routing with Arc Flags. When benchmarking, Dijkstra's Algorithm can also be used. The rest of the arguments differs based on the command and the selected method. We give one example to illustrate the CLI:

- `create tnraf xengraph slow 5000 graph.xeng outputgraph`.

This command precomputes the data structures required for Transit Node Routing with Arc Flags. The third argument denotes the format of the input graph. The fourth argument denotes the preprocessing mode. Some of the methods support multiple preprocessing modes that differ in their speed and memory requirements. The fifth argument denotes the desired size of the transit node-set, and the last two arguments are the input file path and the output file path respectively. A suffix based on the chosen method is automatically appended to the output file path. The obtained data structure can then be used for benchmarks using:

- `benchmark tnraf nomapping outputgraph.tgaf queryset.txt [results.txt]`.

In this case, the third argument indicates whether the query set contains node indices corresponding to the indices used inside the library, or if a custom mapping will be used. The fourth and fifth arguments are the path to the data structure and the query set that will be used for the benchmark. Using the last optional argument, the user can supply a file name into which the results of the individual queries will be stored. This makes it easier to check the correctness of the computed values. All the options that can be used along with more examples are described in more detail on GitHub [7].

[7] The GitHub project page: <https://github.com/aicenter/shortest-distances>

5.2 Library

The second component of our implementation is the *library*. This component is a shared library that provides an interface for loading the structures precomputed by the preprocessor and answering queries using those structures. The library can be used in other C++ projects and it also contains the code necessary for the usage of the library from *Java*. We describe the integration into Java in chapter 6. The library allows the user to use any of the three implemented methods (Contraction Hierarchies, Transit Node Routing, and Transit Node Routing with Arc Flags).

For each method, there is a special class called *QueryManagerAPI* (for example *TNRAF-QueryManagerAPI* for Transit Node Routing with Arc Flags) than loads the corresponding data structure from a binary file obtained from the preprocessor. The user can also specify a custom node mapping. This class can then be used to answer queries using the chosen shortest distance algorithm.

5.2.1 Interface

The library provides an interface in the form of three classes (each representing one supported method). The classes are the `CHDistanceQueryManagerAPI`, `TNRDistanceQueryManagerAPI` and `TNRAFDistanceQueryManagerAPI`. Each of those classes has exactly three functions. The first function is `initializeCH` (or `initializeTNR` and `initializeTNRAF` for the other methods) and it loads the data-structures required for the chosen shortest distance algorithm from a file precomputed by the preprocessor. The second function is `findDistance`, which answers queries. It must be called after the initialization function, and it uses the structures to obtain the shortest distance using the chosen method. The last function is `clearStructures`. This function frees all the memory required by the structures, and should, therefore, be called after the last query. When using the library from Java, this is required, as the garbage collector in Java is otherwise not able to free the memory allocated by the library. The interface is depicted in figure 5.4.

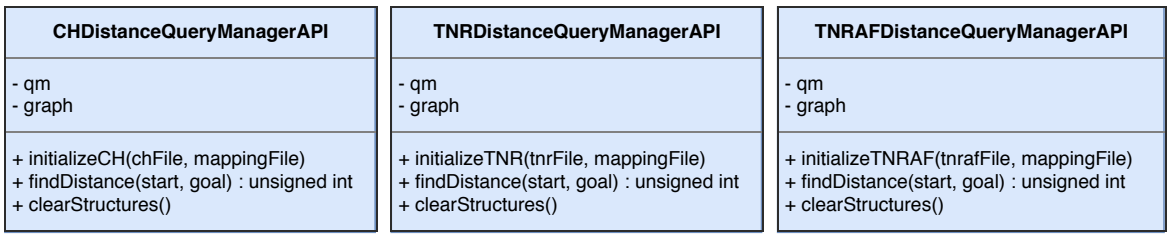


Figure 5.4: The library interface. The *qm* private variable stands for the corresponding query manager that will answer the queries using the chosen algorithm. The *graph* private variable contains the data-structure required for the chosen method and is used by the query manager. The initialization functions expect a mapping file, but the user can supply a dummy mapping that maps node indices to themselves in case he or she does not want to use a custom mapping.

Chapter 6

Integration

One of the main goals of this thesis was to allow an existing framework called *Agentpolis* to use the fast distance computation methods, especially Transit Node Routing with Arc Flags as it is the fastest variant. Agentpolis is a framework for modeling transportation systems. Since it is written in Java, we needed to integrate our implementation with Agentpolis in a way where it would be possible to call the C++ code from Java. To accomplish this, we have decided to use Java Native Interface (JNI) which is a framework that enables Java code running in an *Java Virtual Machine* (JVM) to call (and possibly also be called) native applications and libraries written in other languages. Since JNI requires a lot of wrapper code on both sides, we also used a software development tool called SWIG (Simplified Wrapper and Interface Generator) that simplifies the process. [1]

6.1 Using SWIG to Generate the Glue Code

SWIG uses proprietary interface files to determine which parts of the C++ code the user wants to be able to call from the application written in another language (in our case from Java). When those interface files are supplied to SWIG, it generates "glue code" necessary for the integration. In our case it generates a file called `shortestPathsInterface_wrap.cxx` on the C++ side, that needs to be included in the shared library. It contains the wrapper functions that will be called from the Java application. This file also includes the `jni.h` header which means that this file is the reason why JNI is enforced in the `CMakeLists.txt` file. On the Java side, we obtain multiple files.

For each QueryManagerAPI class (mentioned in section 5.2) we obtain one Java class that corresponds to the C++ class and provides the same functions. We also obtain a file called `shortestPathsJNI.java` that registers all the callable functions of our shared library in Java.

We are integrating the C++ code into Java, but the nice property of the SWIG interface files is that they make it fairly simple to also integrate our C++ code into another language. If there would be a need to integrate our implementation into another language in the future, we can just use the existing SWIG interface files without any changes to generate "glue code"

[1] SWIG is a software development tool to connect C++ code with other languages. More on <http://www.swig.org/>.

for another language, and with those files, the integration would then need very little work. This way we could integrate our code into any language supported by SWIG. The list of all languages supported by SWIG can be found on the homepage of the tool [2].

6.2 Travel Time Providers in Java

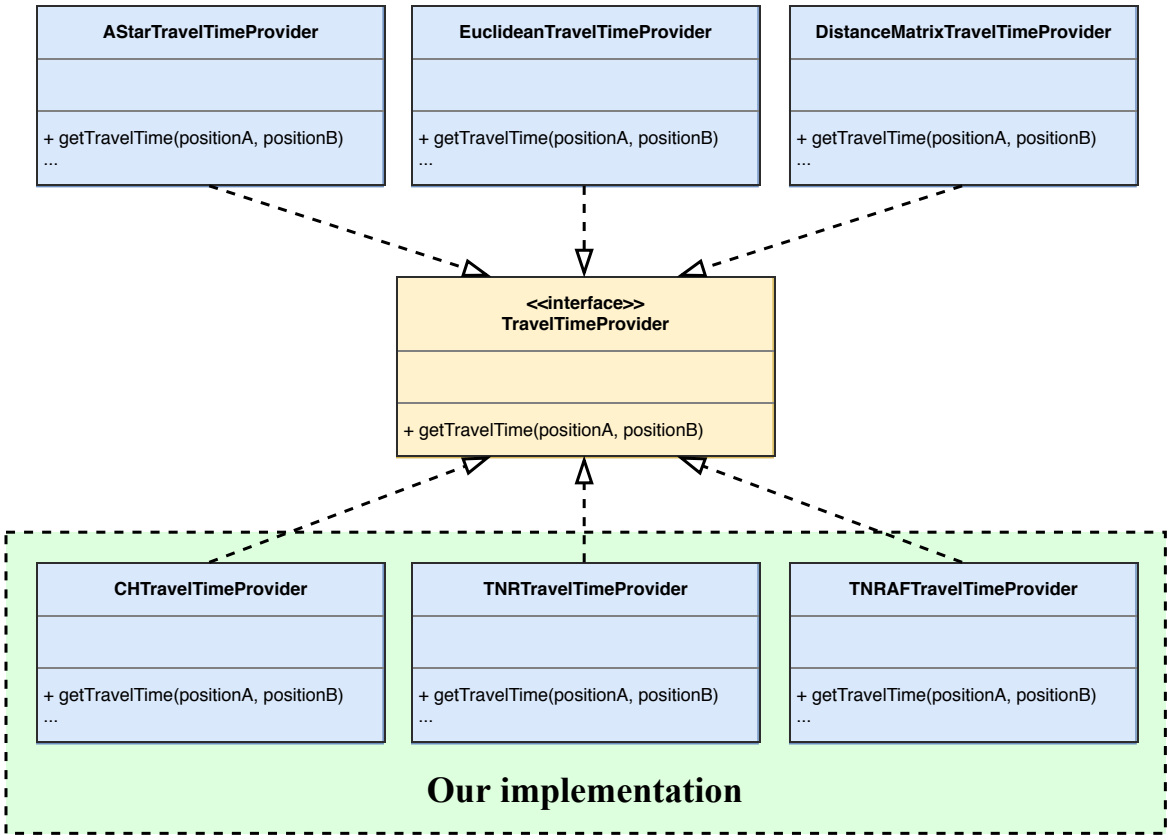


Figure 6.1: Travel Time Providers in AgentPolis. Each provider must implement the `TravelTimeProvider` interface. We have implemented three new providers.

The AgentPolis framework uses a component called the *Travel Time Provider* to compute the shortest distances between two locations in a road network. The Travel Time Provider is invoked each time any of the other components requires the shortest distance from one node to another node. Prior to our thesis, there were already three Travel Time Providers available in the framework. The first one was an *A* Travel Time Provider* that returned exact distances but was too slow for more complex scenarios. The second one was an *Euclidean Travel Time Provider* that approximated the shortest distances by the euclidean distance between the two nodes. The approximation was good enough for some scenarios but insufficiently precise for others. And finally, the third one was a *Distance Matrix Travel*

[2] A complete list of languages supported by the SWIG tool can be found on <http://www.swig.org/compat.html#SupportedLanguages>

Time Provider that used a precomputed complete Distance Matrix to return the distances. While this approach is obviously the fastest possible solution, the memory requirements of this provider are unfulfillable for large road networks.

We added three new providers into AgentPolis that can be now also used to compute the distances. We added a provider for each of the methods, so there is a Contraction Hierarchies provider, Transit Node Routing provider, and Transit Node Routing with Arc Flags provider. Those providers can load the corresponding data structures and then use them to answer queries. Fugre 6.1 shows our new travel time providers along with the already existing providers. We have also implemented parallelization of the providers. When initializing the provider, we create p instances of the *QueryManagerAPI* class where p is the amount of available threads on the system (acquired using `Runtime.getRuntime().availableProcessors()` in Java). Each query is then assigned to one of the QueryManagerAPI instances if there is one available, otherwise it waits until one of them finishes answering its current query, and then it is assigned. This parallelization scheme is shown in figure 6.2. From a parallelization point of view, this is an input data partitioning approach.

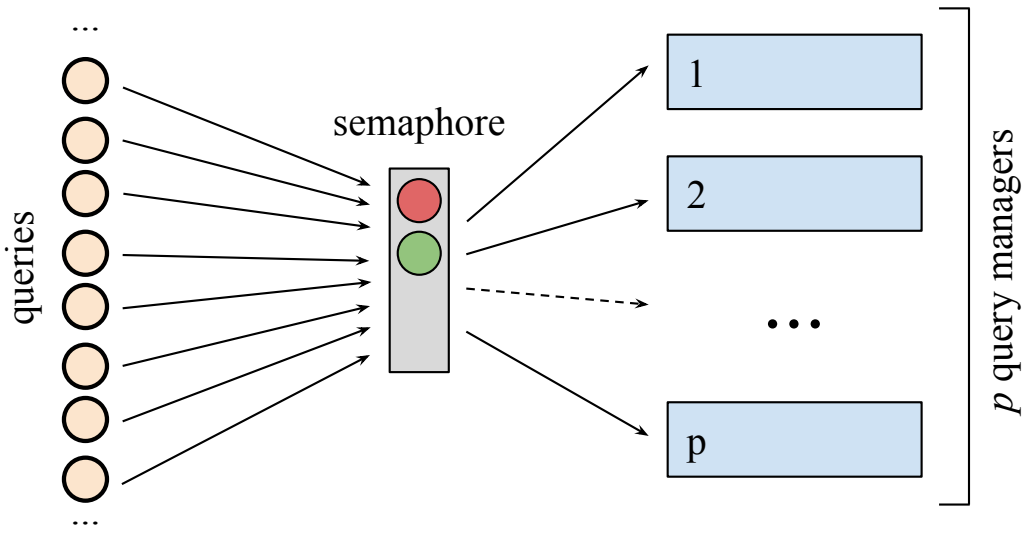


Figure 6.2: Parallelization scheme of the travel time providers. We create p instances of the corresponding QueryManagerAPI class. We then use a semaphore to ensure that at most p queries are being processed at any time.

AgentPolis provides a `config.cfg` file that contains file paths to various files required by the framework along with some settings. We have extended this config file so that the paths to the data structures required by our Travel Time Providers can be configured easily using a local config. The config is also used to set the file paths to the structures required for our new Travel Time Providers (the structures precomputed by the preprocessor).

6.3 Running Agentpolis with Our Library

To use our library in the Agentpolis framework, the user must take a few steps. The process is described in depth with examples on the GitHub page of the project. [3].

1. The user must compile the library for his architecture. This can be achieved using the prepared `CMakeLists.txt` that was tested on Linux using *gcc* (version 7.5.0) and on Windows using *MSVC* (version 16.5.4).

2. If the user does not have any files containing structures required for the library, he must compile the preprocessor and use it to precompute some structures depending on the method the user wants to use. This step can be skipped if the user already has some precomputed files for example downloaded from some other user.

3. The user must create a local config for AgentPolis (or adjust an existing one) to set the file paths to the file containing the data that will be used by the library.

4. AgentPolis must be able to find the library obtained in the first step on the user's system. Since the library is loaded using `System.loadLibrary("shortestPaths")`, we just need to ensure that the directory containing the library obtained in the first step is included in the `java.library.path` property when running AgentPolis. This can be achieved by either moving the compiled library into some directory contained in the `PATH` environment variable or adjusting the variable by adding the directory containing the library.

After those steps the new Travel Time Providers added during this thesis can be used in the existing scenarios in AgentPolis. Before this thesis, the chosen Travel Time Provider was selected directly in the code. It was necessary to build the project again each time the user wanted to change the Travel Time Provider. This did not cause any problems, as the chosen provider was not changed often. We have adjusted the existing config to allow easier choice of the desired provider. Currently, we can select the desired Travel Time Provider by only changing the local config. Figure 6.3 shows an example of a typical interaction of AgentPolis with our implementation.

[3] A step-by-step tutorial on how to use the library with Agentpolis: <https://github.com/aicenter/shortest-distances/blob/master/AMOD_README.md>

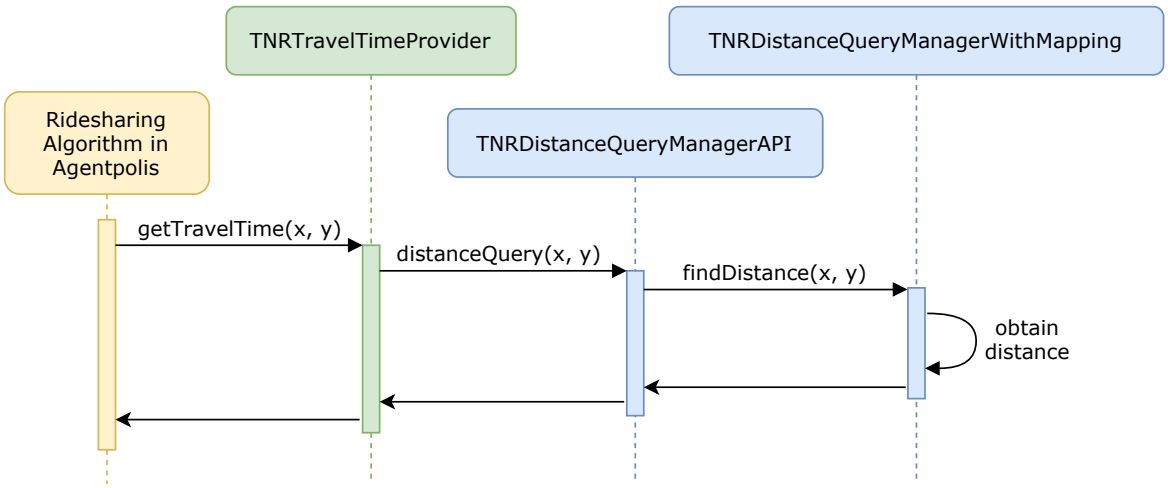


Figure 6.3: Diagram showing the typical interaction of AgentPolis with our implementation. The existing ridesharing algorithm in AgentPolis initiates a shortest distance query. Our Travel Time Provider on the Java side (depicted in green) forwards the query to the library. The library API (`TNRDistanceQueryManagerAPI` in this case) further forwards the query to the actual query manager on the library side, which then processes the query. The query manager transforms the node indices using the mapping and then evaluates the locality filter to decide whether the query is a local query. Then it obtains the shortest distance either using transit nodes or using the fallback algorithm for local queries. This result is then transferred to the ridesharing algorithm in AgentPolis. We have used Transit Node Routing for this example, but the process is similar for the other methods.

Chapter 7

Testing and Evaluation

In this chapter, we discuss the tests and benchmarks we performed with our implementation. We first describe how we ensured that our implementation gives the correct results in section 7.1 and then present the results of conducted performance benchmarks in section 7.2.

All tests were conducted on a machine equipped with a Ryzen 5 3600X processor paired with 32 GB of RAM running on 3200 Mhz. We used the Linux Mint 19.3 operating system (for the compilation we used *gcc* version 7.5.0). We have also verified that the implementation works correctly on Windows 10 when using the *MSVC* (Microsoft Visual C++) compiler (version 16.5.4).

7.1 Ensuring Correctness of Our Implementation

During the implementation part of this thesis, we needed to ensure that the implementation works correctly and returns the correct values for queries all the time. We have established various checks and tests that help us with this. We are using tests of our implementation alone to make sure the algorithms are implemented correctly, tests of the *Java* API to ensure the library can be used from *Java* applications and then tests in the *AgentPolis* framework that test whether everything is set up correctly to make the library work with *AgentPolis*.

7.1.1 Testing the Correctness of the Algorithms Implementation

During the implementation of all the methods, we needed to ensure that they return the correct shortest distances at any point during the implementation. To do this, we used *Dijkstra's Algorithm* [1]. Our code contains an implementation of this algorithm. We first manually verified that this implementation computes the correct distances for a small set of queries. From then on, we used this implementation to verify that the more complex methods in our implementation return the same distances as this reference Dijkstra's Algorithm implementation.

We prepared a set of 50 000 random queries and each time the implementation of some of the more complex methods changed, we verified that the new implementation of the complex method returns the same distances as the reference Dijkstra's Algorithm on this set of queries. When finalizing the implementation at the end of the development, we used an even larger set of 100 000 random queries for those tests.

7.1.2 Automatic Tests of the Java Interface

The next step in our process was to make sure that the library can be invoked from Java and that the distances returned in Java are correct. For this we have built a simple *Maven* [1] managed Java project. This project is also provided on GitHub [2]. This project contains two tests for each of the methods provided by the library interface (Contraction Hierarchies, Transit Node Routing, and Transit Node Routing with Arc Flags). The project contains the precomputed data structures as well as the correct distances. The tests then verify, that the data structures can be correctly loaded and that the distances computed by the library correspond to the precomputed values.

Those tests can be all run by executing `mvn test` or they can be executed individually using `mvn -Dtest=desiredTest test`. The tests are also run automatically when the user tries to package or install the Java project. If those tests finish successfully, then the Java Virtual Machine was able to find the library and it is working correctly.

7.1.3 Automatic Tests in the AgentPolis Framework

Being a large Maven managed Java project, AgentPolis contains a large number of tests that are run during each build to ensure all components are working correctly. Our goal was to add new tests that would verify that the library works correctly when used with AgentPolis.

We added a test for each of the methods provided by the library interface (Contraction Hierarchies, Transit Node Routing, and Transit Node Routing with Arc Flags). In each test, we use a set of 400 queries and compare the values returned by the library with values computed by the existing *A* Algorithm* implementation. If those tests pass, they ensure that when using the library to compute the shortest distances, the scenarios in AgentPolis will evaluate the same way as when using the A* Algorithm.

7.2 Evaluation of the Performance of Our Implementation

After we have ensured that the implementation works correctly, we needed to evaluate the benefit of using the implementation instead of some other methods. We first run a series of benchmarks of our *C++* implementation to evaluate the performance of various methods with various settings. We used three graphs obtained from real road networks for our experiments. The graphs were obtained by a transformation of the GeoJSON obtained from OpenStreetMap [3] to graphs usable in our implementation. The obtained graphs contained some bidirectional edges (they represent two-way roads). We transformed each such edge into two edges.

We compare the performance of our implementation with some baseline methods on each of the graphs in sections 7.2.1, 7.2.2 and 7.2.3. In sections 7.2.4 and 7.2.5 we then discuss how much the transit node-set size influences the performance and the memory requirements of Transit Node Routing and Transit Node Routing with Arc Flags using one of the graphs

[1] Maven is a software project management and comprehension tool. More on <https://maven.apache.org/>

[2] The GitHub project page: <https://github.com/aicenter/shortest-distances>

[3] OpenStreetMap is a collaborative mapping project. More on <https://www.openstreetmap.org/>.

as an example. All of the results presented in those sections were obtained using a query set containing 100 000 random queries. We made 20 runs of all the tests with the query set and present the average values. The exact code used for this evaluation can be found on the GitHub page of this project [4].

Since one of our goals was to improve the capabilities of the *AgentPolis* framework by integrating our implementation into it, we also evaluated the performance of a complex scenario in *AgentPolis* when using our library for the shortest distance computations. We present the results in subsection 7.2.6.

7.2.1 Comparison of Various Methods on the Graph of Prague

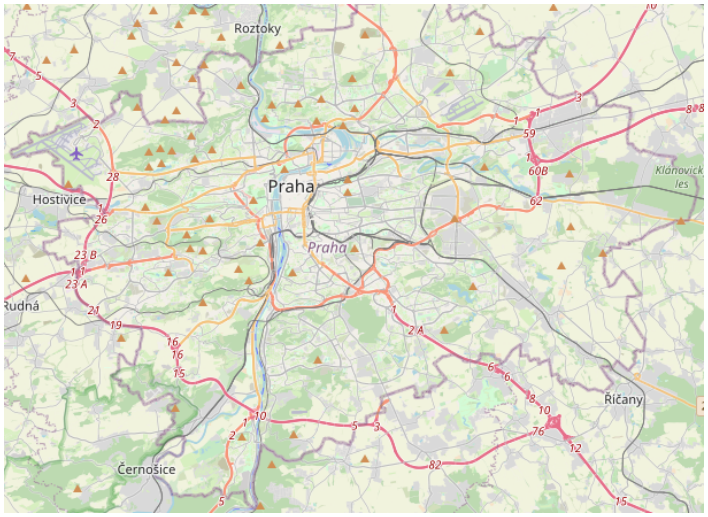


Figure 7.1: Geographical area corresponding to the graph of Prague.

The first graph we have used for our benchmark is a graph of *Prague*. This graph 28 686 nodes and 68 331 edges (some of those edges are bidirectional). The geographical area corresponding to this graph is shown in figure 7.1. We have evaluated five methods on this graph using a query set containing 100 000 random queries. Those methods were:

- Classic *Dijkstra's Algorithm*. For this algorithm, we used the original graph as input, but we simplified the graph before answering the queries, as it contained parallel edges and cycles.
- *Contraction Hierarchies* (CH).
- *Transit Node Routing* (TNR) with a transit node-set containing 2 000 nodes.
- *Transit Node Routing with Arc Flags* (TNRAF) also with a transit node-set containing 2 000 nodes.
- *Distance Matrix* (DM) computed from the original graph.

[4]The GitHub project page: <https://github.com/aicenter/shortest-distances>

We compare those five methods in terms of their time and memory requirements. Figure 7.2 shows the results. Both of the axes have a logarithmic scale in this case. On the x-axis we can see the memory needed for the data structure required for a given method while on the y-axis we can see the average time needed by the method for one query in microseconds.

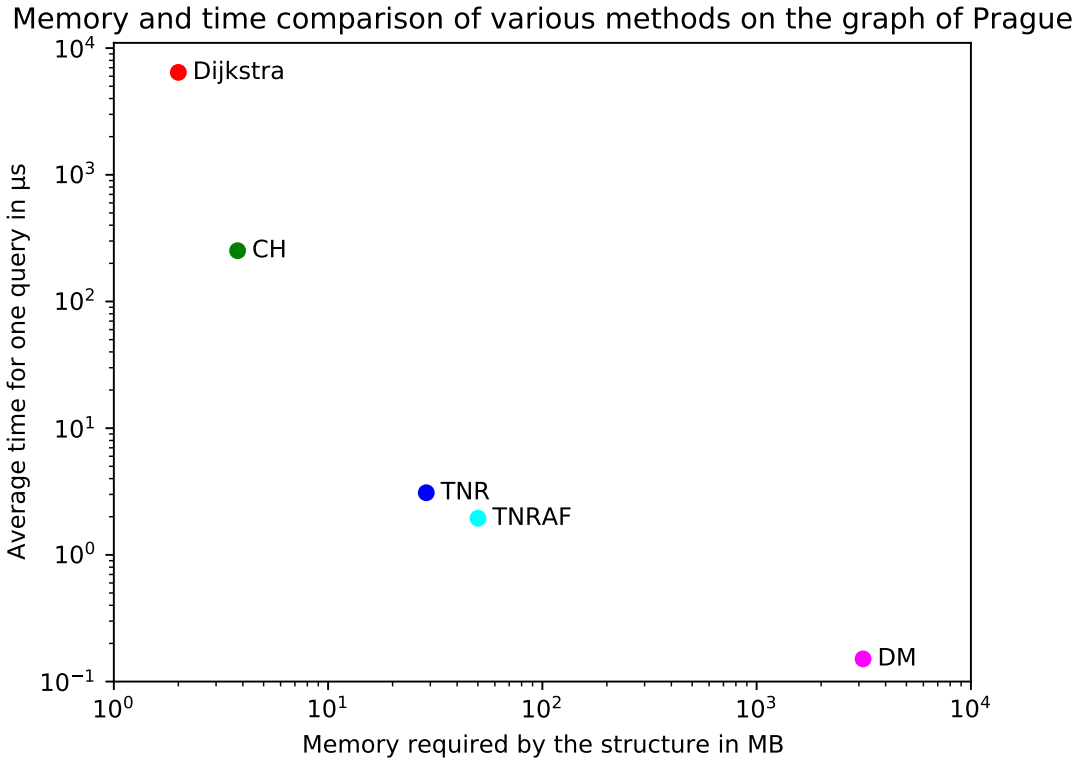


Figure 7.2: Comparison of the methods on the graph of Prague

The exact measurements are also presented in the table 7.1. From the values, we can see that Contraction Hierarchies were 25.55 times faster than Dijkstra's Algorithm, Transit Node Routing was 2 079.55 times faster than Dijkstra's Algorithm and Transit Node Routing with Arc Flags was 3 307.07 times faster than Dijkstra's Algorithm. We can also see that Transit Node Routing with Arc Flags needed 25.03 times the amount of memory of Dijkstra's Algorithm to accelerate the queries 3 307.07 times. On the other hand, when comparing Transit Node Routing with Arc Flags and Distance Matrix, the Distance Matrix needs 62.71 times the amount of memory of Transit Node Routing with Arc Flags, but only accelerates the queries 12.86 times (against TNRAF).

In section 4.1.4, we mentioned that in some cases, Contraction Hierarchies can have negative memory overhead. As we can see, this does not happen for our graph of Prague. This is caused by the fact that a lot of shortcut edges are added into the graph when preprocessing the graph of Prague, outweighing the space saved by representing bidirectional edges only once in Contraction Hierarchies data structure. If we wanted to use Bidirectional

Method	Time for one query in µs	Memory in MB
Dijkstra	6 431.995	2.00
CH	251.719	3.78
TNR	3.093	28.72
TNRAF	1.945	50.06
DM	0.151	3 139.06

Table 7.1: Measurements for the methods on the graph of Prague.

Dijkstra though, we would need to represent all the edges twice, so the representation for Bidirectional Dijkstra would need 4 MB of space, while Contraction Hierarchies already use a bidirectional approach and only need 3.78 MB of space. This means that Contraction Hierarchies have a negative memory overhead with regards to Bidirectional Dijkstra, while at the same time accelerating queries 25.55 times when Bidirectional Dijkstra only speeds up the queries up to 4 times [23].

We can see that all of thee evaluated methods are *cost optimal* in a sense that there is no method that would provide better query times than some other methods while needing less or the same amount of memory. The choice of the ideal method, therefore, depends on the specific use case. Some times the memory requirements of some of the methods might be too demanding. Other times fastest possible query times might be necessary and we might be left with no other option than the Distance Matrix approach. Those results however show, that for example Transit Node Routing with Arc Flags is not that far behind Distance Matrix in terms of performance, while being significantly less memory demanding.

7.2.2 Comparison of Various Methods on the Graph of Berlin

As our second graph for the benchmark, we have used a graph of Berlin that contains 44 698 nodes and 115 910 edges. The geographical area corresponding to this graph is depicted in figure 7.3. We have used the same five methods as in section 7.2.1, but for both Transit Node Routing and Transit Node Routing with Arc Flags we have used a transit node-set containing 5 000 nodes in this case.

Figure 7.4 shows the results. Both of the axes have a logarithmic scale. The exact measured values are reported in table 7.2.

Method	Time for one query in µs	Memory in MB
Dijkstra	10 807.605	3.27
CH	452.609	6.46
TNR	3.182	115.76
TNRAF	1.770	146.60
DM	0.177	7 621.43

Table 7.2: Measurements for the methods on the graph of Berlin.

In this case, Transit Node Routing was 3 396.72 times faster than Dijkstra's Algorithm, while needing 35.40 times the amount of memory. Transit Node Routing with Arc Flags was 6 106.14 times faster than Dijkstra's Algorithm while needing 44.83 times the amount

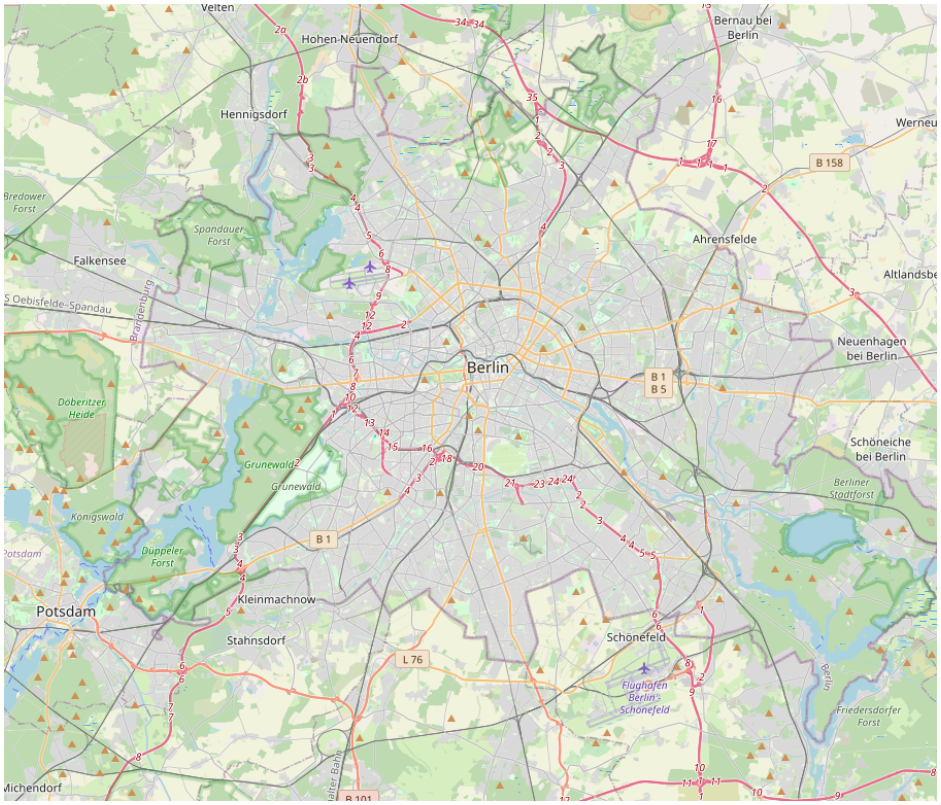


Figure 7.3: Geographical area corresponding to the graph of Berlin.

of memory. Note that in this case we are also using a larger transit node-set than in section 7.2.1 for both Transit Node Routing and Transit Node Routing with Arc Flags. We discuss the influence of the transit node-set size on performance and memory requirements in sections 7.2.4 and 7.2.5. The Distance Matrix is 10.03 times faster than Transit Node Routing with Arc Flags and needs 51.99 times the amount of memory.

The measured values also show, that in this case Transit Node Routing with Arc Flags needs 26.64 % of additional memory in comparison to Transit Node Routing. For the graph of Prague in section 7.2.1, Transit Node Routing with Arc Flags needed 74.30 % of additional memory. This indicates that for larger graphs (and also larger transit node-sets), the proportional difference between Transit Node Routing and Transit Node Routing with Arc Flags in terms of memory decreases.

We also tried using a transit node-set containing only 2 000 nodes for the graph of Berlin. Transit Node Routing with Arc Flags with this transit node-set was only 2 626 times faster than Dijkstra's Algorithm while needing 26.76 times the amount of memory. Our experiments indicate, that for graphs corresponding to road networks of real cities, transit node-sets containing around 10 % of nodes in the graph offer a great combination of performance and reasonable memory requirements. The number can differ based on the structure of a specific graph (city). Sections 7.2.4 and 7.2.5 show how the performances and memory requirements of Transit Node Routing and Transit Node Routing with Arc Flags change when changing the transit node-set size using the graph of Prague as an example.

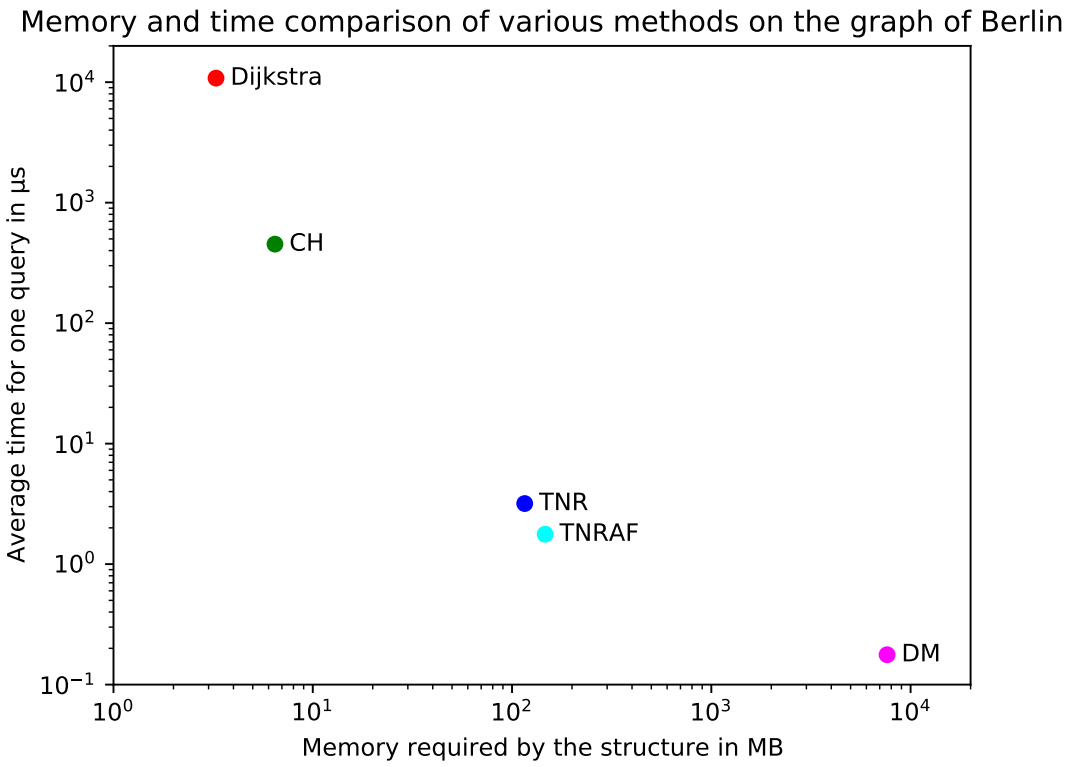


Figure 7.4: Comparison of the methods on the graph of Berlin

7.2.3 Comparison of Various Methods on the Graph of Southwest Bohemia

For our third benchmark, we decided to use a graph containing a larger part of the Czech Republic than just a single city. We chose a rectangular cutout bounded by Prague in one corner and Pilsen in the other corner. This graph contains 70 781 nodes and 173 122 edges. The geographical area corresponding to the graph is shown in figure 7.5. We have used the same five methods as in section 7.2.1, but for both Transit Node Routing and Transit Node Routing with Arc Flags we used a transit node-set size of 7 000.

The results are shown in figure 7.6. Both of the axes have a logarithmic scale. The exact measured values are presented in table 7.2.

In this case, Transit Node Routing with Arc Flags is 10 720.79 times faster than Dijkstra's Algorithm while needing 51.87 times the amount of memory. The Distance Matrix is 8.03 times faster than Transit Node Routing with Arc Flags while needing 74.28 times the amount of memory.

One interesting thing is that we also tried to use 5 000 transit nodes for Transit Node Routing with Arc Flags on the graph of Southwest Bohemia, and we achieved an average query time of 1.9 µs and the structure required 177.89 MB of memory. This is interesting, because in section 7.2.2, we used 5 000 transit nodes and the average query was 6 106.14

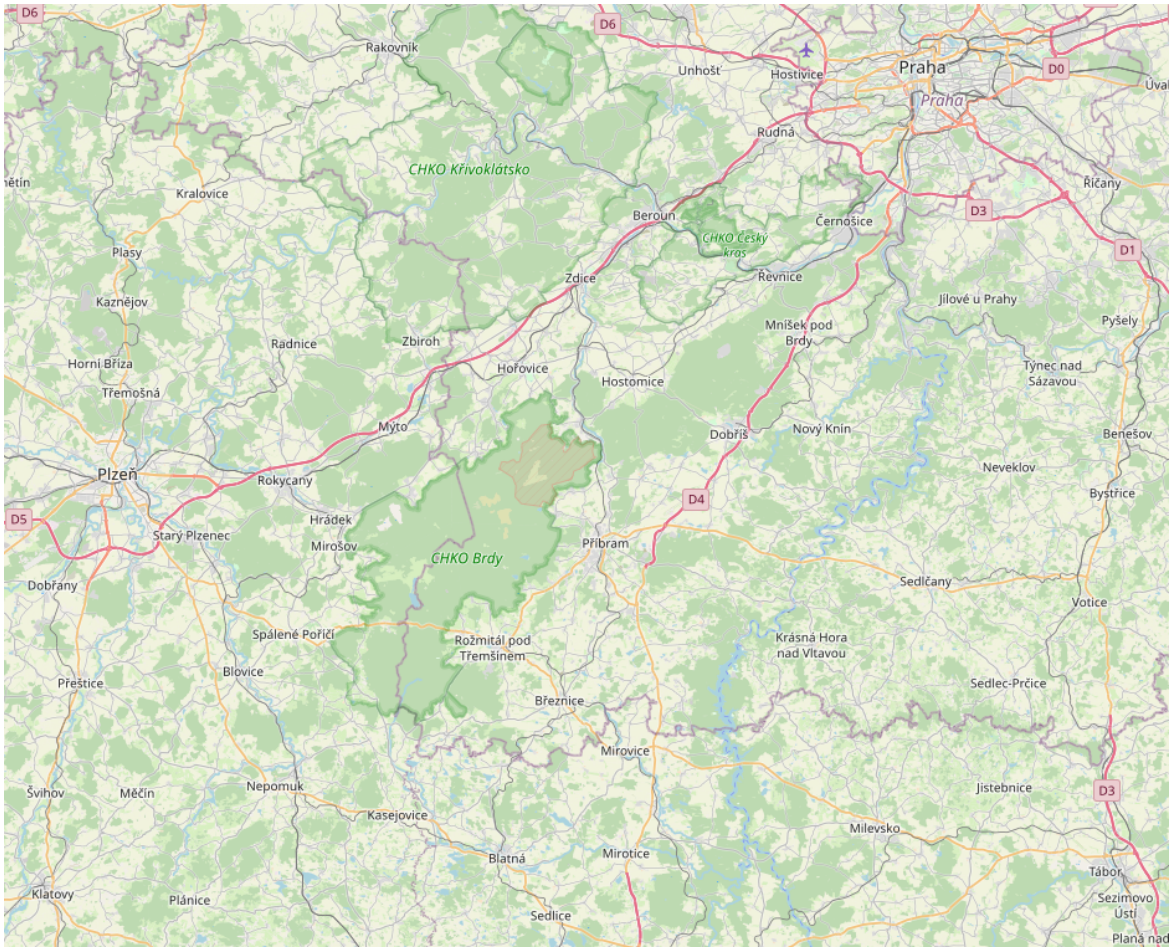


Figure 7.5: Geographical area corresponding to the graph of Southwest Bohemia.

Method	Time for one query in µs	Memory in MB
Dijkstra	16 781.984	4.96
CH	355.226	8.38
TNR	2.623	214.81
TNRAF	1.565	257.29
DM	0.195	19 111.44

Table 7.3: Measurements for the methods on the graph of Southwest Bohemia.

times faster than in the case of Dijkstra's Algorithm while needing 44.83 times the amount of memory. Here on a larger graph with the exact same settings (same number of transit nodes), the average query is 8 804.82 times faster than in the case of Dijkstra's Algorithm while only needing 35.86 times the amount of memory. This means that on a larger graph, we achieve better speedup with a proportionally lower memory overhead. This is caused by the fact that graphs that contain multiple cities and connections between them have a better structure that the algorithms can exploit.

Graphs that contain multiple cities have a more significant road hierarchy. For all queries

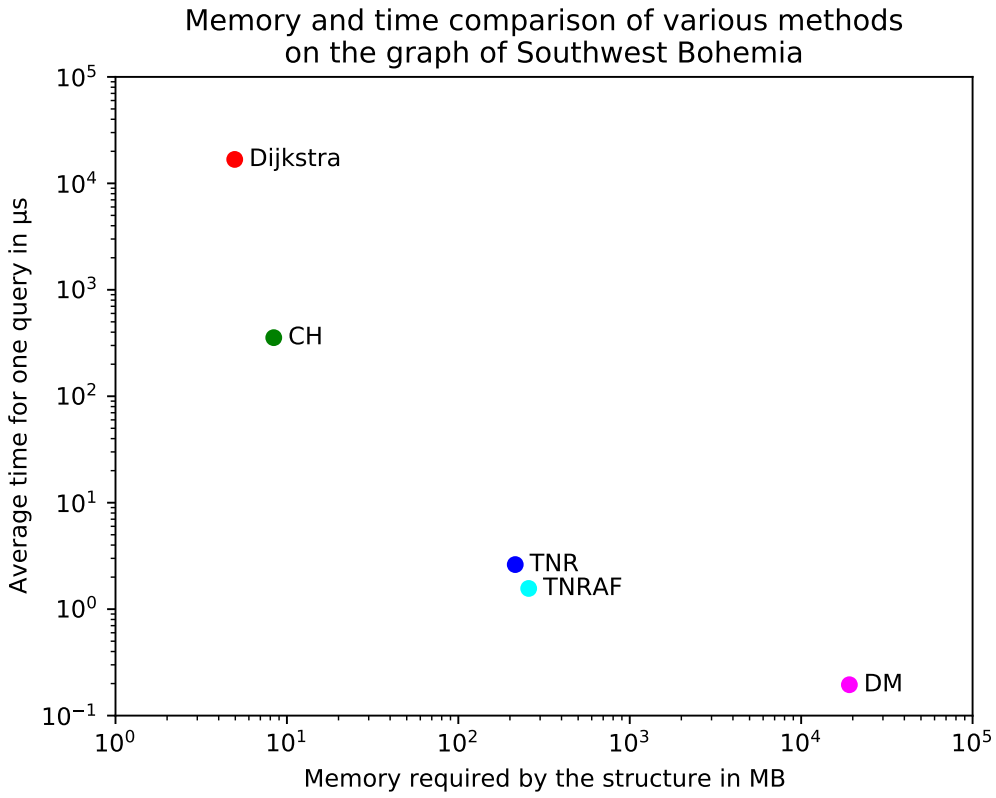


Figure 7.6: Comparison of the methods on the graph of Southwest Bohemia

going from one city to another, it is very common that we enter some highway behind our start city and leave this highway shortly before our goal city. For example in the graph of Southwest Bohemia, every time we want to travel from any location in Prague to any location in Pilsen, we will take the D5 highway. Transit Node Routing can identify this structure and will choose the access points to the highway as transit nodes. Since the amount of those access points is usually not that high, the average number of access nodes in the Transit Node Routing data structure will be fairly low. This is reflected in better query times and also lower memory requirements, as fewer access nodes need to be stored. For graphs only containing one large city, the road hierarchy is usually not that evident. When going from one corner of Prague to the opposite corner, the optimal path often depends on the exact start and goal locations. In such cases there must be different access nodes for different start and goal locations, which will be reflected in slightly worse performance and memory efficiency.

This shows, that the performance does not only depend on the size of the graph and the size of the transit node-set, but also on the structure of the graph. It might happen that on larger graphs Transit Node Routing (with or without Arc Flags) will achieve better results with a smaller transit node-set than on some smaller graphs, due to the structure of the graphs. It is, therefore, a good idea to try multiple transit node-set sizes for the given graph and then choose to option with the desired performance.

7.2.4 Comparison of Various Transit Node-Set Sizes for Transit Node Routing

One useful property of Transit Node Routing is the fact that we can control the performance by changing the size of the transit node-set. A larger transit node-set usually means better performance for the price of bigger memory overhead. When we have more transit nodes, the average number of access nodes for each node should decrease, accelerating the queries as fewer combinations of access nodes need to be tried. On the other hand, we need to store the full distance matrix for the transit node-set in memory, which increases the memory requirements.

To see how much the size of the transit node-set influences both the performance and the memory requirements, we tried 6 different transit node-set sizes on the graph of Prague. We tried 200, 500, 1 000, 2 000, 5 000, and 10 000 transit nodes. Figure 7.7 shows the results. Again, both of the axes have a logarithmic scale. The obtained values can then be seen in table 7.4.

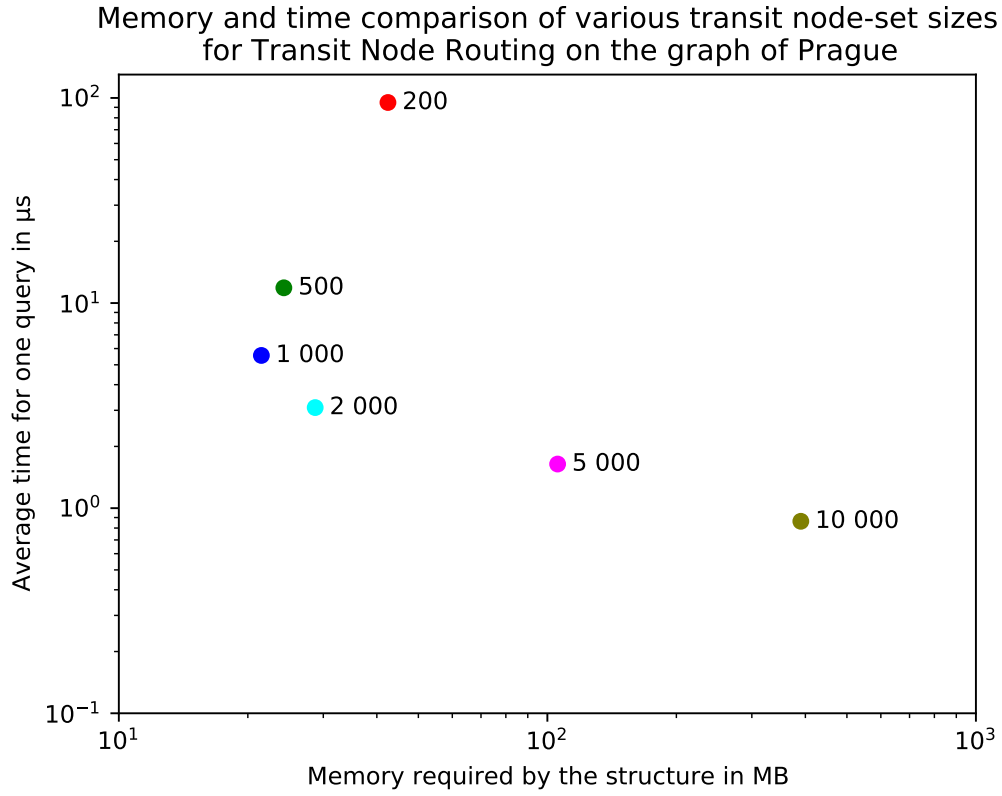


Figure 7.7: Comparison of various transit node-set sizes for TNR on the graph of Prague.

From the values, we can see, that the performance truly increases when the size of the transit node-set also increases. For the memory requirements, we can see that for example when using only 200 transit nodes, the memory requirements are larger than when using 500 transit nodes. This is caused by the fact, that the average number of access nodes for

Transit node-set size	Time for one query in µs	Memory in MB
200	95.021	42.46
500	11.867	24.27
1 000	5.554	21.51
2 000	3.093	28.72
5 000	1.641	105.66
10 000	0.863	390.33

Table 7.4: Measurements for various transit node-set sizes for TNR on the graph of Prague.

nodes in the graph is higher since transit nodes are on average farther away from the nodes. Also, the search spaces that need to be kept for the locality filter will be fairly large and therefore will require a significant amount of memory as well. Even though the distance matrix required for 200 transit nodes is small (needs only 156.25 KB), the memory needed to store the access nodes and search spaces outweigh the memory needed for the transit node-set distance table in this case. When we choose larger transit node-sets, the memory needed for the transit node-set distance table starts to outweigh the memory needed for other components of the data structure and has the biggest influence on the overall size of the structure (in case of 5 000 transit nodes, the distance table takes 95.37 MB out of the total 105.66 MB, while in the case of 10 000 transit nodes it takes 381.47 MB out of the total 390.33 MB).

This shows that when choosing the optimal transit node-set size, we should take multiple aspects into consideration. It usually does not make sense to choose too small transit node-sets, as they provide worse performance than larger transit node-sets while also needing more memory. On the other hand, too large transit node-sets bring us closer to the full distance matrix approach, as the memory requirements increase quadratically with the increase of the transit node-set size.

7.2.5 Comparison of Various Transit Node-Set Sizes for Transit Node Routing with Arc Flags

For Transit Node Routing with Arc Flags, we can alter the size of the transit node-set in the same way as for Transit Node Routing. We again tried 6 different transit node sizes on the graph of Prague. The results can be observed in figure 7.8. Both axes of the table have a logarithmic scale. The exact measured values are reported in table 7.5.

Transit node-set size	Time for one query in µs	Memory in MB
200	90.970	97.24
500	8.513	60.63
1 000	3.563	50.42
2 000	1.945	50.06
5 000	1.114	119.39
10 000	0.716	399.47

Table 7.5: Measurements for various transit node-set sizes for TNRAF on the graph of Prague.

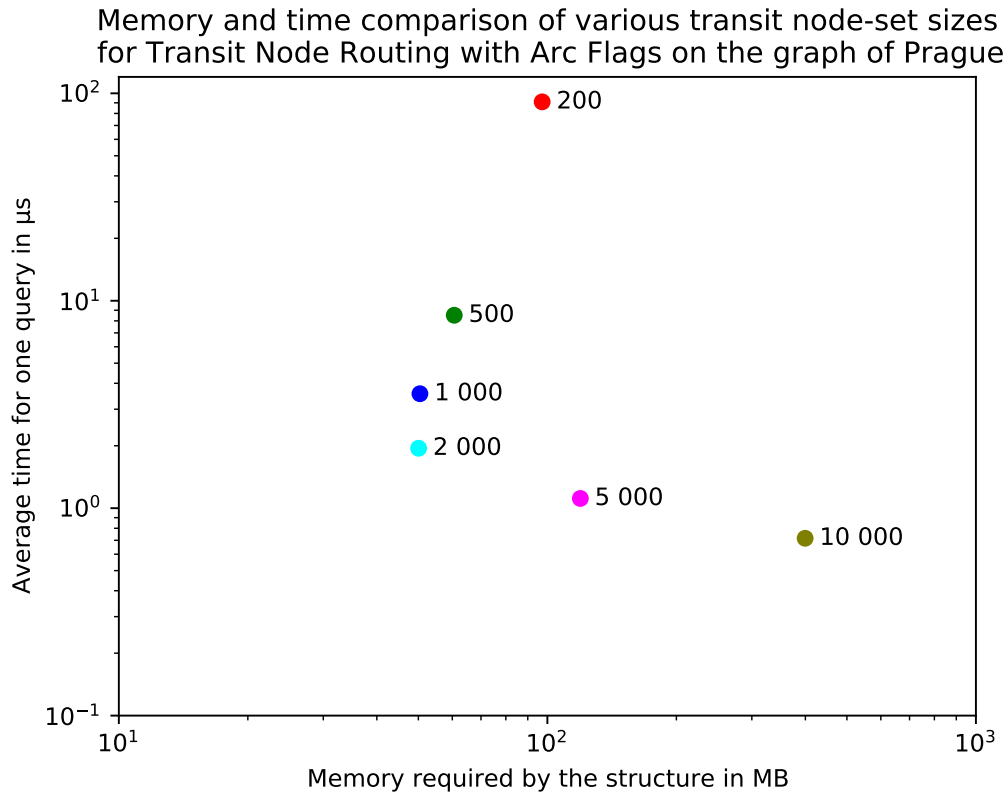


Figure 7.8: Comparison of various transit node-set sizes for TNRAF on the graph of Prague.

The additional memory overhead in Transit Node Routing with Arc Flags is caused by the fact that for each access node, we also need to store its Arc Flags. In our implementation, those Arc Flags take additional 4 Bytes for each access node. We also need to store regions for all nodes in the graph, but those data do not change when changing the transit node-set size.

The measured values show that for smaller transit node-sets, the additional memory overhead caused by the Arc Flags has a significant influence on the total memory required for the data structure. For larger transit node-set sizes, the influence of the Arc Flags becomes less noticeable, as the memory needed for the transit node-set distance table starts to greatly outweigh the memory required for the other components of the data structure. For example for 10 000 transit nodes, Arc Flags only require an additional 9.14 MB of memory when compared with the memory requirements of the data structure needed for Transit Node Routing without Arc Flags. This is only 2.28 % out of the total 399.47 MB needed for the data structure. Due to this, we recommend to always use Transit Node Routing with Arc Flags when using larger transit node-sets. When using smaller transit node-sets, it could make sense to only use Transit Node Routing without Arc Flags to decrease the memory requirements, as in those cases the data for Arc Flags represent a bigger portion of the overall data structure size.

7.2.6 Benefit of Our Implementation for the AgentPolis Framework

To evaluate whether the library will be beneficial for the AgentPolis framework, we used an existing complex scenario in AgentPolis. The scenario deals with an online ride-sharing problem. The problem is as follows. There is a large fleet of vehicles (for example from a taxi service) scattered over Prague and a large number of customers that want to get from their location to some other location. The problem changes in time, new customers appear, meaning the optimal plan for some set of customers might stop being optimal when considering the new customers. The goal is to minimize the waiting time of the customers.

An algorithm for solving the ride-sharing problem is already implemented in AgentPolis and its principle is not important for this thesis. What is important, though, is the fact that the algorithm needs to compute a large amount of shortest distances. The algorithm needs to know the distances from all vehicles to all the customers and also distances from all the goal locations of customers to the start locations of all the other customers.

We can simulate various time intervals. We first started with a short time interval of five minutes and compared the time needed for the whole simulation when using the existing A* implementation to compute the travel times, the time needed when using Transit Node Routing with Arc Flags, and the time needed when using Distance Matrix. When using A*, the simulation needed about 4 300 seconds on average. When using Transit Node Routing with Arc Flags for the travel times computation, the simulation only took about 26 seconds on average. With Distance Matrix, the simulation only took about 17 seconds on average. The difference between Transit Node Routing with Arc Flags and Distance Matrix is not that significant for such a short simulation, because there are other components of the ride-sharing algorithm that outweigh the time needed for the travel time computation in this case.

Travel Time Provider	5 minutes simulation	90 minutes simulation
A*	4 300	not measured
TNRAF	26	3 200
Distance Matrix	17	520

Table 7.6: Average times needed for the simulation with different Travel Time Providers. All times are in seconds.

We have, therefore, also tried a more realistic time interval of 90 minutes. Similar time intervals are also used in practice for real scenarios in AgentPolis. The longer the simulation is, the more complex the problem becomes, because new customers arrive every once in a while, but the already present customers might not be transported before new customers appear, which complicates the process of finding the optimal plan. Because of this the time needed for a longer simulation is not linearly proportional to the increase of the simulated time interval. For our 90 minutes time interval, the simulation took 3 200 seconds on average with Transit Node Routing with Arc Flags. When using Distance Matrix to compute travel times, the whole simulation took about 520 seconds on average. If we wanted to use A*, we would have to wait several days for the simulation to finish (based on times obtained with the 5 minutes time interval). This means, that using our library, we got from days in the case of A* to under an hour with our library. With our library the whole 90 minutes simulation takes 6.15 times longer than with Distance Matrix. The speedup of the whole

simulation achieved by using the Distance Matrix is smaller than the speedup measured in section 7.2.1 because there are also other components of the ride-sharing algorithm than travel time computation and those are not influenced when the travel time providers change. The times needed for the simulation are presented in table 7.6. During the 90 minutes long simulation, the Travel Time Provider answers over 1.85 billion travel time queries needed by the ride-sharing algorithm. Note that the Transit Node Routing with Arc Flags travel time provider already uses parallelization as described in section 6.2. The time needed for the simulation without the parallelization would be higher.

Chapter 8

Conclusion

Computing the shortest distance between two nodes in a graph is a common problem in computer science. In a lot of the cases, multiple queries have to be answered while the graph does not change between the queries. For such scenarios, it makes sense to precompute some auxiliary structures that allow us to answer subsequent queries faster. Since in many of the use cases the shortest distance computation takes a significant portion of the whole computation, speeding up the component responsible for the shortest distance computation results in a significant speedup of the whole application.

This thesis had two main goals. One main goal was to implement one such method that makes use of precomputed structures to answer queries faster. This method is called *Transit Node Routing with Arc Flags*. The second main goal of this thesis was to integrate the implementation of this method with an existing framework. A large number of applications that could benefit from faster shortest distance computation component along with the challenge of implementing a method based on the state-of-the-art approaches and then integrating it into an existing project was the main motivation for this thesis.

We have successfully implemented Transit Node Routing with Arc Flags based on Contraction Hierarchies. One result of this thesis is a C++ library that can compute the shortest distances using three different methods. Those methods are Contraction Hierarchies, Transit Node Routing, and Transit Node Routing with Arc Flags. This library is now available on GitHub [1] and can be used by other researches and developers in the future. We have also successfully integrated our implementation into the AgentPolis framework. This integration allows the framework written in Java to use our library written in C++ as a component for computing the shortest distances.

The implementation was extensively tested to ensure it provides correct results. Automatic tests were added to AgentPolis. The performance of the implementation was measured and the results show that the implementation provides significant speedup when compared with Dijkstra's Algorithm while having a reasonable memory overhead. On the largest tested graph, our implementation of Transit Node Routing with Arc Flags is over 10 000 times faster than Dijkstra's Algorithm while only needing about 50 times the amount of memory. The average query time for Transit Node Routing with Arc Flags on the graph of Prague is 1.95 microseconds, which makes the implementation usable even in the AgentPolis

[1] The implementation on GitHub: <https://github.com/aicenter/shortest-distances>

framework where the scenarios often require to answer billions of shortest distance queries in a reasonable time.

Future work could expand the library even further. The referenced literature mentions various improvements that could improve the performance of our implementation. This includes different locality filters for Transit Node Routing (both with and without Arc Flags) [22], other priority computation strategies for the contraction process in Contraction Hierarchies [8] or other graph clustering approach for the Arc Flags [13]. The library could be also expanded by adding more methods such as Hub Labels that provide even better query times than Transit Node Routing with Arc Flags for the price of bigger memory requirements.

References

[1] Thomas H. Cormen et al. "Section 24.3: Dijkstra's algorithm". In: *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009, pp. 658–664. ISBN: 0262033844.

[2] Sneha Sawlani. "Explaining the Performance of Bidirectional Dijkstra and A* on Road Networks". Master's Thesis. 2017. URL: <https://digitalcommons.du.edu/etd/1303>.

[3] Daniel Delling et al. "Engineering Route Planning Algorithms". In: *Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 117–139. ISBN: 9783642020933. URL: <https://doi.org/10.1007/978-3-642-02094-0_7>.

[4] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Jan. 2009.

[5] Peter Sanders and Dominik Schultes. "Engineering Highway Hierarchies". In: *ACM J. Exp. Algorithms* 17 (Sept. 2006), pp. 804–816. DOI: 10.1007/11841036_71.

[6] Peter Sanders and Dominik Schultes. "Highway Hierarchies Hasten Exact Shortest Path Queries". In: *Algorithms – ESA 2005*. Ed. by Gerth Stølting Brodal and Stefano Leonardi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 568–579. ISBN: 978-3-540-31951-1.

[7] Peter Sanders and Dominik Schultes. "Robust, almost constant time shortest-path queries in road networks". In: *The Shortest Path Problem* (July 2009). DOI: 10.1090/dimacs/074/08.

[8] 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.

[9] Robert Geisberger et al. "Exact Routing in Large Road Networks Using Contraction Hierarchies". In: *Transportation Science* 46 (Aug. 2012), pp. 388–404. DOI: 10.2307/23263550.

[10] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. *Customizable Contraction Hierarchies*. 2014. arXiv: 1402.0402 [cs.DS].

[11] Holger Bast et al. "In Transit to Constant Time Shortest-Path Queries in Road Networks". In: *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. New Orleans, Louisiana: Society for Industrial and Applied Mathematics, 2007, pp. 46–59.

[12] Hannah Bast et al. “TRANSIT: Ultrafast Shortest-Path Queries with Linear-Time Preprocessing”. In: *9th DIMACS Implementation Challenge — Shortest Path* (Jan. 2006).

[13] Moritz Hilger et al. “Fast Point-to-Point Shortest Path Computations with Arc-Flags”. In: *9th DIMACS Implementation Challenge — Shortest Path*. 2006.

[14] Gianlorenzo D’Angelo, Daniele Frigioni, and Camillo Vitale. “Dynamic Arc-Flags in Road Networks”. In: *Experimental Algorithms*. Ed. by Panos M. Pardalos and Steffen Rebennack. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 88–99. ISBN: 978-3-642-20662-7.

[15] Daniel et al. Delling. “Hub Label Compression”. In: *Experimental Algorithms*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 18–29. ISBN: 978-3-642-38527-8.

[16] Andrew V. Goldberg. “The Hub Labeling Algorithm”. In: *Experimental Algorithms*. Ed. by Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 4–4. ISBN: 978-3-642-38527-8.

[17] Daniel Delling et al. “Hub Labels: Theory and Practice”. In: *Proceedings of the 13th International Symposium on Experimental Algorithms - Volume 8504*. Berlin, Heidelberg: Springer-Verlag, 2014, pp. 259–270. ISBN: 9783319079585. DOI: 10.1007/978-3-319-07959-2_22. URL: <https://doi.org/10.1007/978-3-319-07959-2_22>.

[18] Reinhard Bauer et al. “Combining Hierarchical and Goal-Directed Speed-up Techniques for Dijkstra’s Algorithm”. In: *J. Exp. Algorithmics* 15 (Mar. 2010). ISSN: 1084-6654. DOI: 10.1145/1671970.1671976. URL: <https://doi.org/10.1145/1671970.1671976>.

[19] Hannah Bast et al. “Route Planning in Transportation Networks”. In: vol. 9220. Nov. 2016, pp. 19–80. ISBN: 978-3-319-49486-9. DOI: 10.1007/978-3-319-49487-6_2.

[20] Sebastian Knopp et al. “Computing Many-to-Many Shortest Paths Using Highway Hierarchies”. In: *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. New Orleans, Louisiana: Society for Industrial and Applied Mathematics, 2007, pp. 36–45.

[21] Robert Geisberger. “Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks”. Diploma Thesis. July 2008.

[22] Julian Arz, Dennis Luxen, and Peter Sanders. “Transit Node Routing Reconsidered”. In: *SEA*. Vol. 7933. Feb. 2013. DOI: 10.1007/978-3-642-38527-8_7.

[23] Rolf Möhring et al. “Partitioning Graphs to Speed Up Dijkstra’s Algorithm”. In: *Lecture Notes in Computer Science* 3503 (May 2005), pp. 189–202. DOI: 10.1007/11427186_18.

Appendix A

List of Used Abbreviations

CH Contraction Hierarchies

CLI Command-Line Interface

JNI Java Native Interface

JVM Java Virtual Machine

TNR Transit Node Routing

TNRAF Transit Node Routing with Arc Flags

Appendix B

Contents of the Attached CD

- `readme.txt` ..... A brief description of the individual contents of the CD
- `shortest-distances` ..... Contains the whole implementation
  - `src` ..... The source codes of the implementation.
  - `doc` ..... The project documentation generated using Doxygen
  - `thesisTestsData` ..... All the input data used for the tests present in this thesis
  - `CMakeLists.txt` ..... A script that allows simple building of the implementation
- `thesis`
  - `cvachmic_thesis.pdf` ..... This thesis in the PDF format
  - `src` ..... Source files for this thesis in the XeLaTeX format