



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



Institut für
Technische Informatik und
Kommunikationsnetze

Easy Breathe: Health-Optimal Routing in Urban Areas

Semester Thesis

Ivo de Concini

December 23rd 2014

Advisors: Prof. Dr. Lothar Thiele, David Hasenfratz
Computer Engineering Group, ETH Zürich

Abstract

Air pollution, especially particulate matter (PM), has very serious effects on human health. According to the World Health Organization, diseases caused by PM reduce the average life expectancy in Europe by 9 months. This is a significant number, especially considering that 80% of the European population lives in cities with levels of PM exceeding the levels considered safe by the WHO Air Quality Guidelines [1].

In this project we propose a smart phone application, which enables inhabitants of the city of Zurich to choose health-optimal paths between arbitrary locations in the city. We show in this thesis that by using health optimal paths, the exposure to particulate matter can be reduced by 6.5% on average. The paths are computed using the high resolution urban air pollution maps obtained by the OpenSense project [2].

The application does not rely on server infrastructure for the computation of the optimal paths: It uses kd-tree based nearest neighbour search along with a bidirectional shortest path search using a priority queue based variant of Dijkstra's algorithm to compute the health-optimal paths on the limited smart phone hardware.

We tested the application on a real smart phone, achieving an average computation time of 170ms for a health-optimal path.

Contents

Contents	iii
1 Introduction	1
1.1 Contributions	1
1.2 Related Work	2
1.3 Overview	2
2 Nearest Neighbour Search	5
2.1 KD-Tree	6
3 Optimal Path Problem	9
3.1 Dijkstra's Algorithm	10
3.2 Speeding up Dijkstra's Algorithm: Priority Queue	12
3.3 Bidirectional Search	13
4 Graph Implementation	17
4.1 Graph Structure	17
4.2 Graph Generator	18
5 Final Application	21
5.1 Functionality	21
5.2 Structure	22
6 Results	25
6.1 Dataset	26
6.2 Performance of Nearest Neighbour Search	26
6.3 Dijkstra's Algorithm with Priority Queue	27
6.4 Bidirectional Search using Dijkstra's Algorithm	27
6.5 Performance Improvements on Android	29
6.6 Pollution and Distance Scores	30

CONTENTS

7	Conclusions and Outlook	33
A	Dijkstra's Algorithm	35
	Bibliography	37

Chapter 1

Introduction

The adverse health effects of air pollution on human health have been documented for many years. The World Health Organization (WHO) especially warns from the substantial burden of disease created by particulate matter (PM), which reduces the average life expectancy in Europe by 9 months. This is a significant number, considering that 80% of the European population lives in cities with levels of PM exceeding the levels considered safe by the WHO Air Quality Guidelines [1].

The OpenSense project of ETH Zurich and EPF Lausanne investigates means of using distributed wireless sensor network technology to monitor air pollution and to gain measurements with high spatial and temporal resolution. The high precision of this data creates a variety of new possibilities to reduce the daily exposure to air pollution of the population of urban areas, for instance by enabling people to choose healthier paths in their daily commute.

1.1 Contributions

In this project we developed an application for Android smart phones, which provides users with the healthiest path and the shortest path between two arbitrary locations in the city of Zurich, using the urban air pollution maps derived in [2].

Additionally, the application provides the user with a comparison of the two paths, indicating how much less polluted the healthiest path is compared to the shortest path in percentages and how much longer it is in meters. This allows users to make an informed choice, when deciding which path to take.

Although the application does not rely on external server infrastructure to compute the optimal paths, it is still fast and responsive. To achieve this, we studied and compared the performance of various routing algorithms. We then implemented the most promising algorithms and benchmarked them

on real smart phones and evaluated how well they perform on the limited hardware of the mobile devices. We chose the best performing algorithm and further optimized it for Dalvik, the Java Virtual Machine (JVM) running on Android, and used it to develop the final application.

1.2 Related Work

A prototype for a health-optimal route planner for Android was already developed as part of a previous project and presented as a possible use for air pollution maps in [2]. However, this previous application works with a server-client infrastructure, using a server to compute the healthiest and the shortest paths, and is not optimized for performance. While it works well as proof of concept, it has some significant drawbacks: On one hand, the cost of maintaining the server infrastructure on the other hand, the high average computation time for the healthiest path (4.1 seconds) [3]. Those two problems make it unsuitable for public distribution on an Android marketplace.

1.3 Overview

The health-optimal route planner developed in this project will be introduced in this report using a bottom-up approach. The first sections will present the graph algorithms which were used to develop an *optimal path provider*, whose actual *implementation* into a working smart phone application will be described later on.

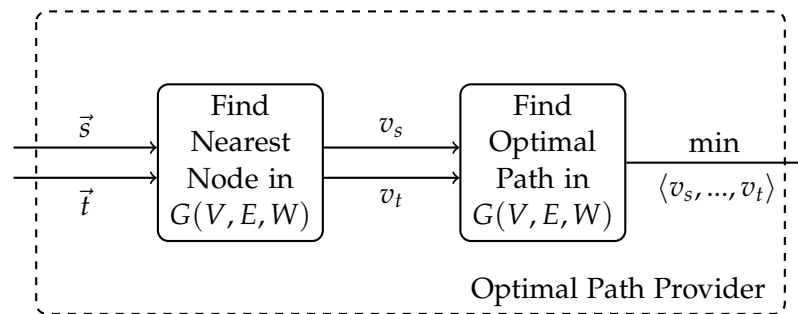


Figure 1.1: Illustrates the operations performed by the Optimal Path Provider in order to find the optimal path in a road graph $G(V, E, W)$ between the latitude / longitude pairs \vec{s} and \vec{t} .

Optimal Path Provider. The optimal path provider is at the core of the developed application. It contains a *weighted directed graph* $G(V, E, W)$, whose nodes and edges represents the road-network of the city of Zurich and whose weights contain the distance and the pollution data. The input to

the optimal path provider are the coordinates of two arbitrary positions in the city. It then performs a *nearest neighbour search* to determine the closest nodes inside the graph, between which it will then perform an *optimal path search*, using the two sets of edge-weights to determine the healthiest and the shortest path respectively.

Implementation. The finally implemented Android application takes the names of two locations in Zurich and uses the Google Places API to map them to real GPS coordinates. These coordinates are then fed to the optimal path provider and the resulting path is then displayed on a map, along with the current user location. Furthermore, already computed paths are stored into a database and made available through a history view.

During the implementation of the Android application, special attention was posed to performance aspects. This includes the choice of suitable algorithms and data-structures, as well as an implementation which takes into account particular properties of the Android Java Virtual Machine (JVM).

The last section of this report will present some performance comparisons and results, gathered from tests performed on a smart phone.

Nearest Neighbour Search

As illustrated in Figure 1.1, the first step performed by the optimal path provider in order to determine the shortest path between two arbitrary locations \vec{s} and \vec{t} in the city of Zurich, is to match those two locations to the closest nodes inside the road graph. This problem is known in the literature as the *Nearest Neighbour Search*, which for the two dimensional case of finding the closest geographical node inside a road-graph $G(V, E)$ can be formally described as follows:

We are given a set of nodes V , where each node $v_i \in V$ maps to a point:

$$\vec{v}_i = \begin{pmatrix} x_i \\ y_i \end{pmatrix}, \quad (2.1)$$

and where x_i and y_i represent a latitude / longitude pair inside a determined geographic area. We are also given some input vector

$$\vec{s} = \begin{pmatrix} x_s \\ y_s \end{pmatrix}, \quad (2.2)$$

which represents a point in the same area. We now want to find the node $v_s \in V$ that minimizes the euclidean distance $d(\vec{s}, \vec{v}_s)$ for all $v_i \in V$:

$$v_s = \arg \min_{v_i \in V} |\vec{s} - \vec{v}_i| \quad (2.3)$$

Linear Search. The naive solution to the nearest neighbour search is to compute the distance between \vec{s} and all $\vec{v}_i \in V$ and then to choose the node v_i with the smallest distance $d(\vec{s}, \vec{v}_i)$. This approach has a time complexity of $\mathcal{O}(|V|)$, which does not scale well for larger road graphs and is quite high, considering that the shortest path algorithm chosen for this project (see Chapter 3) runs in $\mathcal{O}(|V| \log(|V|))$. Therefore, a special data structure was adopted, which allows significantly faster searches.

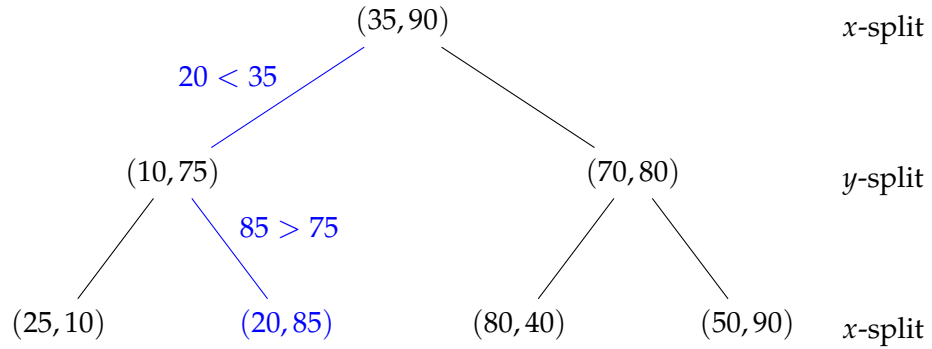


Figure 2.1: Shows how a new (x, y) pair is sorted into a two dimensional kd-tree using the x -coordinate in the first level and the y -coordinate in the second.

2.1 KD-Tree

A kd-tree (*k-dimensional-tree*) is a spatial data structure that hierarchically decomposes a k -dimensional space into a smaller number of cells, each of which contains some points from an input-set, by alternatingly splitting it along its dimensions [4].

In the two-dimensional case road-graph case, a kd-tree is obtained by sorting the nodes $\vec{v}_i \in V$ into a binary tree by comparing the latitude (x -split) at even levels and the longitude (y -split) at uneven levels.

Algorithm 1 Recursive function to build two dimensional kd-tree

```

▷ Input: Set  $|V|$  of nodes to insert in tree, current level of tree.
▷ Output: Balanced kd-tree.
function 2D-TREE( $V, level$ )
    if  $level \bmod 2 = 0$  then                                     ▷ Even level
         $v_m \leftarrow$  select  $v_i \in V$  where  $x_i = \text{median } \forall x_i$ 
    else                                                         ▷ Uneven level
         $v_m \leftarrow$  select  $v_i \in V$  where  $y_i = \text{median } \forall y_i$ 
    end if
     $v_m.\text{leftChild} \leftarrow$  2D-TREE( $v_i < v_m, level + 1$ )
     $v_m.\text{rightChild} \leftarrow$  2D-TREE( $v_i > v_m, level + 1$ )
    return  $v_m$ 
end function

```

Construction. The kd-tree can be built recursively as shown in Algorithm 1 starting from the root node at level 0, and inserting smaller nodes to the left and larger nodes to the right according to the current splitting axis. In order to obtain a *balanced* kd-tree, in which each leaf node has about

the same distance from the root node, one has to select the node with the median coordinate along the splitting axis from the set of nodes V in each step. This operation is quite expensive, which however is not a problem for this application, since the kd-tree can be computed upfront once and stored together with the graph.

Nearest Neighbour Search. A balanced kd-tree allows to perform nearest neighbour searches very efficiently in $\mathcal{O}(\log |V|)$ time [5]. The idea is to recursively go down the tree, as if the search node was being inserted, until reaching a leaf node. This node is not necessarily the nearest neighbour, since there still could be closer nodes on another sub-tree, but it is a good first estimate. After reaching this node, the algorithm walks up the tree again, looking at each node if it is closer than the current estimate and if the sub-tree on the other side of the splitting axis could potentially contain a closer node. If that is the case, it starts a new search on that sub-tree but if not, a whole part of the tree can be dismissed. How the algorithm can determine if a sub-tree can be dismissed, will be explained in the following step by step description of the algorithm:

The search of the node \vec{v}_i closest to the input coordinate \vec{s} is performed as follows:

1. Recursively move down the tree starting from the root node, as if \vec{s} was being inserted into the tree.
2. Once a leaf node v_b is reached, mark it as current best.
3. Walk up the tree (unwind the recursion) performing the following steps at each node v_i :
 - Compute the distance $d(\vec{s}, \vec{v}_i)$ and set $v_b \leftarrow v_i$ if $d(\vec{s}, \vec{v}_i) < d(\vec{s}, \vec{v}_b)$.
 - Check if there could exist any closer points on the other side of the splitting axis, by looking if the circle centred at the search point \vec{s} and with the minimum distance $d(\vec{s}, \vec{v}_b)$ as radius crosses the splitting coordinate, which can be performed by a very simple comparison operation:

$$d(x_i, x_s) < d(x_b, x_s), \quad (2.4)$$

assuming that x is the current splitting axis. If the radius crosses the splitting coordinate, there could be a nearer point on the other sub-tree of v_i and the algorithm starts a new search on that sub-tree. Else, the algorithm can continue walking up, dismissing the other sub-tree.

4. The algorithm terminates, as soon as it reaches the root node again and returns the nearest neighbouring node v_b .

Chapter 3

Optimal Path Problem

A road-network can be considered as a graph, where road junctions are represented by nodes and the roads are represented by positively weighted edges between two junctions. Therefore, finding the best route inside a road-network, can be achieved by solving the single source least-cost path problem for a graph, which consists in finding, the path from a source node v_s to a target node v_t which minimizes the sum of the weights of its edges.

In this section, we will assume that we are given a weighted directed graph $G(V, E, W)$, where V denotes the set of vertices v_i , E the set of edges $e_{i,j}$ between two nodes $v_i, v_j \in V$ and $W : E \rightarrow \mathbb{R}$ a weight function, mapping edges to real-valued weights. In order to fulfil the requirements of showing the user the healthiest path and the shortest path, there are two sets of weights: one indicating the air pollution density in particles per cm^3 between two nodes and one indicating the distance in meters.

The potential φ_p of a path $p_{0,k} = \langle v_0, v_1, \dots, v_k \rangle$ is given by the sum of the weights of its constituent edges:

$$\varphi_p = \sum_{i=1}^k W(e_{i-1,i}). \quad (3.1)$$

The lowest potential of a path from v_s to v_t is defined as:

$$\mu_{s,t} = \begin{cases} \min\{\varphi_p : v_s \rightsquigarrow v_t\} & \text{if at least one } p_{s,t} \text{ exists} \\ \infty & \text{otherwise.} \end{cases} \quad (3.2)$$

The least-cost path between v_s and v_t is defined as any path with the potential $\mu_{s,t}$ [6].

3.1 Dijkstra's Algorithm

Dijkstra's algorithm, which was first published by Edgar Dijkstra in 1959 [7], is a greedy algorithm which solves the least-cost path problem on a weighted, directed graph $G = (V, E, W)$ for the case in which all edge weights are non-negative.

The idea of the algorithm is to visit all nodes of the graph in ascending order of their distances from the source node (closest nodes first), until the target node is reached.

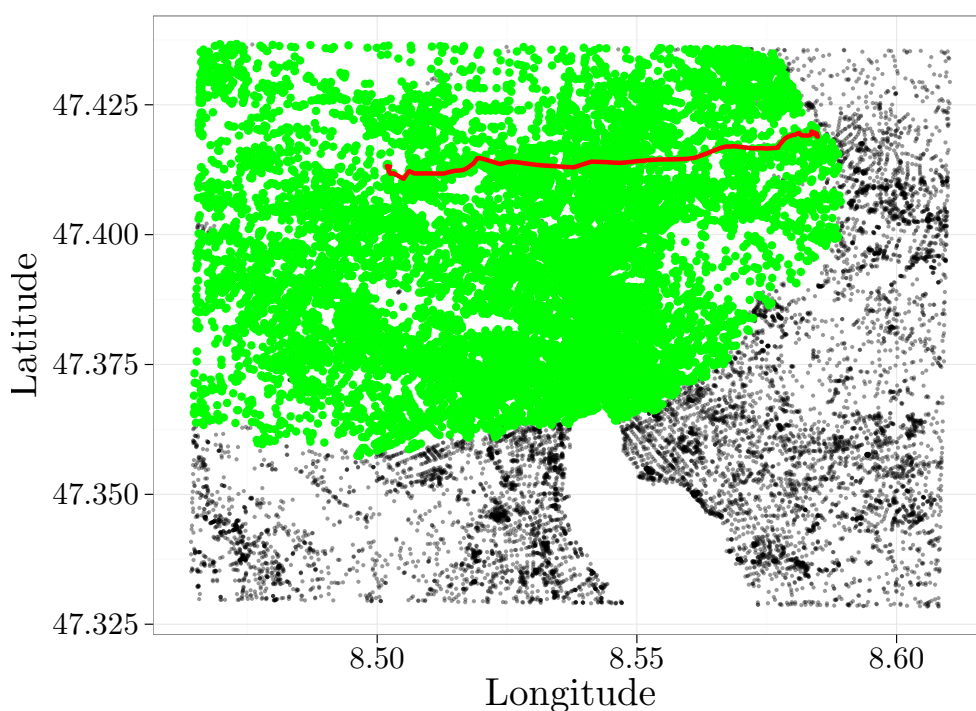


Figure 3.1: Map showing the nodes visited during the execution of a shortest path search between two nodes in the city of Zurich using Dijkstra's algorithm. The resulting shortest path is depicted in red. One can see, that the algorithm terminates as soon as the target (below) is reached.

Formal Description. More formally, the algorithm maintains a set of already visited nodes S , whose potential φ , the shortest path weight to the source, has already been determined. Furthermore it keeps a set of unvisited nodes Q , a map $\Phi(V \rightarrow \varphi)$, that contains the potentials φ_i of all $v_i \in G$ and a map $P(V \rightarrow V)$, mapping each node in the shortest path to its previous node in the path. To find the shortest path between the source node v_s and the target node v_t , the algorithm proceeds as described in Algorithm 2.

After the initialization phase, the set of visited nodes S is empty, while the

Algorithm 2 Dijkstra's algorithm.

▷ Input: Weighted graph $G(V, E, W)$, start and target nodes $(v_s, v_t) \in V$
 ▷ Output: Least-cost path $v_s \rightsquigarrow v_t$

function DIJKSTRA(G, v_s, v_t)

INITIALIZE(G, v_s) ▷ Initialize S, Q, Φ and P

while $Q \neq \emptyset$ **do**

$v_u \leftarrow \text{GET-MIN}(Q)$ ▷ Get unvisited node closest to source

if $v_u = v_t$ **then**

return P ▷ Found closest path to target!

end if

$S \leftarrow S \cup \{v_u\}$ ▷ Add v_u to set of visited nodes

for all Neighbours v_i of v_u **do**

if $v_i \in Q$ **then**

▷ Update potential φ_i , if path through v_u has lower potential

UPDATE-POTENTIAL(v_u, v_i)

end if

end for

end while

return Error: Target node $v_t \notin G$ or not reachable.

end function

set of unvisited nodes Q contains all nodes $v_i \in G$. The potentials φ_i are set to ∞ for all nodes, except for the source node v_s , where the potential is set to 0. The set of previous nodes in the optimal path P is set to *null* for each node.

In each step, the algorithm visits the unvisited node $v_u \in Q$ with the lowest potential, which corresponds to the unvisited node closest to the source v_s . Since in the beginning the potential φ_i of all nodes is set to ∞ , except for the source node v_s , the algorithm starts at v_s .

The algorithm proceeds by going through all adjacencies of v_u , scanning each unvisited neighbour $v_i \in Q$ and computing its distance from the source through v_u . More formally, if v_u has the source potential φ_u and the potential between v_u and v_i is given by the edge-weight $w_{u,i}$, the potential $\varphi_{s \rightarrow u \rightarrow i}$ for node v_i going through v_u is equal to the sum $\varphi_u + w_{u,i}$. If $\varphi_{s \rightarrow u \rightarrow i} \leq \varphi_i$, the potential φ_i is updated to $\varphi_{s \rightarrow u \rightarrow i}$ and the predecessor p_i of v_i is set to v_u (see Algorithm 3).

The algorithm terminates as soon as it reaches the target node and returns the map of previous nodes P , which can then be used to trace back the shortest path from the target to the source.

Algorithm 3 Function to update the potential.

▷ Input: Visited node v_u and one of its neighbours v_i
 ▷ Output: Updated potential of v_i going through v_u if it is smaller than the previous potential.

function UPDATE-POTENTIAL(v_u, v_i)
 $\varphi_{s \rightarrow u \rightarrow i} := \varphi_u + w(v_u, v_i)$ ▷ Potential of v_i going through v_u
 if $\varphi_{s \rightarrow u \rightarrow i} \leq \varphi_i$ **then**
 $\Phi(v_i) \leftarrow \varphi_{s \rightarrow u \rightarrow i}$ ▷ Update potential
 $P(v_i) \leftarrow v_u$ ▷ Update predecessor
 end if
end function

Optimality. Since Dijkstra’s algorithm chooses to visit the closest unvisited node in each step and does not change the least-cost path to this node after that, it can be classified as a greedy algorithm. However, it still yields the optimal solution to the least-cost path problem. The proof of optimality is based on the fact, that if v_i is a node on the least-cost path between v_s and v_t , the knowledge of the latter implies the knowledge of the least-cost path between v_s and v_i [7]. Since the algorithm searches the closest unvisited node to the source in each step, it can be shown by induction that the least-cost path is known to each visited node $v_i \in S$ [6].

Time Complexity. As shown above, in order to find the shortest path between to nodes $(v_s, v_t) \in V$, Dijkstra’s algorithm needs to visit at least all nodes on the path and scan all of their adjacencies each time it visits a node. This yields a running time of $\mathcal{O}(|V| + |E|)$, where $|V|$ is the number of nodes $v_i \in G$ and $|E|$ the number of edges. Additionally in each step the algorithm needs to find the closest unvisited node to the source $v_u \in Q$, which assuming a naive implementation of Q as an array- or linked-list adds $\mathcal{O}(|V|)$ computation steps for each visited node (see Appendix A for implementation). This yields a running time of $\mathcal{O}(|V|^2 + |E|) = \mathcal{O}(|V|^2)$. The next section will show, how the running time can be significantly reduced by using a more efficient data structure for Q .

3.2 Speeding up Dijkstra’s Algorithm: Priority Queue

As shown in the previous section, the linear search of the next unvisited node to visit adds a complexity of $\mathcal{O}(|V|)$ to each computation step in Dijkstra’s algorithm. This complexity can be significantly reduced by storing the set of unvisited nodes Q in a priority queue.

A priority queue is a queue structure, in which each element has an attached priority. It supports methods to enqueue elements with a certain priority,

change an elements priority and to dequeue elements in order of their priority. In 1987, Fredman et al. proposed the Fibonacci-Heap implementation of a priority queue [8], which allows to dequeue the element with the minimum priority in $\mathcal{O}(\log |V|)$ time and to decrease a priority of an element in constant time $\mathcal{O}(1)$ [6].

Algorithm 4 Function to update the potential using a min-priority-queue.

▷ Input: Visited node v_u and one of its neighbours v_i
 ▷ Output: Updated potential of v_i going through v_u if it is smaller than the previous potential.

function UPDATE-POTENTIAL(v_u, v_i)
 $\varphi_{s \rightarrow u \rightarrow i} := \varphi_u + w(v_u, v_i)$
 if $\varphi_{s \rightarrow u \rightarrow i} \leq \varphi_i$ **then**
 $\Phi(v_i) \leftarrow \varphi_{s \rightarrow u \rightarrow i}$
 $P(v_i) \leftarrow v_u$
 ▷ Lower priority of v_i using the new potential $\varphi_{s \rightarrow u \rightarrow i}$
 Q.DECREASE-PRIORITY($v_i, \varphi_{s \rightarrow u \rightarrow i}$)
 end if
end function

Time Complexity. By using the Fibonacci-Heap min-priority queue implementation for the set of unvisited nodes Q and by changing the implementation of the update-potential function according to Algorithm 4, to always decrease the priority when the potential of an unvisited node is decreased, the time complexity of the search of the closest node in each step can be reduced to $\mathcal{O}(\log(|V|))$. Considering the time complexity needed to decrease the priority of the unvisited nodes in each update, this yields a total execution time of $\mathcal{O}(|V| \log |V| + |E| * 1) = \mathcal{O}(|V| \log |V|)$.

3.3 Bidirectional Search

One shortcoming of Dijkstra's algorithm, is that it uniformly searches in all directions, when computing the shortest path between a node v_s and a node v_t , regardless of the direction of v_t (see Figure 3.1). Some algorithms, like the A* algorithm, correct this shortcoming by adding heuristics based on the knowledge of the geographical position of the target [9]. This heuristics are however usually based on the euclidean distance between the two points, which requires the triangle inequality to be valid [9]. Unfortunately, this assumption can not be made when dealing with pollution data. A* finds the optimal path if the heuristic is a admissible heuristic. Hence, it is possible to compute the health-optimal path using A*, but the heuristic needs to be very pessimistic in order to be optimal (e.g., heuristic could be the multiplication

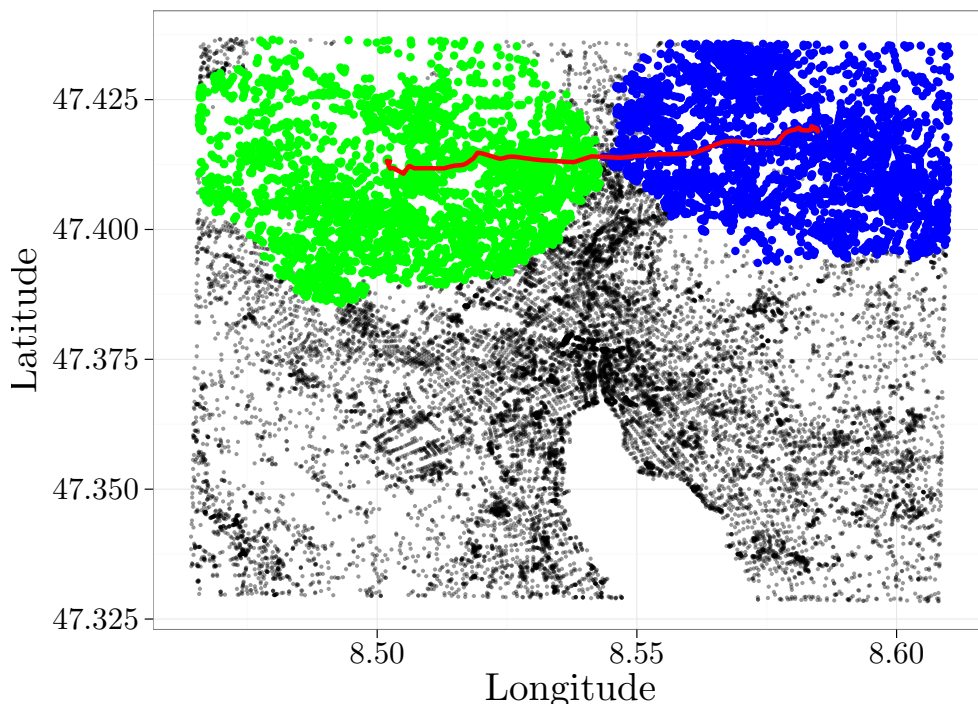


Figure 3.2: Map showing the nodes visited during the front (green) and the back shortest path search (blue), between two random nodes in the city of Zurich, while performing a bidirectional search using Dijkstra's algorithm. The resulting shortest path is depicted in red.

of the distance with the minimum pollution level in the whole area). As a result A^* does not perform very well and, hence, is often even slower than bidirectional Dijkstra [3].

While in unidirectional search algorithms the destination plays a minor role than the origin, bidirectional search algorithms utilize both the origin and the destination uniformly by searching alternately from the origin side and from the destination side [9]. When the two searches meet at an intersection and the optimal stopping criterion is reached, the path can be traced back from the intersection to the source and to the target.

Termination. Intuitively, a bidirectional search from node v_s to v_t grows two search areas, one around the source and one around the target, until they both meet (see Figure 3.2). Therefore, a possible stopping criterion could be when a center-node v_c was visited by both the forward and the backward search. More formally, assuming that S_f is the set of nodes visited by the forward search, while S_b is the set of nodes visited by the backward search, it would terminate as soon as $S_f \cap S_b = \{v_c\} \neq \emptyset$.

Although this criterion yields a valid path between v_s and v_t , it does not guarantee an optimal path, since there could still exist a valid edge $e_{i,j}$ be-

Algorithm 5 Bidirectional search using Dijkstra's algorithm.

▷ Input: Weighted graph $G(V, E, W)$, start and target nodes $(v_s, v_t) \in V$
 ▷ Output: Least-cost path $v_s \rightsquigarrow v_t$

function BIDIRECTIONAL-SEARCH(G, v_s, v_t)
 $\mu := \infty$ ▷ Score of best path seen so far.
 $f := \text{new DIJKSTRA}(v_s, v_t)$ ▷ Initialize forward search.
 $b := \text{new DIJKSTRA}(v_t, v_s)$ ▷ Initialize backward search.
 while $\varphi_{f,next} + \varphi_{b,next} \geq \mu$ **do**
 $f.\text{DO-STEP-AND-UPDATE-}\mu\text{-IF-NECESSARY}()$
 $b.\text{DO-STEP-AND-UPDATE-}\mu\text{-IF-NECESSARY}()$
 end while
end function

tween node $v_i \in S_f$ and node $v_j \in S_b$ which leads to a path with a better score than the path through v_c . Therefore, a stronger condition must be introduced.

In order to achieve this, the algorithm must maintain the length μ of the best path seen so far, which is initially set to $\mu = \infty$. For simplicity we can assume that the algorithm is in the forward search, since the solution is symmetric for the backward search. When Dijkstra's algorithm scans through the neighbours of a node $v_i \in S_f$ and sees a node $v_j \in S_b$, it determines that there is a potential shortest path from v_s to v_t with the score:

$$\mu_{i,j} = \varphi_{f,i} + w_{i,j} + \varphi_{b,j}, \quad (3.3)$$

where $\varphi_{f,i}$ is the shortest path score from v_s to v_i , $w_{i,j}$ the edge-weight of the edge between v_i and v_j and $\varphi_{b,j}$ the shortest path score between v_j and v_t . If the newly observed path-score is better than all previously observed, μ is updated $\mu = \mu_{j,i}$.

The algorithm terminates as soon as the sum of the potentials of the closest unvisited nodes in both, forward and backward search, is greater or equal than the best observed path score μ :

$$\varphi_{f,next} + \varphi_{b,next} \geq \mu \quad (3.4)$$

which due to the greedy nature of Dijkstra's algorithm guarantees that μ is the score of the optimal path [10].

Graph Implementation

The graph lies at the core of the *optimal path provider*, as it contains all the geographical and pollution information that allow health-optimal routing. The graph data is provided by the OpenSense project [2], and consists of a list of approximately 27'000 *nodes*, representing road intersections, and 74'000 *weighted edges*, representing the walkable paths between those intersections. An edge has two distinct weights. One indicates the pollution grade and one the distance between two intersections.

This chapter describes the data structures and methods chosen to implement a graph, which can be efficiently stored and loaded by the application and enables fast nearest neighbour and shortest path queries.

4.1 Graph Structure

In general there are two ways to store a directed weighted graph $G(V, E, W)$ with n nodes: As *adjacency matrix*, an $n \times n$ matrix where each entry $a_{i,j}$ represents the weight of the edge between v_i and v_j , or as an *adjacency list*, a collection of lists containing the neighbouring nodes and the weight of the edge connecting them for each node v_i . Since in given road graph there are approximately three edges for each node, it was chosen to implement the graph as *adjacency list*, which is more memory efficient for sparse graphs.

As illustrated by the class diagram in figure 5.2 the graph implementation uses four different classes of objects:

Graph The main graph object, which holds an array of nodes, an array containing an array with the adjacencies for each node and the kd-tree representation. All those arrays can be directly queried with the the id of a node, which allows a very efficient constant access speed.

It also provides the methods to access the road-network data needed

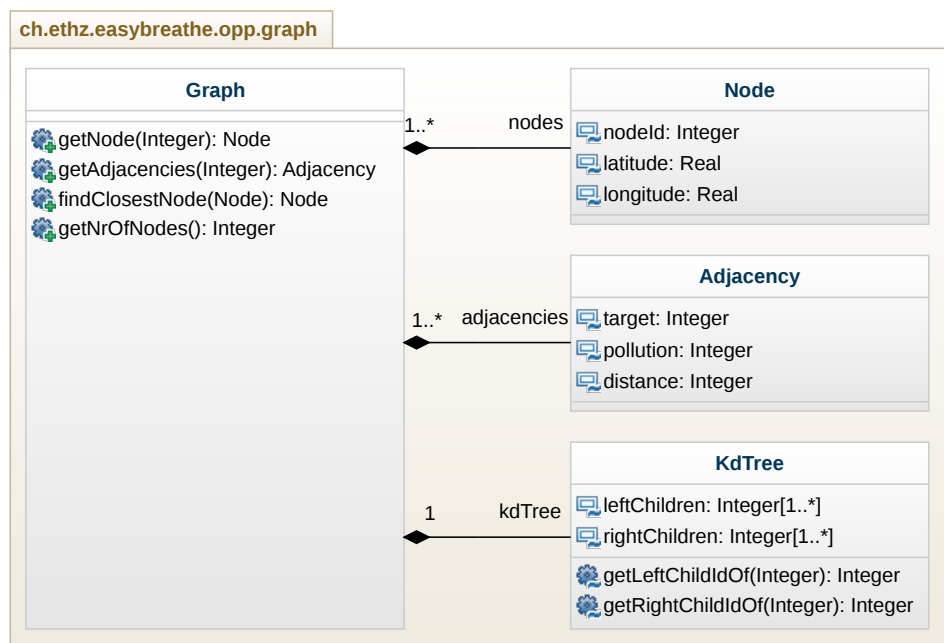


Figure 4.1: Class diagram of the graph implementation, showing the different classes, fields and methods implemented.

by the shortest path algorithm and a method to perform nearest neighbour queries using the kd-tree.

Node A simple data representation of a node holding its unique id, as well as its latitude and longitude.

Adjacency Contains the target of an adjacency, as well as the distance and the pollution of that edge.

KdTree Stores the child-nodes in the kd-tree for each node and is queried by the graph to perform nearest neighbour searches.

4.2 Graph Generator

The graph is generated by a stand-alone program, which takes two lists of comma-separated values, one containing the nodes and one containing the adjacencies, and creates a serialized graph object, which can then be loaded and used by the health-optimal routing application. The approach of generating the graph object in advance and to only load the serialized graph-object during the application runtime, has proven to be very advantageous, because it allows to perform a series of rather expensive computation steps in advance, without negatively influencing the application's performance:

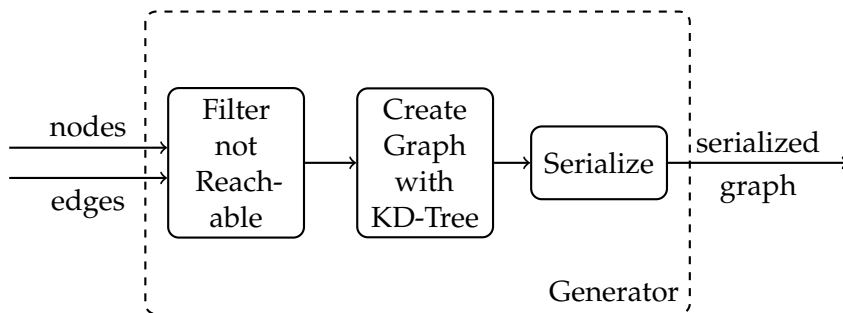


Figure 4.2: Shows the steps performed by the generator in order to create the serialized graph object from a list of nodes and a list of adjacencies.

Filtering Unreachable Nodes. The data provided for the road-graph, which is based on OpenStreetMap¹, contains a few nodes ($\approx 1\%$) that are not connected to the rest of the road-network. These can cause faulty results and therefore need to be removed from the graph before the shortest path algorithm can be applied. This is accomplished by running Dijkstra’s algorithm from a well connected graph node, without terminating when the target node is reached, and by removing all nodes that have not been visited by the algorithm after its execution.

Generate Graph The filtered data is then used to generate the graph: All nodes are instantiated with a unique sequential id starting from 1, the adjacencies are instantiated and mapped to the corresponding node-id in the adjacency list and the kd-tree is computed - a rather expensive task, which does not have to be performed at application runtime.

Graph Serialization Serialization is the process of translating an object into a state, from which it can be permanently stored and later re-loaded. Java provides two mechanisms to serialize an object:

The first approach is automatic serialization. In order to do this, all objects in the object graph that need to be serialized need to be marked, by implementing the empty “Serializable” interface. Java then applies a generic recursive algorithm, which first writes out the meta-data of all super classes of the object being serialized until it finds the “java.lang.Object” super-type and then starts to recursively serialize all objects top-down, starting from the topmost super-class and going down to the most derived class². This approach is very easy to use, but better performance can be achieved by using manual serialization.

¹<http://www.openstreetmap.org>

²Java Object Serialization Specification: <https://docs.oracle.com/javase/7/docs/platform/serialization/spec/serialTOC.html>

4. GRAPH IMPLEMENTATION

Manual serialization in Java is achieved by implementing the "Externalizable" interface, which contains two methods to serialize and to de-serialize the object. This allows to iteratively serialize the components of the graph, which leads to a significant speed-up (in the case of this project $\approx 50\%$) in the de-serialization time.

The finally serialized graph object has a size of 3.1MB.

Final Application

The final application allows the user to query the name of a starting and a target location in the city of Zurich and displays the healthiest along with the shortest path on a map, along with information comparing the two paths.

5.1 Functionality

The first view displayed upon starting the app is the main view, which consists of two input fields and a history view. It allows to perform the following actions:

Select a new pair of locations. This is achieved simply by starting to type the names of a start and a target destination in the dedicated query fields. While typing, the application provides suggestions for places in a drop-down list. As soon as a valid pair of locations is selected, the healthiest and the shortest path will be computed and displayed on a map, together with information comparing the two paths.

Select a pair of locations from the history. The user can also select a recent pair of start and target destinations from the history view by simply tapping on it. In this case it is not necessary to compute the paths and they can be directly displayed.

The healthiest and the shortest path are displayed along with the current position of the user on a Google map provided by the Android SDK. The healthiest path is displayed in green, while the shortest path is displayed in red. When the two paths overlap, a thinner green line is displayed on a broader red line.

Above the visualization of the map, the application displays information about the two computed paths. This information consists of how much

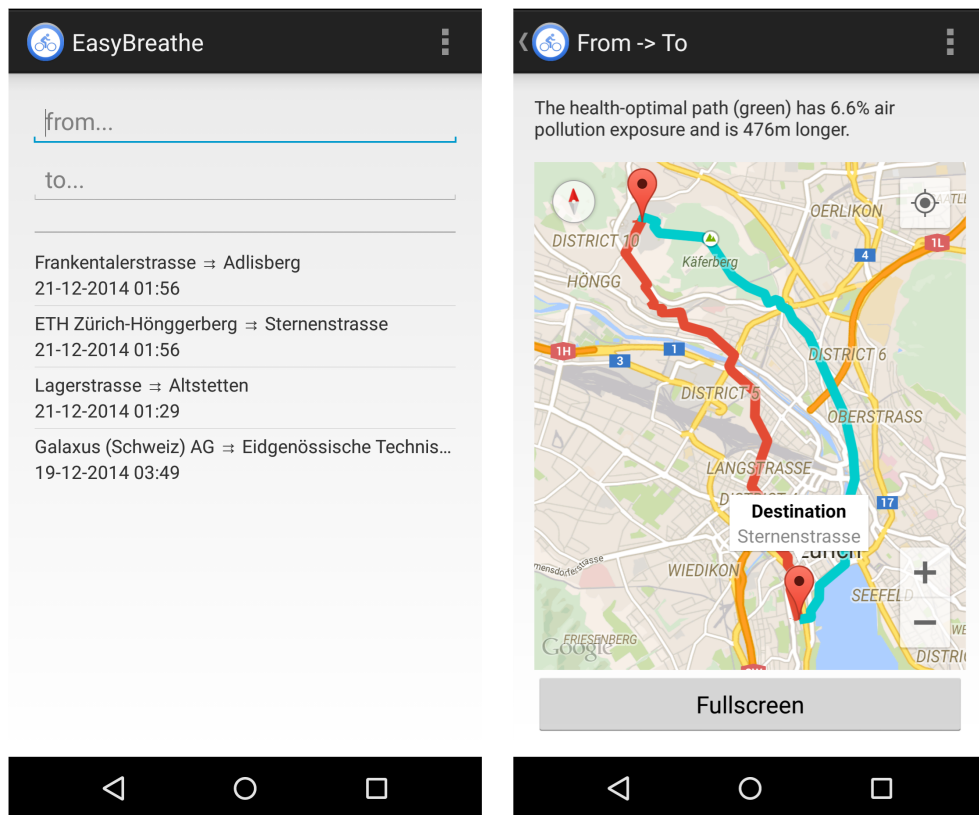


Figure 5.1: Shows the two main activities of the developed application. The activity on the left, allows to choose the start and destination locations, the activity on the right, displays the shortest and the healthiest path along with some information.

pollution can be avoided by choosing the healthiest path in percentages and how much longer the healthiest path is in meters. Below the map there is a button, which allows the user to view the map in full-screen.

The navigation between the views is consistent with the Android design guidelines¹: New views are opened by performing actions, while the back-button of the smart-phone can be used to navigate back.

5.2 Structure

The functionality of this application is achieved through various providers and background jobs, which will be described in this section by following the standard use cases.

¹<http://developer.android.com/design/patterns/index.html>

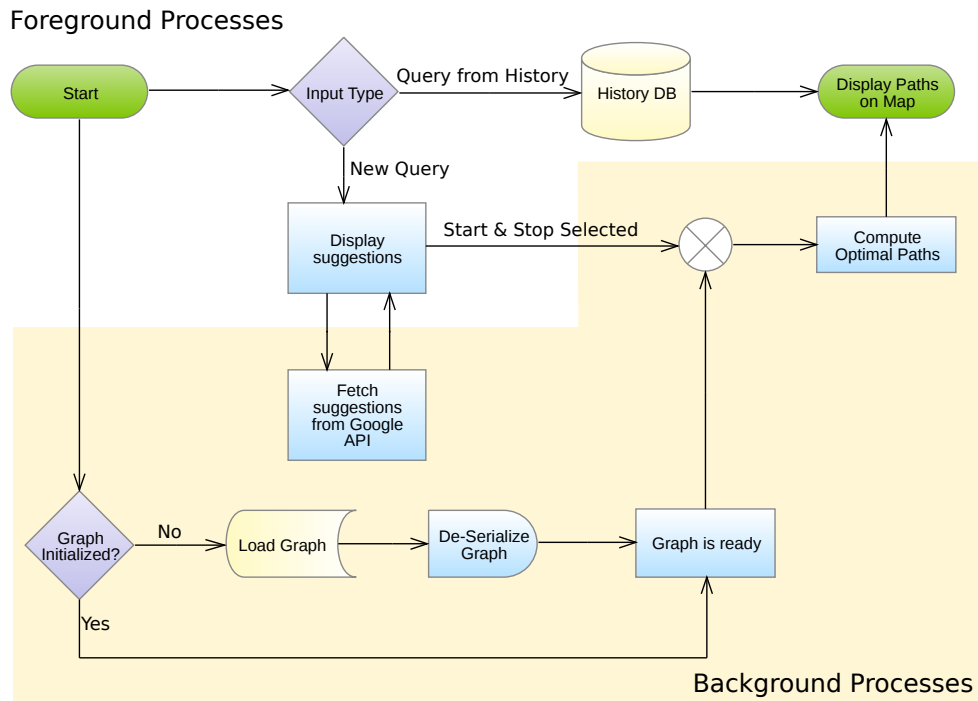


Figure 5.2: Shows how the main activity handles user input and initialization at the same time.

When the application starts, it needs to load and de-serialize the graph from the assets folder in the application directory in order to initialize the *Optimal Path Provider*. Since this operation is relatively time consuming, it is not performed on the main-thread of the application, but in a background thread. The ongoing de-serialization process is displayed to the user through a rotating progress indication in the bottom right corner of the screen.

While the application is initializing, the user can already input a new pair of locations in the two search-fields. While the user is typing, the application uses the partial input to get suggestions from the Google Places API² by sending JSON³-queries over http. This task is performed by the *Places Auto Complete Provider*, which, along with the name suggestions, also retrieves the GPS-coordinates of those places. These are first used to make sure, that only suggestions inside the boundaries of the city of Zurich are displayed and are later needed to compute the optimal paths.

Once two valid locations inside Zurich are selected, the application requests the healthiest and the shortest path from the *Optimal Path Provider*, which will then proceed to compute them in a background thread using a bidirectional search with Dijkstra's Algorithm. If the graph is not initialized yet at

²<https://developers.google.com/places/>

³<http://json.org>

this point in time, then the background thread will request a callback from the graph loader.

Once the two paths are computed, the *Optimal Path Provider* returns them to the main activity, which first proceeds to display the map with the paths and the path information and then stores the new paths, along with their path scores, into the history database.

Since the history database contains all information needed to display the optimal paths, history queries can be displayed directly, without waiting for the graph initialization.

Results

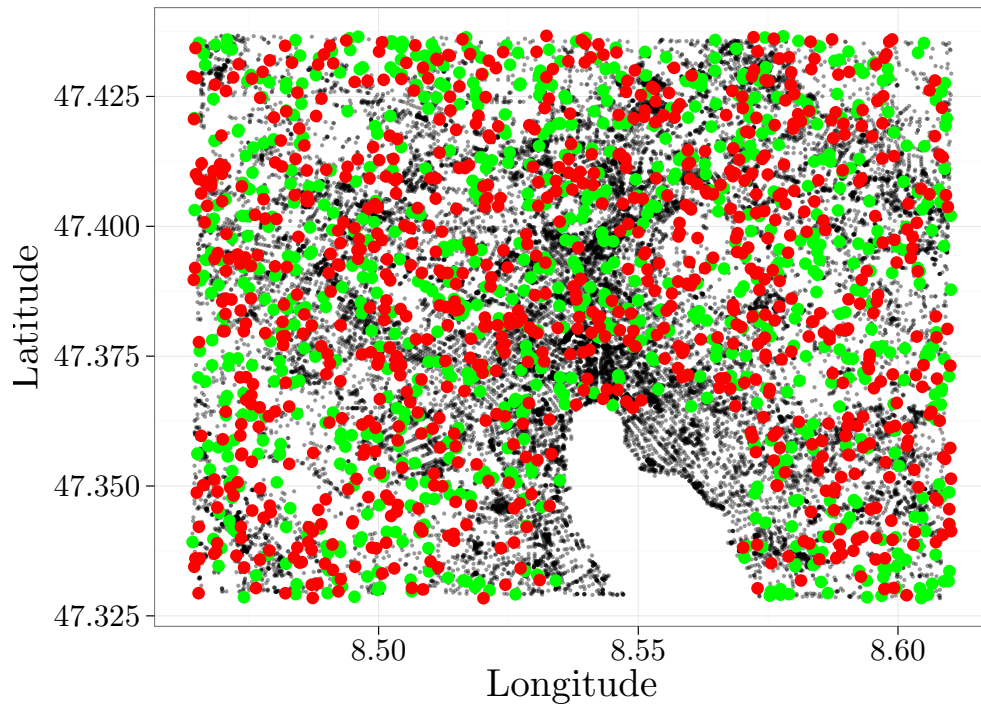


Figure 6.1: Displays the distribution of the 1000 randomly selected coordinate pairs used to perform the performance tests.

In order to compare different solution approaches to the shortest path problem for healthy routing on Android platforms and to choose and improve the most promising, various performance tests were performed on a LG Nexus 5 smart phone running Android 4.4 KitKat with 2 GB RAM and a quad-core 2.3GHz Snapdragon CPU.

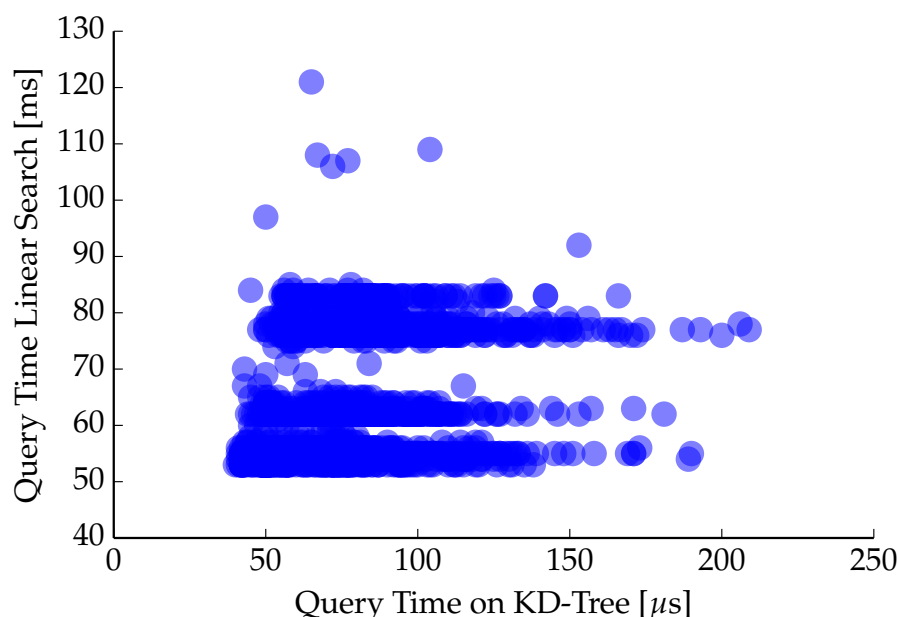


Figure 6.2: Compares the execution times of the priority queue implementation of Dijkstra's algorithm, with the execution times of its bidirectional variant for 1000 randomly distributed coordinate pairs in Zurich.

6.1 Dataset

The following results were gathered using on a road-network graph of the city of Zurich containing 26'748 nodes and 73'582 edges, with distance and pollution weights. The performance of the shortest path algorithms was measured using 1000 uniformly randomly selected start and target coordinates inside the city of Zurich, that are at least 1000 meters apart and excluding points that are not accessible, like the lake. (Figure 6.1).

6.2 Performance of Nearest Neighbour Search

To better quantify the speed-up obtained by adopting a kd-tree based search, rather than the linear search, both algorithms were run for 1000 points of the dataset. The average time for a linear nearest neighbour search was $65289\mu\text{s}$ ($\approx 65\text{ms}$), while it was $73\mu\text{s}$ for searches on the kd-tree.

This means, the kd-tree search was almost 1000 times faster than the linear nearest node search. Considering, that the road graph contains roughly 30'000 nodes, this result is consistent with the time complexities $\mathcal{O}(|V|)$ vs. $\mathcal{O}(\log(|V|))$ derived in Chapter 2.1.

As Figure 6.2 illustrates, the query times on the kd-tree are, with the excep-

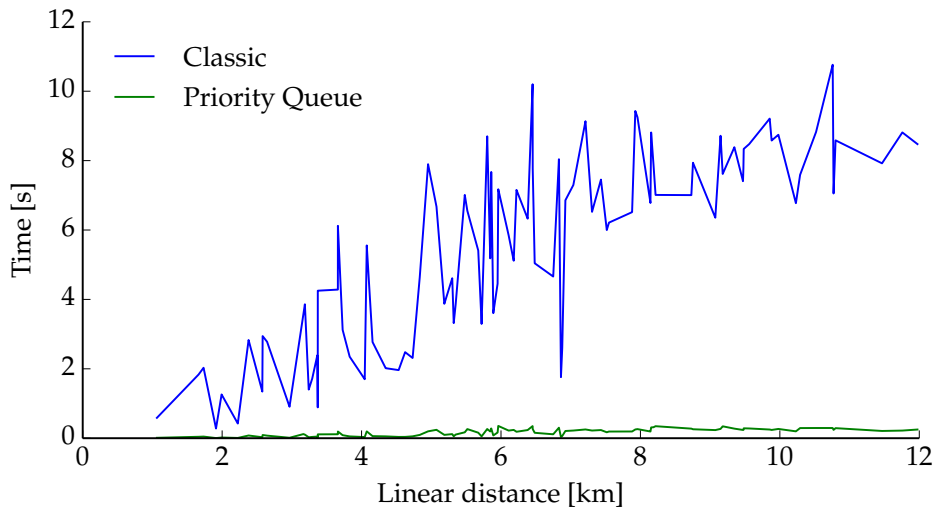


Figure 6.3: Compares the execution times of the classic implementation of Dijkstra's algorithm with the execution times of the priority-queue implementation for 100 randomly distributed coordinate pairs in Zurich.

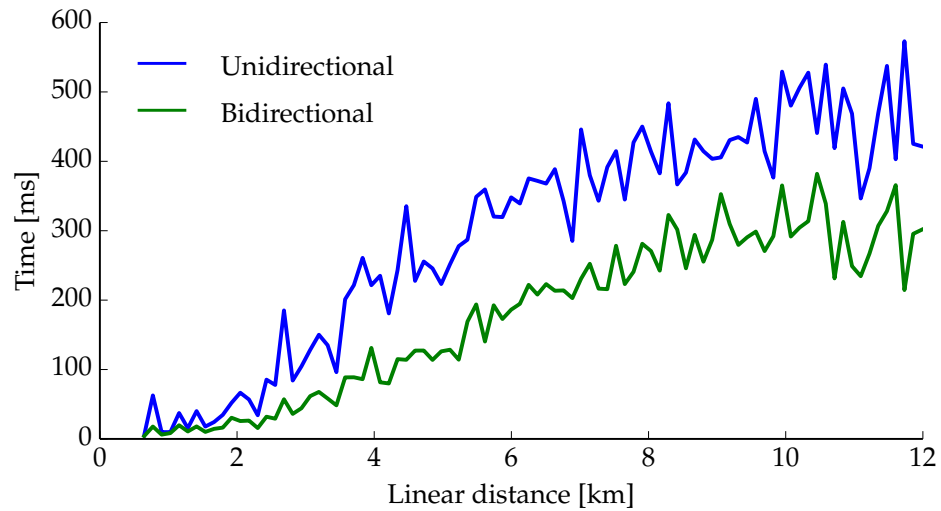
tion of a few outliers, distributed quite regularly around the average. This shows that the tree is well balanced.

6.3 Dijkstra's Algorithm with Priority Queue

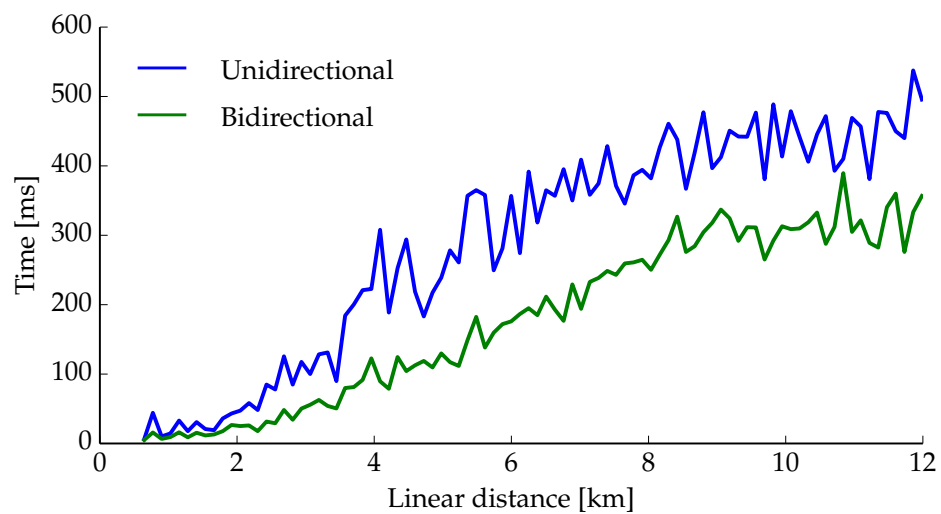
In Section 3.2 it was claimed that the time complexity of Dijkstra's algorithm could be reduced by an exponential factor by introducing a priority queue to determine the next node to visit in each step. This claim was confirmed by the time measurements taken while computing the healthiest (Figure 6.3) and the shortest paths between 100 locations from the dataset displayed in Figure 6.1.

6.4 Bidirectional Search using Dijkstra's Algorithm

After introducing the implementation of Dijkstra's algorithm using a priority queue, it was claimed that the shortest path search could be further sped by using the algorithm in a bidirectional search, that alternately searches from source and target. Intuitively this result was already illustrated in Figures 3.1 and 3.2, that showed that the bidirectional search needs to visit less nodes during a shortest path search than the unidirectional search. However, the bidirectional search also introduces additional overhead, because it needs to allocate memory for two unidirectional searches and needs to syn-



(a) Healthiest path computation time.



(b) Shortest path computation time.

Figure 6.4: Compares the execution times of the priority queue implementation of Dijkstra's algorithm, with the execution times of its bidirectional variant for 1000 randomly distributed coordinate pairs in Zurich. The average computation time of the healthiest path (a) was brought down from 290ms to 170ms, the average computation time for the shortest path (b) from 283ms to 196ms.

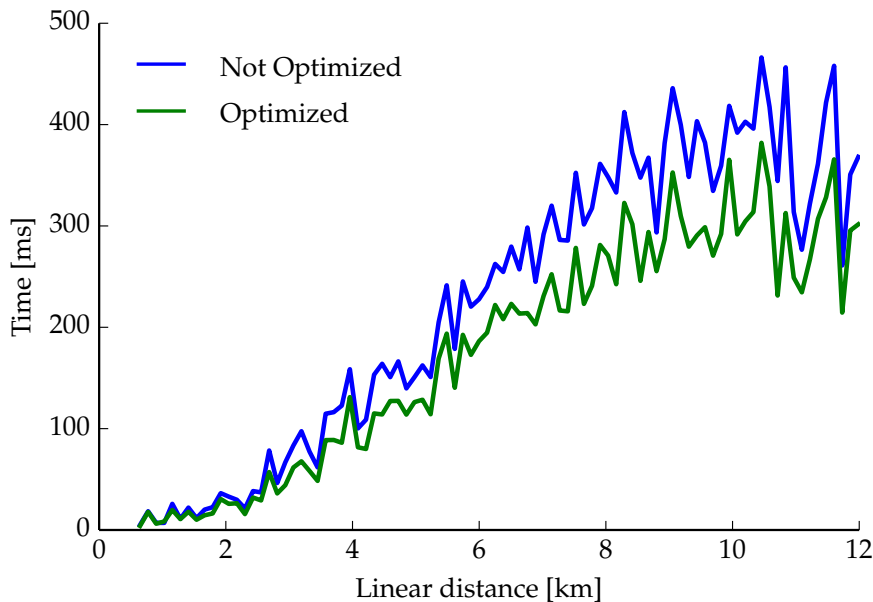


Figure 6.5: Compares the execution times of the priority queue implementation of Dijkstra's algorithm, with the execution times of its bidirectional variant for 1000 randomly distributed coordinate pairs in Zurich.

chronize between both searches, in order to achieve an optimal termination criterion.

Nevertheless, as shown in Figure 6.4, the bidirectional search performed better than the unidirectional search by a constant factor during 1000 executions between the points from our dataset, decreasing the average computation time from 290ms to 170ms for the health-optimal paths and from 283ms to 196ms for the shortest paths.

Additionally to the theoretical proof outlined in Section 3.3 the optimality of the bidirectional search was empirically verified by direct comparison to the results obtained by the unidirectional search.

6.5 Performance Improvements on Android

There are a few differences between Dalvik, the Android JVM¹, and other JVMs. Two of those had a particular impact on the performance of the shortest path algorithms implemented in this project:

¹Java Virtual Machine

Internal Setters and Getters. In object oriented programming languages, it is common practice to access internal fields through setter and getter methods (e.g. `int s = getSize()` rather than: `int s = this.size()`), because it has various advantages for the developer, some of which are centralized access control, easier debugging or more flexibility through sub-classing.

In native languages, such as C++, such methods get simply inlined at compilation time and therefore do not introduce any overhead at run-time. In standard JVMs, the inlining is usually performed by the Just In Time-compiler (JIT), a feature which is however not available in Dalvik. Therefore, according to the official Android documentation², direct field access is about three times faster than invoking a trivial getter-method.

Enhanced Loop Syntax. When iterating over an array `A[] a` in the following way: `for (int i = 0; i < a.length; i++) {...}`, Dalvik, not having a JIT, does not cache the length of the array and therefore needs to query it at every iteration step. This can be avoided using the *enhanced loop syntax*: `for (A element : a){...}`, which again makes to loop about three times faster.

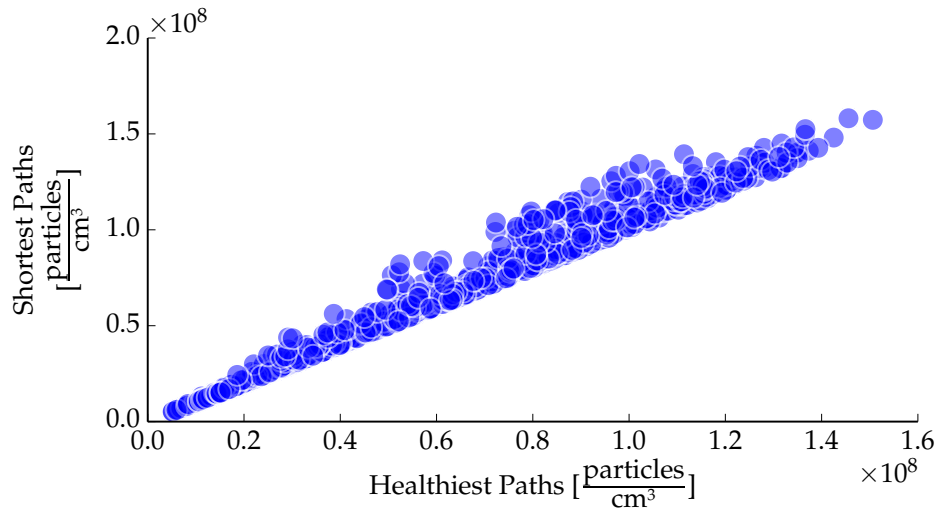
Figure 6.5 shows the performance improvement achieved by taking into account the aforementioned performance tips.

6.6 Pollution and Distance Scores

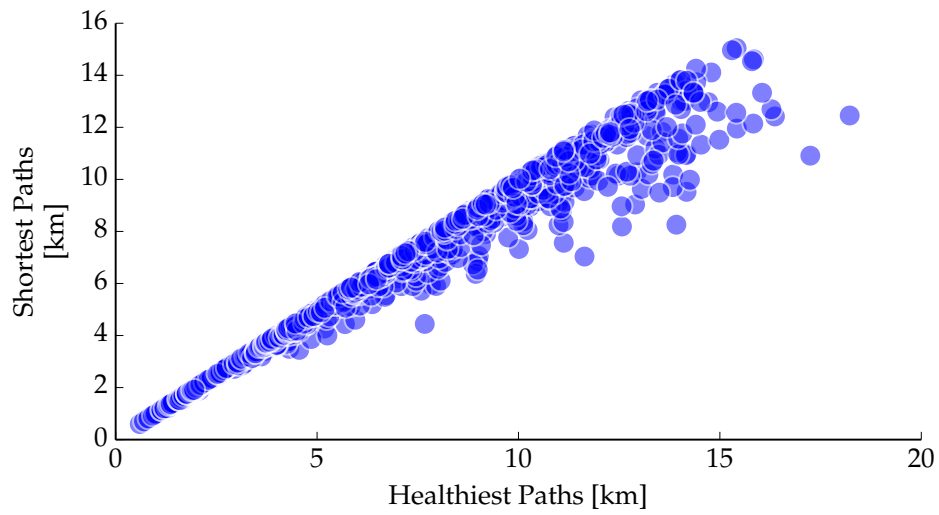
Running the shortest path algorithm on the test data showed, that the average healthiest path in Zurich has approximately 6.5% less pollution exposure and is approximately 489m longer, than its corresponding shortest path.

The scores for the healthiest and the shortest paths are displayed in Figure 6.6. The fact, that no shortest path has a lower pollution than the healthiest path and the fact, that no healthiest path is shorter than a shortest path, suggest that the bidirectional search was implemented correctly and gives optimal results.

²<http://developer.android.com/training/articles/perf-tips.html>



(a) Pollution scores.



(b) Distance scores.

Figure 6.6: Compares the pollution and the distances of the healthiest and the shortest paths.

Conclusions and Outlook

In this project we proposed a combination of the kd-tree nearest neighbour search and the bidirectional shortest path search using Dijkstra's algorithm enhanced by a priority queue in order to solve the problem of health optimal routing on the limited hardware resource provided by a smart phone.

The implementation of this approach into a working smart phone application for Android has proven its quality, reducing the average computation time for a health-optimal path in the city of Zurich to 170ms.

Particular attention was also posed to the efficient implementation of the graph, which in the case of Zurich can be stored into a file of approximately 3.1MB size and loaded into memory by the application in around 2.3 seconds on average.

The final application is ready to be distributed through the Google Play Store, but leaves room for further development: While at the moment the road-graph is deployed as part of the application and has to be updated together with the latter, it would be advantageous to de-couple the deployment of the graph data from the deployment of the application. This should be a quite easy task to achieve, especially given the contained size of the graph-file, and will enable users to profit from the high temporal resolution offered by the data gathered in the OpenSense project.

Appendix A

Dijkstra's Algorithm

This appendix contains the pseudo-code implementation of a few methods which were omitted in chapter 2.

Algorithm 6 Function to initialize S , Q , Φ and P

```
function INITIALIZE( $G, s$ )  
   $S \leftarrow \emptyset$  ▷ Initially, set of visited nodes is empty  
  for all  $v_i \in V$  do  
    if  $v_i \neq s$  then  
       $\Phi(v_i) \leftarrow \infty$  ▷ Initially, distance from source is  $\infty$   
    else  
       $\Phi(s_i) \leftarrow 0$   
    end if  
     $P(v_i) \leftarrow \text{null}$  ▷ Initially, all previous nodes are null  
     $Q \leftarrow Q \cup \{v_i\}$  ▷ Add all nodes to set of unvisited nodes  
  end for  
end function
```

Algorithm 7 Function to extract closest unvisited node from Q assuming a naive implementation of Q as array- or linked-list.

```
function GET-MIN( $Q$ )  
   $d_{min} := \infty$   
   $v_{closest} := null$   
  for all Nodes  $v_i \in Q$  do  
    if  $d_{min} \leq \Phi(v_i)$  then  
       $d_{min} \leftarrow \Phi(v_i)$   
       $v_{closest} \leftarrow v_i$   
    end if  
  end for  
   $Q \leftarrow Q \setminus \{v_{closest}\}$   
  return  $v_{closest}$   
end function
```

Bibliography

- [1] WHO, "Review of evidence on health aspects of air pollution – REVI-HAAP project: final technical report," 2013.
- [2] D. Hasenfratz, O. Saukh, C. Walser, C. Hueglin, M. Fierz, T. Arn, J. Beutel, and L. Thiele, "Deriving High-Resolution Urban Air Pollution Maps Using Mobile Sensor Nodes.," *Journal of Pervasive and Mobile Computing, Elsevier*, 2015.
- [3] T. Arn, *Healthy Navigation*. Semester thesis, ETH Zurich, 2014.
- [4] S. S. Skiena, *The Algorithm Design Manual*. London: Springer London, 2008.
- [5] J. H. Freidman, J. L. Bentley, and R. A. Finkel, "An Algorithm for Finding Best Matches in Logarithmic Expected Time," *ACM Transactions on Mathematical Software*, vol. 3, pp. 209–226, Sept. 1977.
- [6] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to algorithms*. 2001.
- [7] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, Dec. 1959.
- [8] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM*, vol. 34, pp. 596–615, July 1987.
- [9] T. Ikeda, H. Imai, S. Nishimura, H. Shimoura, T. Hashimoto, K. Tenmoku, and K. Mitoh, "A fast algorithm for finding better routes by AI search techniques," in *Proceedings of VNIS'94 - 1994 Vehicle Navigation and Information Systems Conference*, pp. 291–296, IEEE.

BIBLIOGRAPHY

- [10] A. V. Goldberg and C. Harrelson, "Computing the shortest path: A search meets graph theory," pp. 156–165, Jan. 2005.