



GPS-BASED NAVIGATION  
IN  
STATIC AND DYNAMIC ENVIRONMENTS

by  
Shahid Jabbar  
Institut für Informatik  
Universität Freiburg  
Freiburg, Germany

Supervisor: PD Dr. Stefan Edelkamp (Universität Dortmund)  
Co-supervisor: Prof. Dr. Th. Ottmann (Universität Freiburg)

*Submitted in partial fulfillment of  
the requirements for the degree of  
Master of Science in Applied Computer Science*

December 1, 2003



Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

**Shahid Jabbar**, Matrikelnummer: 1314192,  
Freiburg, den 1. Dezember 2003.



# Preface

*I have now the proof that the route I take to come to the university from my home is both the shortest and the quickest one.*

Maps available in the market are very expensive for a normal user and are static in the sense that it is not possible to update them due to the construction of a new street or a road. Also, it is not possible to query them for a route that is to be traveled at a particular point of time.

This thesis presents a novel approach that allows the construction of maps from Global Positioning System (GPS) trajectories and efficient computational geometry algorithms. Given a set of GPS points, the input is refined by geometric filtering and rounding algorithms. For constructing the graph from these GPS points and the according point-localization structure, fast scan-line and divide-and-conquer algorithms are used.

For accelerating the search algorithms, the geometrical structure of the graph is exploited in two ways. The graph is compressed while retaining the original information for unfolding resulting shortest paths. It is then annotated by refined topographic information: e.g., by the bounding boxes of all shortest paths that start with a given edge.

The sudden changes in the road network: e.g., a road accident or a traffic jam can affect the topology of the graph by increasing the travel time along the edges. This in turn can affect the pre-computed information. This thesis presents a novel approach for introducing dynamics in the system by defining a whole area as affected. Two approaches for dealing with navigation in a dynamic environment are presented.

**Keywords:** GPS, Navigation System, Computational Geometry, AI Planning, Dynamic Shortest Paths.

## Acknowledgments

After *Allah* the Almighty, there are 7 people behind what I am right now - my parents, who brought me from the skies to this world, Prof. Susanne Albers, Dr. Stefan Edelkamp, Prof. Th. Ottmann, and Dr. Abbas K. Zaidi, who put me on the way to the skies, and Karolina Sjöberg, who is continuously accelerating my flight. I am deeply grateful to all of them.

I am also very thankful to my friends: Salman, Ankit and Arun - thanks for your company, the long sessions of cooking and discussions, Zafar - *kardish* - thanks for the Turkish music, and Kiril - thanks for the Bulgarian cooking and music. Alberto - my *amigo* - deserves a special thank because of helping me in both scientific and personal matters. Many thanks to Shahid Hussain for proofreading a part of this thesis. My apologies to the readers for the remaining errors - I am far from being perfect.

I dedicate this thesis to Dr. Stefan Edelkamp and Karolina Sjöberg - the *fastest brains* I have ever met.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Data Collection and Preprocessing . . . . .	2
1.2	Navigation in Static Environment . . . . .	3
1.3	Navigation in Dynamic Environment . . . . .	4
1.4	Implemented System - <b>GPS-Route</b> . . . . .	5
1.5	Contributions . . . . .	5
1.6	Possible Future Extensions . . . . .	6
<b>2</b>	<b>Data Collection</b>	<b>7</b>
2.1	Geodetic Coordinates . . . . .	7
2.2	GPS . . . . .	7
2.3	Accuracy of GPS Data . . . . .	9
2.4	GPS data format . . . . .	9
2.5	Summary . . . . .	10
<b>3</b>	<b>Geometric Filtering and Rounding</b>	<b>11</b>
3.1	Geometric Filtering - Kalman Filter . . . . .	11
3.2	Geometric Rounding - Douglas-Peucker Algorithm . . . . .	12
3.3	Experimental Results . . . . .	14
3.4	Summary . . . . .	14
<b>4</b>	<b>Graph Construction and Node Localization</b>	<b>15</b>
4.1	Graph Construction . . . . .	15

4.2	Node Localization . . . . .	16
4.3	Experimental Results . . . . .	18
4.4	Summary . . . . .	19
<b>5</b>	<b>Graph Compression</b>	<b>21</b>
5.1	Compression Algorithm . . . . .	21
5.2	Partition of a Path . . . . .	24
5.2.1	Prefix Path $\Pi_{pre}$ Computation . . . . .	25
5.2.2	Postfix Path $\Pi_{post}$ Computation . . . . .	25
5.3	Decompression . . . . .	26
5.4	Experimental Results . . . . .	26
5.5	Summary . . . . .	27
<b>6</b>	<b>Navigation in Static Environment</b>	<b>29</b>
6.1	Static Models . . . . .	30
6.1.1	The Basic Model . . . . .	30
6.1.2	The Time Model . . . . .	30
6.1.3	The Absolute Time Model . . . . .	30
6.2	Dijkstra's Shortest Path Algorithm . . . . .	31
6.2.1	Dijkstra's Algorithm for the Basic Model . . . . .	32
6.2.2	Dijkstra's Algorithm for the Time Model . . . . .	32
6.2.3	Dijkstra's Algorithm for the Absolute Time Model . . . . .	33
6.3	Heuristic Search - A* . . . . .	34
6.3.1	A* for the Time Model . . . . .	37
6.3.2	A* for the Absolute Time model . . . . .	39
6.4	Geometric Pruning . . . . .	39
6.4.1	Bounding Box Pruning for the Basic Model . . . . .	41
6.4.2	Bounding Box Pruning for the Time Model . . . . .	42
6.4.3	Bounding Box Pruning with A* . . . . .	42
6.5	Re-initialization of Shortest Distance Values . . . . .	43
6.6	Experimental Results . . . . .	44

6.7 Summary . . . . .	45
<b>7 Navigation in Dynamic Environment</b>	<b>47</b>
7.1 Problem Characterization . . . . .	48
7.2 Dynamic Models . . . . .	48
7.2.1 Individual Edge Dynamics . . . . .	49
7.2.2 Disturbances as Geometrical Objects . . . . .	49
7.3 Affects of Disturbances on Containers . . . . .	50
7.4 Summary . . . . .	53
<b>8 Graph Update</b>	<b>55</b>
8.1 Graph Update in Individual Edge Dynamics . . . . .	55
8.2 Graph Update in Geometrical Objects . . . . .	58
8.3 Summary . . . . .	62
<b>9 Exploration-Time Checking</b>	<b>63</b>
9.1 General Search Strategy . . . . .	64
9.2 Exp.-Time Checking in Individual Edge Model . . . . .	64
9.3 Exp.-Time Checking in Geometrical Objects . . . . .	66
9.4 Summary . . . . .	69
<b>10 Architecture of GPS-Route</b>	<b>71</b>
10.1 General design . . . . .	71
10.2 Interface . . . . .	72
10.2.1 Standard Interface . . . . .	72
10.2.2 VEGA Interface . . . . .	73
10.2.3 Network Interface . . . . .	74
10.3 Preprocessor . . . . .	75
10.4 Shortest-Path-Algorithm . . . . .	76
10.5 Summary . . . . .	77



# List of Figures

1.1	Flowchart of Preprocessing. . . . .	3
1.2	Types of Dynamics . . . . .	4
2.1	The location of a point on earth. . . . .	8
2.2	Visualization of a GPS trace. . . . .	10
3.1	Kalman filter for integration of GPS and INS information . . . . .	12
3.2	Working of the Douglas-Peucker line-simplification algorithm. . . . .	13
4.1	A Voronoi diagram. . . . .	17
5.1	Working of the graph compression algorithm. . . . .	23
5.2	Partition of a path . . . . .	24
7.1	Affect of Disturbance on Bounding Boxes . . . . .	52
8.1	Conversion of segments to rectangles . . . . .	60
9.1	Exploration time checking. . . . .	67
9.2	Types of disturbances. . . . .	68
10.1	CGI-based light-weight interface of <b>GPS-Route</b> . . . . .	72
10.2	Visualization of a trace using VEGA. . . . .	73
10.3	Network Interface . . . . .	74
10.4	Class diagram of <b>Preprocessor</b> . . . . .	75



# Chapter 1

## Introduction

In the present world of increasing road-traffic and fast-growing transport infrastructure, navigation is an ubiquitous task. Electronic maps available in the market are very expensive for a normal user and are static in the sense that it is not possible to update them on the construction of a new street or a road. Sometimes due to a sudden change in the path suggested by a route finding system a second route has to be established. In the usual case, we need a large infrastructure for monitoring disturbances to cope with the changes in the map.

Also, it is not possible to pose timed queries to those maps i.e., to request a shortest path from a source to a destination during a specific period of time or day. The need for timed queries arises from the observation that the travel time can vary drastically during different types of days: workdays and holidays. Travel time can also vary during different time periods of a day - there are more cars on the streets during 8 to 9 AM than during 10 to 11 PM.

This report discusses a novel approach for the design of a navigation system for mobile objects, be it cars, bicycles or hikers. Our approach allows people to efficiently construct their own maps with the help of a GPS (Global Positioning System) device. The GPS points are collected using a GPS device connected to a portable computer e.g., a palm- or a laptop. Efficient computational geometry algorithms are used to construct maps from these GPS points. These maps can then be queried for a shortest/fastest path between a source and a destination point. In order to facilitate a collaborated map construction, we allow the facility of integrating different GPS points to construct a common map that can be maintained on a server and shared between multiple clients. These maps can also be visualized along with the resulting shortest path using a client/server visualization interface called VEGA (Visualization of Efficient Geometric Algorithms).

A traffic jam or some other disturbances can increase travel time along the affected roads. These disturbances are incorporated by an offset to the weights of

the affected edges of the underlying graph. Computational geometry algorithms and data structures efficiently store and keep track of the affected edges in order to cancel the influence of a disturbance after a specific time.

We have divided the discussion into four parts: Data collection and preprocessing, navigation in a static environment, navigation in a dynamic environment and architecture of the implemented system. In the following we provide an overview of these parts along with the references to the chapters, where they are discussed in detail. Readers are also referred to [11] for an overview of our approach.

## 1.1 Data Collection and Preprocessing

GPS is a mechanism to determine the coordinates of an object relative to the earth in 3-space. We collected GPS points by mounting a GPS device to a bicycle and then riding it on the path whose map was required. Chapter 2 presents a brief introduction to GPS.

GPS point are prone to inaccuracies due to various environmental factors. These inaccuracies need to be filtered out from GPS data. The standard in GPS data filtering is to use Kalman filter [19] to integrate inertial information collected from another source. As the second source, we utilized the speed information collected from the speed-o-meter of the mobile object.

To decrease the number of GPS points and to smoothen the shape of a trace in order to bring it close to the representation of a road, a geometrical filtering algorithm called Douglas-Peucker line-simplification [8] is used. It approximates a given poly-line by one with fewer points. Filtering and rounding of GPS traces are discussed in Chapter 3.

GPS data is converted to a graph representation by transforming the points to the nodes of the graph and imagining a straight edge between two consecutive points. The layout of the nodes is borrowed from the longitude and latitude of GPS points. But this leaves space for the calculation of the intersections of traces that actually represent road-crossings. We use the Bentley-Ottmann sweep-line algorithm [2] for this purpose.

Since query points might not have been visited and hence not be on the map, we suggest to start the search from the points that are closest to the query. A point localization structure is devised to assist the nearest neighbor search. We use a Delaunay triangulation [6] of the nodes to answer closest point queries efficiently. Chapter 4 discusses graph construction process and point localization in detail.

A careful look at the problem reveals that the most important vertices of the graph are either the ones that mark the start and end of a trace, or the ones that define the crossings. This analysis is transformed into a compression algorithm

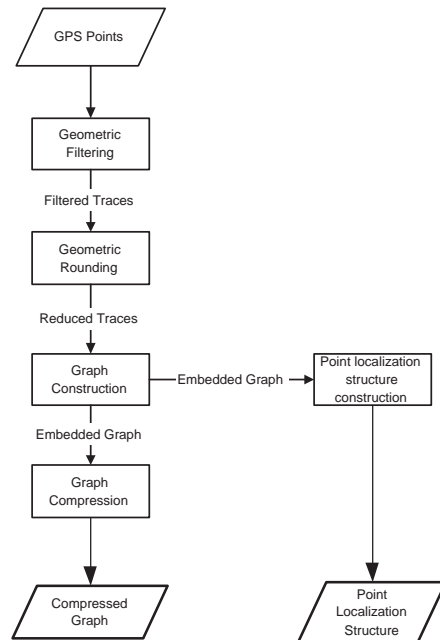


Figure 1.1: Flowchart of Preprocessing.

in Chapter 5, which merges all nodes having an *in-degree* and *out-degree* of 1.

Figure 1.1 summarizes the preprocessing procedure in a flowchart along with the inputs and outputs of individual steps. Data is shown with parallelograms and processes are shown with rectangles.

## 1.2 Navigation in Static Environment

Once the graph is constructed and compressed, it is ready to be used to answer shortest path queries. In Chapter 6 we first formalize the models of searching in a static setting. The graph can be queried not only for the shortest path but also for the fastest path or a weighted combination of both. The standard in shortest path searching, Dijkstra’s algorithm [7], has been adapted to our models. The search can be accelerated by using domain-dependent knowledge to guide the exploration. We use A\* in this context.

Since the frequency of queries is much higher than any update in the topology of the graph, the search can be accelerated by pre-computing some of the information needed for search. We coin the terms off-line and on-line processing. Before actually answering the queries, the graph is annotated by geometric

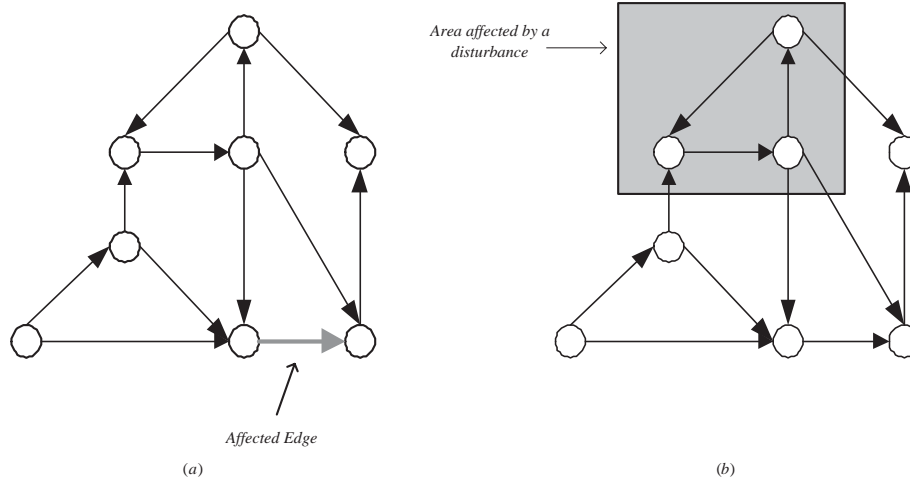


Figure 1.2: Types of Dynamics

containers as an off-line processing to answer queries [29]. These containers are associated with each edge and contain at least all the nodes that can be reached on a shortest path originating from that particular edge. Containers can be computed by running an all-pairs shortest path algorithm on the graph.

The frequent on-line queries can then be answered efficiently by using only the edges that contain the target node in their associated containers. Chapter 6 discusses the developed algorithms for navigation in a static environment.

### 1.3 Navigation in Dynamic Environment

Due to a road accident or some other disturbance, a road can become unusable or the travel time along that road can increase. This situation is modeled as an increase in the edge weight of the graph. Due to this increase, some of the pre-computed information becomes unusable. Chapter 7 presents two different models for introducing dynamics in our problem (see [10] for an overview).

In the first model, due to a disturbance an edge is affected individually and its weight is increased (see Figure 1.2-a). Sometimes, however, it is not possible to pin-point the exact location of the affected road. A whole area could be affected e.g. due to heavy snow. Hence, in the second model we define a disturbance as a geometrical object on a separate layer on top of the graph and affecting all the edges underneath it (see Figure 1.2-b). In the presence of these disturbances, we characterize our pre-computed information as **valid** or **invalid** for a particular query.

In Chapter 8 we discuss an approach for dealing with dynamics in which, due to a disturbance, the graph is updated by increasing the weights of the affected edges. The search is then invoked in the updated graph. The search space is pruned only if the pre-computed information is valid for that particular query.

In contrast to the first approach, Chapter 9 discusses the possibility of a delay for the update of edge weights until the edges are actually used during exploration. The main motivation behind this approach is that it is possible that some of the disturbances have disappeared by the time a mobile object reaches the affected area in reality.

## 1.4 Implemented System - GPS-Route

We have implemented the presented approach up to the static part in a software system called *GPS-Route*. It relies on the *LEDA* [22] library for stable implementations of computational geometry algorithms. Chapter 10 discusses the architecture of the software. For the visualization purposes we have used a client/server based system called *VEGA* (Visualization of Efficient Geometric Algorithms) [17]. It provides the facility of visualizing the graphs and shortest paths through a Java-Applet based web-client.

We have also designed a CGI-based light-weight interface. Given a set of traces and query points, it returns the searched shortest paths in a format that can be uploaded to a GPS device for navigation.

## 1.5 Contributions

An approach for generating a map from GPS traces that can be queried for shortest/fastest paths, is presented. This thesis effectively combines traditional computational geometry algorithms with artificial intelligence and graph algorithms. Following are the novel achievements in this thesis.

- A vehicle-independent approach for map generation from GPS traces.
- A linear time compression algorithm for graphs that also preserves the layout of the graph.
- A linear time decompression algorithm for shortest paths.
- Proof of the optimality of A\* algorithm in the case of compressed graph.
- Absolute Time Model in static setting that allows to pose timed-queries to the graph.
- An unconventional dynamic model that allows to define a whole area as affected by a disturbance.

- Concrete characterization of the unaffected precomputed information in the presence of disturbances.
- Two different approaches for dealing with dynamics: Graph update and Exploration-time checking that utilize computational geometry algorithms to effectively utilize the unaffected information.
- An implemented system of the discussed approach, up to the basic static model, along with a CGI-based light weight interface and VEGA-based interface for visualization.

## 1.6 Possible Future Extensions

In the following, we present some of the possible extensions and related references that could serve as the starting points in those directions.

- Visualization of traces on a topographical map for easier navigation. A starting point in this direction is the decoding of topographical maps provided by Surveying Authorities of the States of the Federal Republic of Germany<sup>1</sup>.
- Management of secondary memory issues that can arise if the graph size is larger than the main memory. In [9] and [21], these issues are addressed in the context of search algorithms.
- We approximated a compressed edge by a straight edge. The other option is to use curved edges to better approximate a compressed edge. These issues are addressed in the EXACUS project<sup>2</sup> at MPI, Saarbrücken.
- The bridges on top of a road call for a navigation system in a three-dimensional environment. In the WITAS<sup>3</sup> project at Linköping University, Sweden, efforts are being going on for the design of a 3D navigation system.

---

<sup>1</sup><http://www.adv-online.de/english/products/index.htm>

<sup>2</sup><http://www.mpi-sb.mpg.de/projects/EXACUS>

<sup>3</sup><http://www.ida.liu.se/ext/witas/>

## Chapter 2

# Data Collection

Data gathering can be done using a GPS device mounted on any mobile vehicle. In our case, we have installed a GPS device on a bicycle, hence providing a very cost-effective solution. This section proceeds by first explaining the Geodetic coordinate system and then the GPS mechanism. An example of the collected data set along with its visualization is shown in the last section.

### 2.1 Geodetic Coordinates

The position of an object in the earth's sphere can be described in terms of Geodetic coordinates consisting of its longitude, latitude, and height (see Figure 2.1).

- *longitude* of a point is the angle from the plane of the *prime meridian* to a plane passing through the point, both planes being perpendicular to the Equator.
- *latitude* of a point is the angle from the equatorial plane to the vertical direction of a line normal to the earth's ellipsoid and passing through the point.
- *height* is the distance between the point and the earth's ellipsoid.

### 2.2 GPS

GPS is a mechanism for finding out the Geodetic coordinates of an object relative to the earth in 3-space. A set of 24 satellites that are orbiting around the earth are used to find the position of an object through a GPS receiver.

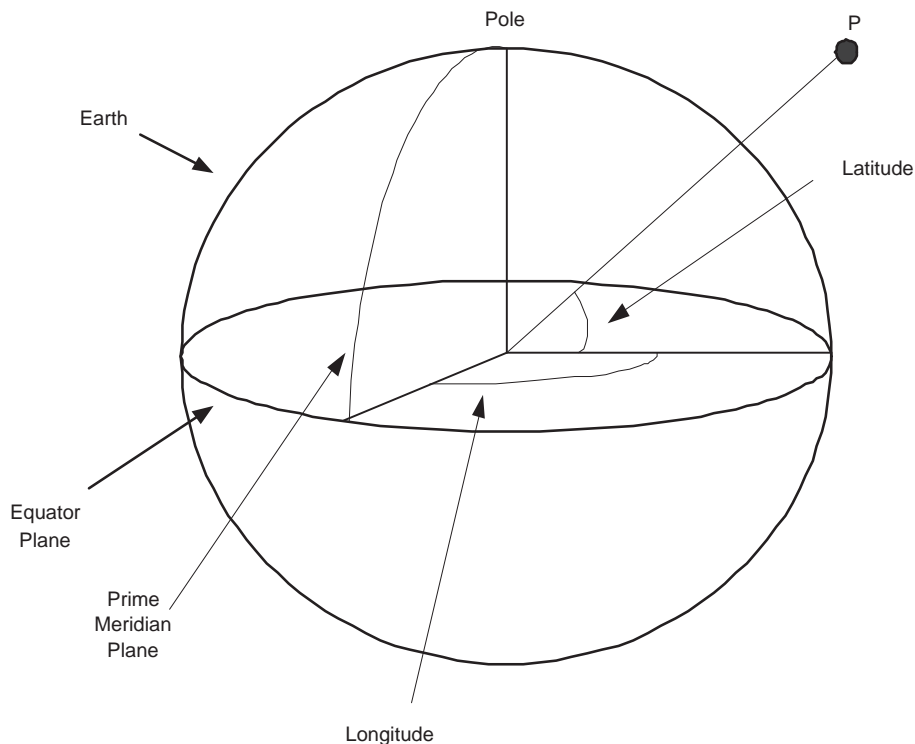


Figure 2.1: The location of a point on earth.

A GPS receiver first tries to find the maximum number of satellites around the user. Then it computes the distance between satellites and user by the time taken for the signals to reach from satellites to the user's device. Each satellite induces a sphere in its coverage region - with itself being the center of the sphere - and gives a set of points in 3-space where the user might be. The intersection of three spheres plus the earth's sphere gives the actual position of the user. The more satellites visible, the more accurate the data. This process is called *trilateration* [5].

The device can be connected to a laptop or to a palmtop through the RS-232 interface. When connected to an external machine the GPS device emits continuous signals at a user-defined frequency. These signals can be collected using different softwares. We used the Fugawi<sup>1</sup> software by Fugawi corporation for our purpose. This software provides several other features such as exporting GPS data to different formats.

---

<sup>1</sup><http://www.fugawi.com>

## 2.3 Accuracy of GPS Data

GPS is an outcome of the cold war era. Two major technologies emerged at that time. One is this **GPS** - designed and maintained by United States and the other one is **Glonass** - designed by Soviet Union.

GPS was later permitted to be used by civilians. But because of its military importance the data provided to the civilians was not very accurate and the United States Department of Defense has all the rights to turn off the satellite coverage at any time. The current accuracy figures for the basic GPS are [5]

- 100 meter horizontal accuracy,
- 156 meter vertical accuracy, or
- 340 nanoseconds time accuracy.

In the later times, base stations were set up all around North America to enhance the accuracy of GPS data for civilian use, and the technologies such as Differential-GPS (DGPS) and Wide Area Augmenting System (WAAS) came into being. These technologies provide an accuracy of less than 3 meters.

In Europe, navigation relies on the old GPS system and cannot make use of DGPS and WAAS technologies. The European Union has now started a project called **Galileo** for this purpose (See [31] for more details). It will be operational by 2008 and will constitute of a net of 30 satellites and base stations giving highly accurate data of up to 1 meter accuracy.

## 2.4 GPS data format

The original data format from the GPS follows NMEA (National Marine Electronics Association) standard. It constitutes of about 20 different attributes, out of which Geodetic coordinates, UTC (Universal Coordinated Time) and date are essentially important for our purpose. These attributes are exported in the plain text format using the Fugawi software.

A typical example of data collected from GPS device is in the following form:  
*<longitude>*, *<latitude>*, *<date>*, *<time>* e.g.

```
48.0070783,7.8189867,20030409,100156
48.0071067,7.8190150,20030409,100158
48.0071850,7.8191400,20030409,100200
48.0071650,7.8191817,20030409,100202
48.0071433,7.8191867,20030409,100204
48.0071383,7.8191883,20030409,100206
48.0071333,7.8191917,20030409,100208
48.0071317,7.8191917,20030409,100212
```

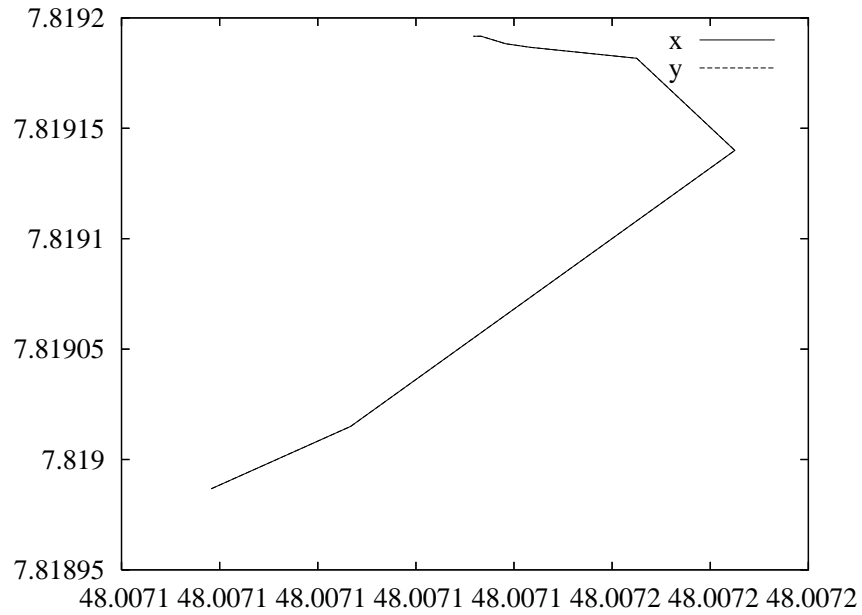


Figure 2.2: Visualization of a GPS trace.

Figure 2.2 shows the visualization of the above trace using GNUPlot.

For experiments, we collected four different data sets. Table 2.1 shows the number of points in individual data sets and how data is collected.

#GPS points	how collected ?
1,277	Bicycle
1,706	Bicycle
2,365	Hiking
50,000	Car (taxi)

Table 2.1: Collected Data Sets.

## 2.5 Summary

An overview of Geodetic coordinate system is presented along with an overview of GPS mechanism and its current accuracy figures. A GPS device connected with a palm- or laptop can be used to gather GPS data. We devised a very cost effective solution by installing a GPS device along with a palmtop on a bicycle.

## Chapter 3

# Geometric Filtering and Rounding

GPS data is very prone to errors. Any change in the surroundings like an unclear sky or high rise buildings can decrease the accuracy of the GPS data. In order to get a more accurate estimate of the positions it needs to be filtered from outliers. This chapter introduces the Kalman filter for the purpose of filtering the traces collected.

Since the points are gathered at a frequency of 1 Hz, there is a need to reduce the number of points while still having an acceptable trajectory approximating the road geometry. In Section 3.2 we discuss the Douglas-Peucker algorithm for rounding GPS traces and present results on the collected data.

### 3.1 Geometric Filtering - Kalman Filter

The most popular method in GIS community is to use Kalman filter. The main idea behind Kalman filter is to use an extra data input like an Inertial Information System (INS)<sup>1</sup> to remove the outliers [20]. A typical INS consists of accelerometers for the measurement of acceleration and gyroscopes for the measurement of rotational accuracy.

Kalman filter uses statistical models to provide a better estimate of current position based on the past GPS data and extra input. It is basically a recursive process that, starting with an initial position estimate and covariance, combines it with the current estimates to produce a more accurate estimate. Based on new measurements the covariance for the next recursive cycle is updated. Figure 3.1 [20] describes the integration process of GPS with INS information.

---

<sup>1</sup>See <http://www.electronic-engineering.ch/study/ins/ins.html> for an INS design

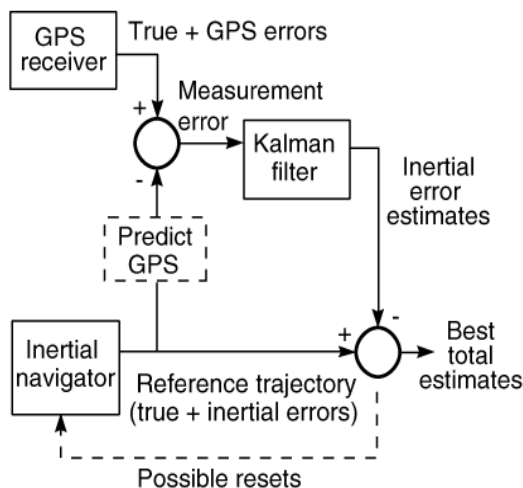


Figure 3.1: Kalman filter for integration of GPS and INS information

For a more detailed treatment of Kalman filter, we refer the reader to [19] and [20]. In [14], the authors have provided an application of Kalman filtering combined with the Markov model in robot navigation resulting in better position estimates.

## 3.2 Geometric Rounding - Douglas-Peucker Algorithm

This section gives an overview of the Douglas-Peucker [8] algorithm for line simplification. Given a polyline of  $n$  points  $p_1, p_2, \dots, p_n$ , *Douglas-Peucker* searches for an approximate polyline with fewer points. The algorithm can best be described recursively: to approximate the polyline from  $p_i$  to  $p_j$  the algorithm assumes a segment  $p_i p_j$ . If the farthest vertex  $p_k$  from this segment has a distance smaller than a given threshold  $\theta$ , the algorithm accepts this approximation. On the contrary, if the farthest vertex  $p_k$  is  $\theta$  or more distance apart, the segment is split into two segments  $p_i p_k$  and  $p_k p_j$ . Both of these segments are then recursively approximated.

Figure 3.2 demonstrates the working of the Douglas-Peucker algorithm on a polyline of 4 vertices. In the first step vertex 3 turned out to be the split vertex as its distance from the line joining vertices 1 and 4 is more than  $\theta$ . Vertex 3 is then removed in the second step because its distance from the line 1-3 is less than  $\theta$ .

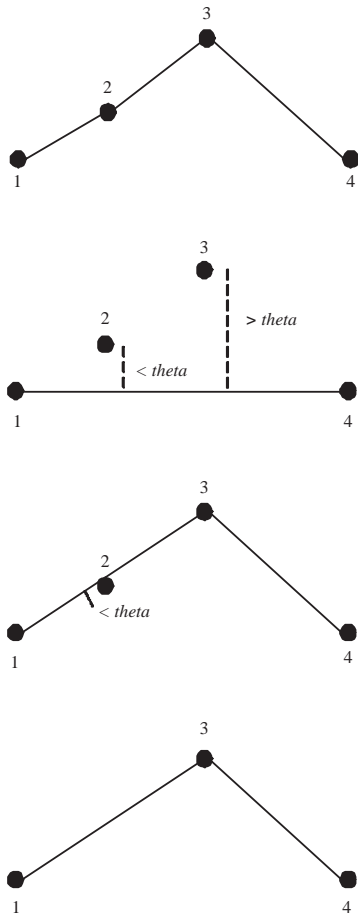


Figure 3.2: Working of the Douglas-Peucker line-simplification algorithm.

The best case of Douglas-Peucker algorithm is when the splitting vertex is at the center of the chain giving us a recurrence

$$D(1) = 0 \tag{3.1}$$

$$D(n) = n - 1 + D\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + D\left(\left\lceil \frac{n}{2} \right\rceil\right) \tag{3.2}$$

This gives us a running time of  $O(n \log n)$ .

The worst case is when the splitting vertex is the second vertex in the chain from either of the corner vertices. This gives us the recurrence

$$D(1) = 0 \tag{3.3}$$

$$D(n) = n - 1 + D(1) + D(n - 1) \tag{3.4}$$

This recurrence gives us a solution of  $\Theta(n^2)$ .

The main reason for this worst case running time is the search for the splitting vertex. Hershberger *et al.*'s implementation [16] exploits the geometrical structure of the points to find the splitting vertex and hence achieving a bound of  $O(n \log n)$  time. The core idea of their search strategy is to search the splitting vertex only on the convex hull of the chain.

### 3.3 Experimental Results

Table 3.1 shows the results of using the Douglas-Peucker algorithm to reduce the GPS traces. Our experience is that with  $\theta = 10^{-7}$ , the reduced data is sufficient to capture the road geometry.

#points	$\theta = 10^{-7}$	$10^{-6}$	$10^{-5}$	$10^{-4}$	$10^{-3}$
1,277	766	558	243	77	22
1,706	1540	1162	433	117	25
2,365	2083	1394	376	28	7
50,000	48,432	42,218	17,853	4,385	1,185

Table 3.1: Reduction of traces with Douglas-Peucker.

### 3.4 Summary

Because of the environmental disturbances, GPS data needs to be filtered. GPS data can be filtered from the outliers by combining it with an external source through Kalman filter. Kalman filter exploits statistical information collected from both the sources for filtration.

To reduce the number of GPS points, we exploited Douglas-Peucker line-simplification algorithm. The procedure approximates a given poly-line of  $n$  points by fewer points in time  $O(n \log n)$ . By choosing a proper value for the approximation threshold, we have managed to reduce the number of points along with an acceptable representation of road geometry.

## Chapter 4

# Graph Construction and Node Localization

The GPS traces have to be converted into a graph representation for a rigorous treatment through different graph algorithms like searching for the shortest/fastest path between two points. Since we do not gather data of all the points in the Euclidean space, a user could query for a shortest path between the points that are not in the existing GPS traces. This motivates the use of efficient node localization techniques to search for the points in the existing data that are closest to the query points.

We start this chapter with a discussion of a geometric algorithm that converts GPS traces into a graph representation. Section 4.2 presents the node localization structure. Finally, in Section 4.3, the experimental results for these steps are presented.

### 4.1 Graph Construction

The traces themselves have a natural transformation to graphs by treating points as vertices and considering a straight line edge between two consecutive nodes. As far as the axes are concerned, we mapped longitude to the  $x$ -axis and latitude to  $y$ -axis - a natural mapping. But some of these traces might be overlapping or intersecting. These intersections might be due to road crossings. In case we integrate all of the traces just as they are, these crossings would be lost.

This problem can be formalized as the line segments intersection problem i.e., given  $n$  line segments, find out all the points where these lines intersect. In our application, we use the Bentley-Ottmann sweep-line algorithm [2] that reports all the intersection points in time  $O((n+k) \log n)$ , where  $k$  is the number

of intersections.

Initially we construct a graph with the GPS points as nodes and making an edge between the two consecutive trace points. The positions of points are inserted in the information structure attached with the corresponding nodes. With each segment we attach the length of that edge in the Euclidean space as the weight of that edge, and the start and end time when that edge was traversed. The reason why we do not save time information with the nodes is to allow multiple edges between two nodes. This can be the case when an edge is visited more than once but during different times.

The graph  $G$  can now be defined as a 4-tuple,  $G = (V, E, d, T)$  where  $V$  is the set of vertices,  $E$ , the set of edges. The function  $d : E \rightarrow \mathbb{R}$  assigns with each edge, the Euclidean distance between the source and target vertex of the edge. The function  $T : E \rightarrow Time \times Time$  provides the temporal information about the edge, precisely - the date and time when that edge was traversed.

The algorithm proceeds by sweeping a vertical line through the whole graph and maintaining two data structures: *line status*, that contains the segments currently being intersected by the sweep-line, according to the increasing  $y$ -coordinate, and *event queue* defining the halting points of the sweep line. Initially all the start and end points of the segments are inserted in the *event queue* in order of increasing  $x$ -coordinate. The line is then swept across the traces stopping at the start and end points of lines, and inserting the starting lines in the *line status* and deleting the ending lines. Intersection of the lines that are currently in the *line status* are then computed and inserted in the *event queue* to serve as the next halting points for the sweep line. When the sweep line reaches such an intersecting point, the order of the intersecting lines is reversed in the *line status* structure.

While sweeping, new vertices - representing the intersection points - are added to the graph. Let  $v_{in}$  be a new vertex added as the intersection of two segments  $e_1 = (u, v)$  and  $e_2 = (w, x)$ . Then  $T(x_{in})$  is calculated through the velocity of the mobile object from either through  $e_1$  or  $e_2$  (for simplicity, we assume a uniform velocity through an edge and equal velocities through  $e_1$  and  $e_2$ ) and the position of the new vertex.

The result is a Euclidean graph satisfying the *triangular property* i.e. if  $(i, k)$  is an edge from vertex  $i$  to vertex  $k$  then for all vertices  $j$ , we have  $d(i, k) \leq d(i, j) + d(j, k)$ .

## 4.2 Node Localization

Imagine the scenario in which a user queries the graph for a shortest path from a source to a target point, but where the points do not correspond to any node in the constructed graph. This might be the case when those points are never

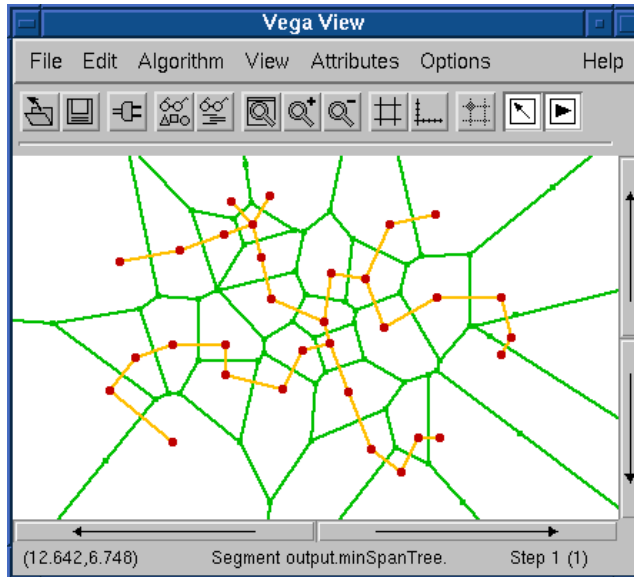


Figure 4.1: A Voronoi diagram.

visited and hence their positions are never recorded. It can also be the case that the user's supplied query points though lying on a street that is already traversed, but because of imprecision in GPS data, do not match with any existing node.

An obvious solution to this problem is to start the search from the nodes whose locations are closest to the query points. The naive algorithm performs a linear search on the existing nodes to find those closest nodes. But in large graphs, like in our case, a linear search for every query is not feasible.

Voronoi diagrams provide an efficient solution to this problem. A Voronoi diagram for a set of points  $P$  in Euclidean space gives a subdivision corresponding to every point. A given point  $q \notin P$  is closest to the point  $p \in P$  if  $q$  lies in the subdivision of  $p$ . Figure 4.1 shows a Voronoi diagram of a set of points.

A Voronoi diagram can be constructed in time  $O(n \log n)$  using a divide-and-conquer strategy [25]. The algorithm divides a set of points from its median according to the  $x$ -coordinate and constructs the subdivisions for both the sets. The two subdivisions are then merged together to obtain a common subdivision. This running time is optimal, because the problem of sorting numbers can in fact be linearly reduced to the computation of a Voronoi diagram [6].

In our application, we have used the dual graph of Voronoi diagram, called Delaunay Triangulation. A Delaunay Triangulation can be defined as a triangulation of a set of points  $P$  such that a circumcircle around the three corners

#GPS Points	#Graph Nodes	Time
1,277	1,473	0.42
1,706	1,777	0.27
2,365	2,481	0.37
50,000	54,267	11.13

Table 4.1: Results of Graph Construction.

of a triangle does not contain any other point. A Delaunay triangulation can be obtained from a Voronoi diagram by making an edge between the points that share a common subdivision boundary.

Delaunay Diagram can be constructed using different algorithms (see [27] for a comparison of different algorithms for the construction of Delaunay triangulation). We have used the LEDA [22] implementation of a randomized incremental algorithm for our purpose [13]. Starting with a triangulation of three points, it inserts new points selected randomly. Edge flipping converts the resulting diagram into a legal Delaunay triangulation. This strategy results in an  $O(n \log n)$  expected running time.

The nearest neighbor query is answered by inserting the query point in the diagram and converting the resulting diagram back to a Delaunay diagram using edge flipping. The nearest point is then one of the adjacent points to the query point. The query point is then removed from the Delaunay diagram and the resulting diagram is converted back to a Delaunay diagram giving an expected query time of  $O(\log n)$ .

Allowing the addition of nodes in the graph requires a dynamic point localization structure. For a detailed treatment of dynamic point localization structures, we refer the reader to VoroGlide<sup>1</sup>, an applet, implementing dynamic Voronoi diagrams. A more detailed treatment of dynamic localization structure is given in [18].

In a sparse graph, using segments instead of points to construct the point localization structure can result in better localization strategy especially in road networks where a sparse graph represent long roads. In this case, an extended Voronoi diagram like in [3] or trapezoidal map [23] can be the candidates for a point localization structure.

### 4.3 Experimental Results

In Table 4.1, we present the results of the sweep-line algorithm on our set of example traces. Column 1 shows the actual number of GPS points. The total

<sup>1</sup><http://web.informatik.uni-bonn.de/I/GeomLab/VoroGlide/>

nodes in the embedded graph obtained after the Bentley-Ottmann sweep-line algorithm are presented in Column 2. Column 3 shows the time taken by the algorithm for the graph construction, measured in seconds. The output sensitive behavior of the algorithm i.e. the affect of  $k$ , the number of intersections, can be easily observed by comparing rows 1 and 2.

#points	#queries	Construction Time (sec.)	Searching Time (sec.)	Naive Searching Time (sec.)
1,277	1,277	0.10	0.30	12.60
1,706	1,706	0.24	0.54	24.29
2,365	2,365	0.33	1.14	43.3
50,000	50,000	13.73	14.26	>10,000

Table 4.2: Results of Node Localization

Table 4.2 presents the result of efficient node localization strategy. We posed as many queries as there were points by adding a small offset to the actual points. Column 3 shows the construction time for the Delaunay triangulation. The total query times for the number of queries in Column 2 is shown in Column 4. The effectiveness of this approach over the naive algorithm is established by comparing CPU times in Column 4 to Column 5.

## 4.4 Summary

Merging multiple traces requires us to calculate the intersection points of these traces. These intersection points represents the road-crossings in reality. We employed fast line-sweep algorithms to find the intersections of GPS traces. The algorithm reports all the intersection points in  $O((n + k) \log n)$  time, where  $n$  is the number of line segments and  $k$  is the number of reported intersections.

It is possible that the user provided query points do not correspond to any node in our existing graph. In this situation, we suggested to convert these query points to their nearest neighboring points in the graph. We utilized Delaunay triangulation to answer nearest neighbor queries in only  $O(\log n)$  time as compared to the naive search, requiring  $O(n)$  time. The algorithm we used to construct a Delanuay triangulation has an expected running time of  $O(n \log n)$ .



## Chapter 5

# Graph Compression

Once the graph is constructed we observe that there are a large number of nodes having exactly one in- and out-going edge. These nodes do not create any *choice* point in the exploration phase of a search algorithm. This motivates us to apply a compression scheme where the edges incident to these nodes can be merged into a single edge that starts and ends at intersection nodes or at the nodes representing the start or end points of a trace. The distance and time value of that new edge can then be made equal to the sum of the distance and time information of the intermediate edges.

The edges incident to the nodes of degree 2 cannot be deleted for good, or we will lose the original layout of the graph, which should be retained for two reasons: first, the user is not concerned with the compressed graph and is interested in having the answer to his/her query in the form of GPS points defining the exact path on the streets. Second, in the compressed graph, search can then be performed only among the intersection nodes or the end points of traces. These issues lead us to retain the information about the original graph and maintain a two-layered graph i.e. the compressed graph on top of the original graph.

Section 5.1 discusses the details of the compression algorithm and the information necessary to preserve before the compression in order to decompress the shortest path. In Section 5.2, we present the concept of path partitioning that facilitates the decompression of the searched path. The decompression of a path is discussed in Section 5.3.

### 5.1 Compression Algorithm

The compressed graph  $G_c = (V_c, E_c, d_c, T_c)$  of  $G = (V, E, d, T)$  can be defined as a layer on top of the original graph  $G$ , where  $V_c = \{v_c \in V \mid (indeg(v_c) = 1$

```

Input      : Graph  $G$ , Mapping  $\omega : E \rightarrow \{0, 1\}$ 
Output    : Compressed Graph  $G_c$ , Mapping  $\psi : E \rightarrow E$ 
for All nodes  $v \in G$  do
  if  $indeg(v) = outdeg(v) = 1$  then
     $e_1 \leftarrow InEdge(v)$ 
     $e_2 \leftarrow OutEdge(v)$ 
     $e \leftarrow new\ Edge(Source(v), Target(v))$ 
     $d(e) \leftarrow d(e_1) + d(e_2)$ 
     $\omega(e) \leftarrow false$ 
    if  $\omega(e_1) = true$  then
       $\psi(e) \leftarrow e_1$ 
      Hide ( $e_1$ )
    else
       $\psi(e) \leftarrow \psi(e_1)$  /* Inherit the value from the old edge */
      Delete ( $e_1$ )
    if  $\omega(e_2) = true$  then
      Hide ( $e_2$ )
    else
      Delete ( $e_2$ )

```

**Algorithm 1:** Compression Algorithm

and  $outdeg(v_c) = 0$ ) or ( $indeg(v_c) = 0$  and  $outdeg(v_c) = 1$ ) or ( $indeg(v_c) > 2$ ) or ( $outdeg(v_c) > 2$ )},  $E_c$  are the new compressed edges, and  $d_c$  - the weight function - for an edge  $e_c = (u_c, v_c) \in E_c$  is defined as the sum of the weights of all the intermediate edges  $(u_c = x_1, x_2), (x_2, x_3), \dots, (x_{k-1}, x_k = v_c)$ . i.e.,  $d_c(u_c, v_c) = \sum_{i=1}^{k-1} d(x_i, x_{i+1})$ . Let  $T_{start}, T_{end}$  denote the start and end time of an edge, respectively. The time information of the edge  $e_c$  is defined as  $T_{start}(e_c) = T_{start}(e_1)$  and  $T_{end}(e_c) = T_{end}(e_k)$ .

More precisely, the compression is a surjective mapping  $\phi : V \rightarrow V_c$ , so that  $e_c = (u_c, v_c) \in E_c$  if there exists a path  $p = \langle u = x_1, \dots, x_k = v \rangle$  and  $indeg(x_i) = outdeg(x_i) = 1$  with  $\phi(u) = u_c$  and  $\phi(v) = v_c$ .

In practice, compression is done through a linear time algorithm that proceeds by going through all the nodes of the graph and creating an edge  $e = (u, w)$  if there exists the edges  $e_1 = (u, v)$  and  $e_2 = (v, w)$  with  $indeg(v) = outdeg(v) = 1$ . If  $e_1$  or  $e_2$  are themselves results of a merging process, then they are deleted, otherwise they are marked *hidden* in order to be restored later. The information about the originality of an edge is obtained by a function  $\omega : E \rightarrow \{0, 1\}$  that returns *true* if the edge is original and *false* otherwise.

If  $e_c = (u_c, v_c)$  is an edge in the compressed graph then both  $u_c$  and  $v_c$  have degrees  $> 2$  (except when they are the start or end points of a trace). In order

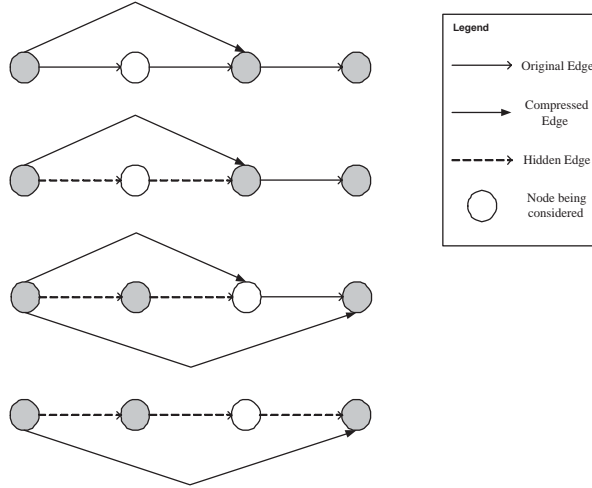


Figure 5.1: Working of the graph compression algorithm.

to decompress  $e_c$ , we need a handle to the correct hidden edge from  $u_c$  that can take us to  $v_c$ . For this purpose, during compression, we maintain a mapping  $\psi : E_c \rightarrow E$ , that when given  $e_c = (u_c, v_c)$ , returns the first hidden edge in the path from  $u_c$  to  $v_c$ .

This approach for compressing the graph reduces the space complexity from  $O(n + k)$  to  $O(l + k)$ , where  $l$  is the number of traces and  $l \ll n$ . Algorithm 1 presents the compression algorithm in detail. Figure 5.1 illustrates the compression process on a small graph segment.

The following result provides a lower bound on the distance values  $d_c$  of edges in the compressed graph.

**Lemma 5.1.1** *The distance value  $d_c$  of an edge  $e_c = (u_c, v_c) \in E_c$  is greater than or equal to the straight line distance from  $u_c$  to  $v_c$ .*

**Proof** Let there exist a path  $p = \langle \phi^{-1}(u_c) = x_1, x_2, \dots, x_k = \phi^{-1}(v_c) \rangle$  in  $G$ . There can be two cases:

1.  $k = 2$ : in this case, no compression occurred between the nodes  $u$  and  $v$ . Since from the construction of the graph we have  $d(e) = \|u - v\|_2$ , the lemma holds.
2.  $k > 2$ : in this case, the result clearly holds even if all the nodes  $x_i$ 's are collinear.

■

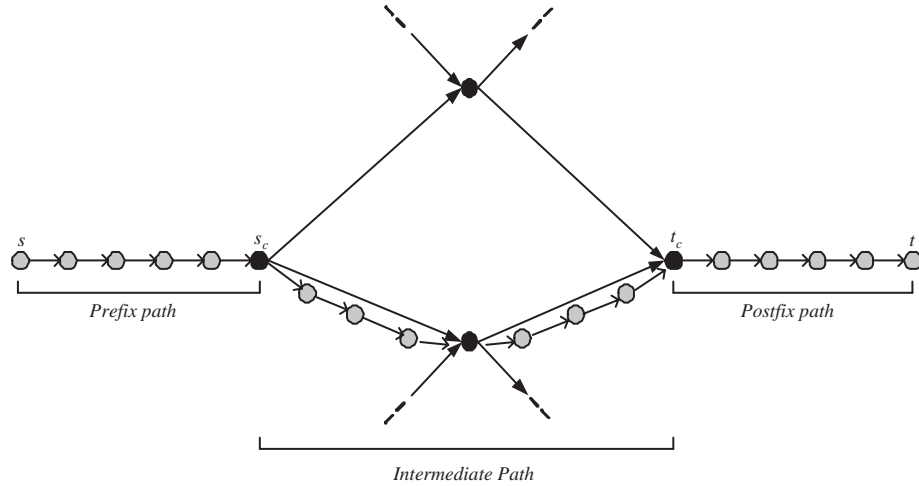


Figure 5.2: Partition of a path

**Proposition 5.1.2** *The triangular property does not hold in the compressed graph.*

**Proof** Immediately follows from Lemma 5.1.1

## 5.2 Partition of a Path

Due to the compression, we can only start the search from the intersecting points or the start and end points of a trace. So, whenever a query  $q = (s, t)$  is posed to the system, it is transformed into  $q_c = (s_c, t_c)$ , where  $s_c \in V_c$  is the nearest reachable vertex in  $G_c$  from  $s$ . Vertex  $s_c$  can be found out by a mere walk starting from the single outgoing edge from  $s$  until we reach a compressed vertex. If  $s \notin V$ , we first use *nearest neighbor search* to find out the vertex  $v \in V$  closest to  $s$ . The compressed vertex  $s_c$  is then searched with reference to  $v$ . The new target,  $t_c$  can then be defined conversely, except that it corresponds to the nearest compressed vertex while *backtracking* from  $t$ .

This leads to the partition of a path  $\Pi$  into a sequence of three sub-paths as  $\langle \Pi_{pre}, \Pi_{inter}, \Pi_{post} \rangle$ , where  $\Pi_{pre}$  - the *prefix* path - is the path from  $s$  to  $s_c$  and  $\Pi_{post}$  - the *postfix* path - is the path from  $t_c$  to  $t$ . The sub-path  $\Pi_{inter}$  is the shortest path from  $s_c$  to  $t_c$ . Note that there can exist only single  $\Pi_{pre}$  and  $\Pi_{post}$  with all the intermediate nodes having an in-degree and out-degree of 1 and hence no ambiguity can arise. (See Figure 5.2)

### 5.2.1 Prefix Path $\Pi_{pre}$ Computation

```

Input      : Graph  $G$ , Mapping  $\rho : V \rightarrow E$ , Vertex  $s$ 
Result    : Vertex  $s_c$ ,  $\Pi_{pre}$ 
 $s_c \leftarrow s$ 
while  $indeg(s_c) = 0$  and  $outdeg(s_c) = 0$  do
   $e \leftarrow \rho(s_c)$ 
  Unhide ( $e$ )
  Enqueue ( $\Pi_{pre}, e$ )
   $s_c \leftarrow \mathbf{Target}(e)$ 
Hide ( $e$ )

```

**Algorithm 2:** Prefix Path  $\Pi_{pre}$  Computation

The prefix path  $\Pi_{pre}$  is computed, as described earlier, by a mere walk starting from the  $s$  and ending at the nearest reachable compressed vertex. But the edges that can take us from  $s$  to  $s_c$  were marked hidden during compression. In order to restore these edges we need a handle to that edge, more precisely, a function that when given a vertex returns the outgoing edge of that vertex. This is accomplished by using a mapping  $\rho : V \rightarrow E$ . Algorithm 2 describes the process of  $\Pi_{pre}$  computation in details.

### 5.2.2 Postfix Path $\Pi_{post}$ Computation

```

Input      : Graph  $G$ , Mapping  $\sigma : V \rightarrow E$ , Vertex  $t$ 
Result    : Vertex  $t_c$ ,  $\Pi_{post}$ 
 $t_c \leftarrow t$ 
while  $indeg(t_c) = 0$  and  $outdeg(t_c) = 0$  do
   $e \leftarrow \sigma(t_c)$ 
  Unhide ( $e$ )
  Push ( $\Pi_{post}, e$ )
   $t_c \leftarrow \mathbf{Source}(e)$ 
Hide ( $e$ )

```

**Algorithm 3:** Postfix Path  $\Pi_{post}$  Computation

The postfix path  $\Pi_{post}$  is computed in a similar fashion except for the fact that here one needs to *backtrack* to reach the target  $t_c$ . For this purpose we make use of a mapping  $\sigma : V \rightarrow E$  that when given a vertex returns the incoming edge to that vertex. Algorithm 3 presents the details of the postfix path computation.

### 5.3 Decompression

<pre> <b>Input</b>      : Graph <math>G</math>, Mappings <math>\omega : E \rightarrow \{0, 1\}</math>, <math>\sigma : V \rightarrow E</math>, <math>\psi : E_c \rightarrow E</math>, Paths <math>\Pi_{pre}, \Pi_{inter}, \Pi_{post}</math>  <b>Result</b>    : <math>\Pi</math> Push (<math>\Pi, \Pi_{pre}</math>) <b>for all</b> <math>e_c \in \Pi_{inter}</math> <b>do</b>   <math>w \leftarrow \text{Target}(e_c)</math>   <b>if</b> <math>\omega(e_c) = \text{false}</math> <b>then</b>     <math>e \leftarrow \psi(e_c)</math> /* <math>e_c</math> is not an original edge */   <b>else</b>     <math>e \leftarrow e_c</math>   <b>Unhide</b> (<math>e</math>)   /* Walk until <math>w</math> is not reached */   <b>while</b> <math>\text{Target}(e) \neq w</math> <b>do</b>     Push (<math>\Pi, e</math>)     <math>v \leftarrow \text{Target}(e)</math>     <b>Hide</b> (<math>e</math>)     <math>e \leftarrow \rho(v)</math>     <b>Unhide</b> (<math>e</math>)   <math>v \leftarrow w</math> Push (<math>\Pi, \Pi_{post}</math>) </pre>
---

**Algorithm 4:** Path Decompression

Since the search has been done using the query  $q_c = (s_c, t_c)$ , the resulting path has to be converted to a path from  $s \in V$  to  $t \in V$  in the original graph. This is done by a simple decompression scheme. Let  $\Pi$  denote the uncompressed path from  $s$  to  $t$ . The algorithm starts by pushing the prefix path  $\Pi_{pre}$  in  $\Pi$ . The decompression of an edge in  $\Pi_{inter}$  is done by first using the mapping  $\psi$  to get the first edge to take and then successively utilizing  $\rho$  mapping to walk on the intermediate edges and pushing them in  $\Pi$ . Once all the edges of  $\Pi_{inter}$  are uncompressed, the postfix path  $\Pi_{post}$  is pushed in  $\Pi$ .

Using a constant time access data structure for the mappings e.g., hashing, decompression can be done in linear time. Algorithm 4 formalizes the decompression algorithm.

### 5.4 Experimental Results

In Table 5.1 we present the results of the graph compression. Column 3 shows the number of nodes after compression. The total time in seconds required for

#GPS Points	#Graph Nodes	#Compressed Nodes	Time
1,277	1,473	199	0.01
1,706	1,777	74	0.02
2,365	2,481	130	0.03
50,000	54,267	4,391	0.59

Table 5.1: Effect of compression.

the compression is shown in Column 4. We observe a significant gain in terms of space reduction with a very small computation time.

#Graph Nodes	#Queries	Compression Time	Decompression Time
199	200	0.01	0.09
74	200	0.02	0.15
130	200	0.03	0.28
4,391	200	0.59	0.65

Table 5.2: Effect of decompression

In Table 5.2, the decompression times in seconds for the paths resulted by 200 random shortest path queries are shown. We also compare it with the compression time in Column 3. In the largest data set the compression time is even lesser than that required for decompressing 200 shortest paths.

## 5.5 Summary

The constructed graph contains a lot of nodes that do not create any *choice* in a search algorithm. We presented a compression algorithm that merges these nodes in such a way that they can be unfolded when we need to show a path in the form of original nodes. The running time of the compression algorithm is  $O(n + k)$ , where  $n$  is the number of GPS-Points and  $k$  is the number of intersection nodes. The space complexity of the original graph is reduced from  $O(n + k)$  to  $O(l + k)$ , where  $l$  is the number of traces and  $l \ll n$ . The experimental results prove the running time in practice.



## Chapter 6

# Navigation in Static Environment

Once the graph is constructed and compressed, it can be used for queries for the shortest paths. In this chapter we look at the shortest path searching problem from a static perspective. We assume that there are no changes in the graph's topology with respect to time. The discussion starts with the formalization of the environment in its simplest form, where edge weights are determined by the distances between the edge end nodes.

This model is then extended to more sophisticated models, where the graph is queried not only for the shortest path but also for the quickest path or for a weighted combination of both. The third model that we discuss in this chapter is based on the observation that the travel time may vary during different times of a day. This model aims at finding the quickest path starting at a particular time.

We then present shortest path algorithms best suited for our domain and how they can be adapted in the models presented. We start by introducing Dijkstra's algorithm, which has become the standard in this field, and discuss its adaptation to our models.

In our domain, where we have a layout with the graph, the Euclidean geometry can be exploited as a heuristic to guide the search algorithm. We review the A\* algorithm in this context. During exploration, the search space can also be pruned by using some pre-computed information. We review one such approach, where pre-computed bounding boxes are used to prune some of the non-promising nodes to be explored. This approach has proved to be very impressive from the point of view of experimental results.

## 6.1 Static Models

This section presents the three different problem settings in our domain. We start by discussing the most basic version, where a graph is queried for a shortest path from a start node to a target node. We then proceed forward to the model, where the graph is queried also for the quickest path as well as for the shortest path or for a weighted combination of both time and distance. Finally, we discuss the model where the graph is queried for the path that is not only shorter in terms of time but also feasible with respect to the desired travel time.

### 6.1.1 The Basic Model

The environment is modeled by a graph  $G_c = (V_c, E_c, d_c, T_c)$  where  $V_c$  is the set of vertices and  $E_c$  is the set of edges. The function  $d_c : E_c \rightarrow \mathbb{R}^+$  assigns weight information with every edge. The function  $T_c : E_c \rightarrow Time \times Time$  gives the temporal information about an edge, precisely - the start time and the end time, that were obtained in traveling along that edge.

For notational simplicity, let us assume that  $T_c$  itself consists of two sub-functions:  $T_{start} : E \rightarrow Time$  and  $T_{end} : E \rightarrow Time$  that when given an edge return the start time and the end time of that edge, respectively.

A query to the system is in the form of a pair of nodes  $(s, t)$ . It aims at the shortest path with respect to weights  $d_c$ .

### 6.1.2 The Time Model

The basic model can be extended by allowing the graph to be queried not only for the shortest path but also for the *quickest* path from  $s$  to  $t$  or for a weighted combination of both distance and time. The query  $q = (s, t, \tau)$  can be posed to the system where,  $\tau \in [0, 1]$  is the *preference* parameter and is used to give the preference of travel time over the distance. Based on  $\tau$ , the new weights  $d_{new}$  of an edge  $e$  can be calculated as:

$$d_{new}(e) = \tau \cdot d_c(e) + (1 - \tau) \cdot [T_{end}(e) - T_{start}(e)]$$

The shortest path algorithm returns the shortest path, if one exists, based on new weight information  $d_{new}$ .

### 6.1.3 The Absolute Time Model

It is a general observation that the travel time on a specific path can vary according to the time of the day - the travel time during rush hours is not same as during the night. Using this observation we now allow our graph to

<b>Input</b>	: Graph $G_c$ , Start Vertex $s$
<b>Output</b>	: Shortest path distances $dist$
1	Create Priority Queue $Q$
2	Insert $(Q, (s, 0))$
3	$dist[s] \leftarrow 0$
4	<b>for all</b> $v \in V \setminus \{s\}$ <b>do</b>
5	Insert $(Q, (v, \infty))$
6	$dist[v] \leftarrow \infty$
7	<b>while</b> $Q$ <i>not empty</i> <b>do</b>
8	$u \leftarrow \text{DeleteMin}(Q)$
9	<b>for all</b> $v \in \text{Adjacent}(u)$ <b>do</b>
10	<b>if</b> $dist[v] > dist[u] + d_c((u, v))$ <b>then</b>
11	$dist[v] \leftarrow dist[u] + d_c((u, v))$
12	DecreaseKey $(Q, (v, dist[v]))$
13	<b>return</b> $\Pi$

**Algorithm 5:** Dijkstra's single-source shortest path algorithm.

be queried for the quickest path to be traveled at a particular time. Formally, a query is a quadruple,  $(s, t, time_s, \epsilon)$ , where  $time_s$  is the desired starting time for travel. It is posed to the system as a request for a quickest path that is *also* feasible with respect to time at every edge in the path up to  $\epsilon$  time unit. We say that an edge  $e = (u, v)$  is *feasible* if the difference between the minimum time to reach  $u$  and the starting time stamp at  $e$  is bounded from above and below by  $\epsilon$ .

In the search algorithm, feasibility can be insured by checking for the following condition before the expansion of a node  $u$  along the edge  $e$ :  $|time_s + t_{min}[u] - T_{start}(e)| \leq \epsilon$ , where  $t_{min}[u]$  is the minimum time required to reach  $u$  from  $s$ .

## 6.2 Dijkstra's Shortest Path Algorithm

Dijkstra's algorithm is a single-source shortest path algorithm, i.e., it tries to find the shortest path from the start node to all the other nodes in the graph. The main principle behind Dijkstra's algorithm is that it performs a best-first search in the graph while keeping track of the shortest distance calculated so far from the start node to the current node.

Initially the shortest distance values of all the nodes except  $s$  are initialized to  $\infty$ ,  $s$  is given the value 0. The algorithm then inserts all the nodes in a priority queue using the shortest distance value as the *key* of the nodes.

The procedure then iterates through the priority queue by removing the node with the minimum shortest distance value. At every iteration it explores all the children of the removed node  $u$ . In case it finds a shorter path to a child  $v$  of  $u$  that goes through the  $u$ , it decreases the shortest distance value of  $v$  to the sum of the shortest path value of  $s$  to  $u$  and the cost of the edge  $(u, v)$ . The iterations end when the queue becomes empty.

Algorithm 5 shows the complete algorithm in pseudo-code. The operations, inserting an element (**Insert**), deleting the minimum element (**DeleteMin**), and decreasing a value (**DecreaseKey**) can be carried out efficiently by using a priority queue data structure called Fibonacci heaps. It supports **create**, **insert**, and **DecreaseKey** in  $O(1)$  amortized time and **DeleteMin** in  $O(\log n)$  amortized time. Using Fibonacci heaps, the running time of Dijkstra's algorithm is  $O(n \log n + m)$  where  $n = |V_c|$  and  $m = |E_c|$ .

The correctness of Dijkstra's algorithm follows from the following lemma.

**Lemma 6.2.1** [4] *The nodes removed at the line 8 in Algorithm 5 have correct shortest distance values.*

## 6.2.1 Dijkstra's Algorithm for the Basic Model

Note that Dijkstra's algorithm is basically a single-source shortest path algorithm. In our case we are only interested in finding the shortest path to the target node  $t$  and not to all nodes. Also, the basic version of Dijkstra (Algorithm 5) calculates only the shortest distance values and needs to be adopted to calculate the actual shortest paths also.

In the light of our requirements, we can extend Algorithm 5 by inserting one more termination condition i.e. to halt when the node removed from the priority queue is  $t$ . The actual shortest path can be saved in a prefix vector ( $\Pi$ ) that maps a node  $v$  to the edge to be taken to reach the previous node in the shortest path from  $s$  to  $v$ . The corresponding changes are shown in Algorithm 6 with lines shaded in black.

**Theorem 6.2.2** *Algorithm 6 terminates with the shortest path from  $s$  to  $t$  in  $\Pi$ , if one exists.*

**Proof** The proof follows from the Lemma 6.2.1 when the target  $t$  is removed from the priority queue.

## 6.2.2 Dijkstra's Algorithm for the Time Model

The variant of Dijkstra's algorithm variant for the time model is a natural extension of Algorithm 6. The main extension is the new edge weight function

```

Input      : Graph  $G_c$ , Start Vertex  $s$ , Target Vertex  $t$ 
Output    : Shortest path  $\Pi$ 
1 Create Priority Queue  $Q$ 
2 Insert  $(Q,(s,0))$ 
3  $\text{dist}[s] \leftarrow 0$ 
4 for all  $v \in V \setminus \{s\}$  do
5   | Insert  $(Q,(v,\infty))$ 
6   |  $\text{dist}[v] \leftarrow \infty$ 
7 while  $Q$  not empty do
8   |  $u \leftarrow \text{DeleteMin}(Q)$ 
9   | if  $u = t$  then
10  |   | /* target reached */
10  |   | return  $\Pi$ 
11  |   for all  $v \in \text{Adjacent}(u)$  do
12  |   |   if  $\text{dist}[v] > \text{dist}[u] + d_c((u,v))$  then
13  |   |   |  $\text{dist}[v] \leftarrow \text{dist}[u] + d_c((u,v))$ 
14  |   |   | DecreaseKey  $(Q,(v, \text{dist}[v]))$ 
15  |   |   |  $\Pi[v] \leftarrow (u,v)$ 
16 return  $\Pi$ 

```

**Algorithm 6:** Dijkstra's algorithm for the Basic Model.

that makes use of the preference parameter  $\tau$  as described in Section 6.1.2. The pseudo-code of the adapted algorithm is shown in Algorithm 7, with changes shown with the lines shaded in black.

Since Dijkstra's algorithm is independent of any layout information about the graph, the new weight function will not affect the optimality of the algorithm.

### 6.2.3 Dijkstra's Algorithm for the Absolute Time Model

A search algorithm suitable for this model should be able to keep track of the time that has been traversed so far and should check the feasibility condition on every edge before exploring on that edge.

Since, we are now only interested in having a *quickest* path between the query points, Algorithm 6 can be extended to use only the *minimum time value* in place of shortest distance value while exploring. The adapted algorithm (Algorithm 8) returns the quickest path, if one exists, between the query nodes that is feasible up to  $\epsilon$  at every internal edge.

The *gain* variable in the algorithm is to normalize the time gain in order to avoid the situation, where the time gain on the nodes adds up to a value that

```

Input      : Graph  $G_c$ , Start Vertex  $s$ , Target Vertex  $t$ , Preference
               parameter  $\tau$ 
Output    : Shortest path  $\Pi$ 
1 Create Priority Queue  $Q$ 
2 Insert  $(Q, (s, 0))$ 
3  $\text{dist}[s] \leftarrow 0$ 
4 for all  $v \in V \setminus \{s\}$  do
5   | Insert  $(Q, (v, \infty))$ 
6   |  $\text{dist}[v] \leftarrow \infty$ 
7 while  $Q$  not empty do
8   |  $u \leftarrow \text{DeleteMin}(Q)$ 
9   | if  $u = t$  then
10  |   | return  $\Pi$ 
11  |   for all  $v \in \text{Adjacent}(u)$  do
12  |     | Edge  $e \leftarrow (u, v)$ 
13  |     |  $d_{\text{new}} \leftarrow \tau \cdot d_c(e) + (1 - \tau) \cdot [T_{\text{end}}(e) - T_{\text{start}}(e)]$ 
14  |     | if  $\text{dist}[v] > \text{dist}[u] + d_{\text{new}}$  then
15  |     |   |  $\text{dist}[v] \leftarrow \text{dist}[u] + d_{\text{new}}$ 
16  |     |   | DecreaseKey  $(Q, (v, \text{dist}[v]))$ 
17  |     |   |  $\Pi[v] \leftarrow e$ 
18 return  $\Pi$ 

```

**Algorithm 7:** Dijkstra's algorithm for the Time Model.

can result in an unreachable  $\epsilon$  time window.

### 6.3 Heuristic Search - A\*

In a search algorithm, the exhaustive exploration can be avoided by using domain-dependent information to give more priority to the promising nodes than to the less promising nodes. In our domain, where we have a layout with the graph, the search algorithm can be guided by exploiting the straight line distances as the heuristic. This heuristic can be used to prioritize the expansion of nodes that are closer to the goal. This approach has been formalized in the A\* algorithm [15].

A\* uses a combination of both the shortest distance from the start node to a node  $u$  and the estimated distance from  $u$  to the target node, to prioritize the nodes to be expanded. It can be seen as a variant of Dijkstra's algorithm with a new priority function. Formally, we define the priority function for a node  $u$  as:  $f'(u) = g(u) + h(u)$ , where  $g(u)$  is the shortest path distance from

```

Input      : Graph  $G_c$ , Start Vertex  $s$ , Target Vertex  $t$ , Start time
                $time_s$ , Tolerance  $\epsilon$ 
Output    : Shortest path  $\Pi$ 
1 Create Priority Queue  $Q$ 
2 Insert ( $Q, (s, 0)$ )
3  $t_{min} [s] \leftarrow 0$ 
4 for all  $v \in V \setminus \{s\}$  do
5   | Insert ( $Q, (v, \infty)$ )
6   |  $t_{min} [v] \leftarrow \infty$ 
7 while  $Q$  not empty do
8   |  $u \leftarrow \text{DeleteMin} (Q)$ 
9   | if  $u = t$  then
10  |   | break
11  |   | for all  $v \in \text{Adjacent} (u)$  do
12  |   |   | Edge  $e \leftarrow (u, v)$ 
13  |   |   |  $gain \leftarrow time_s + t_{min} [u] - T_{start} (e)$ 
14  |   |   | if  $|gain| \leq \epsilon$  then
15  |   |   |   | /* it is feasible to expand the edge  $e$  */
16  |   |   |   | if  $t_{min} [v] > t_{min} [u] + T_{end} (e) - T_{start} (e)$  then
17  |   |   |   |   |  $t_{min} [v] \leftarrow t_{min} [u] + T_{end} (e) - T_{start} (e) - gain$ 
18  |   |   |   |   | DecreaseKey ( $Q, (v, t_{min} [v])$ )
19  |   |   |   |   |  $\Pi[v] \leftarrow e$ 
19 return  $\Pi$ 

```

**Algorithm 8:** Dijkstra's Algorithm for the Absolute Time Model.

$s$  to  $u$  (calculated so far) and  $h(u)$  is the estimated distance between  $u$  and the target  $t$ . In the Euclidean space, the straight line distance can be taken as an estimated cost to the target, i.e.,

$$h(u) = \|u - t\|_2.$$

A heuristic is said to be *admissible* if it never overestimates the actual cost from the current node to the target node.

**Lemma 6.3.1** [26] *If the heuristic is admissible, A\* is optimal and complete.*

In our case, due to compression, the edge weights ( $d_c$ ) do not correspond to the Euclidean distances and the edges do not hold triangular property. But according to Lemma 5.1.1, we have for all  $(u, v) \in E_c$

$$d_c((u, v)) \geq \|v - u\|_2. \quad (6.1)$$

```

Input      : Graph  $G_c$ , Start Vertex  $s$ , Target Vertex  $t$ 
Output    : Shortest path  $\Pi$ 
1 Create Priority Queue  $Q$ 
2 Insert ( $Q, (s, h(s))$ )
3  $\text{dist}[s] \leftarrow h(s)$ 
4 for all  $v \in V \setminus \{s\}$  do
5   | Insert ( $Q, (v, \infty)$ )
6   |  $\text{dist}[v] \leftarrow \infty$ 
7 while  $Q$  not empty do
8   |  $u \leftarrow \text{DeleteMin}(Q)$ 
9   | if  $u = t$  then
10  |   | return  $P_i$ 
11  |   | for all  $v \in \text{Adjacent}(u)$  do
12  |   |   |  $\text{dist}_{\text{new}} \leftarrow \text{dist}[u] + d_c((u,v)) + h(v) - h(u)$ 
13  |   |   | if  $\text{dist}[v] > \text{dist}_{\text{new}}$  then
14  |   |   |   |  $\text{dist}[v] \leftarrow \text{dist}_{\text{new}}$ 
15  |   |   |   | DecreaseKey ( $Q, (v, \text{dist}[v])$ )
16  |   |   |   |  $\Pi[v] \leftarrow (u, v)$ 
17 return  $\Pi$ 

```

**Algorithm 9:** A\* Algorithm.

i.e. the straight line distance never overestimates the actual cost, hence we can say that  $h$  is admissible in our case, too.

**Lemma 6.3.2** *In the compressed graph  $G_c$ , the Euclidean distance heuristic  $h$  corresponds to an admissible heuristic.*

**Proof** Follows from the inequality 6.1.

A\* can result in the reopening of the nodes already visited and deleted from the priority queue. It is proven in [24] that if the heuristic is consistent, no reopening can occur. A heuristic is said to be *consistent* if the function  $f$  always increases during exploration. This consistency can be guaranteed if the edge weights are always non-negative. Note that the edge weights now depend on the heuristic estimates also i.e., the cost of an edge  $e = (u, v)$  can be defined as  $d_c((u, v)) + h(v) - h(u)$ , where the subtraction is due to the fact that  $h(u)$  had already been added to the shortest distance value from  $s$  to  $u$ .

In the following, we will prove that in the compressed graph  $G_c$ , the new edge costs result in a consistent heuristic.

**Lemma 6.3.3** *In the compressed graph  $G_c$ ,  $h$  is consistent, i.e., for all  $(u, v) \in V_c$  we have  $d_c((u, v)) + h(v) - h(u) \geq 0$ .*

**Proof** We will prove the lemma by contradiction. Let us assume that the condition does not hold i.e.,

$$\begin{aligned} d_c((u, v)) + h(v) - h(u) &< 0 \\ \text{or, } d_c((u, v)) &< h(u) - h(v) \end{aligned}$$

According to the definition of  $h$ ,  $h(u) = \|u - t\|_2$ , we can rewrite the above inequality as

$$d_c((u, v)) < \|u - t\|_2 - \|v - t\|_2 \quad (6.2)$$

Inequality 6.1 gives straight line distance as a lower bound on the distance value  $d_c$  of an edge. We can safely replace  $d_c$  by  $\|u - v\|$  in Inequality 6.2 and obtain the following

$$\begin{aligned} \|u - v\|_2 &< \|u - t\|_2 - \|v - t\|_2 \\ \|u - v\|_2 + \|v - t\|_2 &< \|u - t\|_2 \end{aligned}$$

This is a **contradiction** to the triangular property in Euclidean space.

Algorithm 9 shows the pseudo-code for the A\* algorithm without the reopening support.

**Theorem 6.3.4** *Algorithm 9, when applied on the compressed graph  $G_c = (V_c, E_c, d_c, T_c)$  for the shortest distance from a start node  $s$  to a target node  $t$ , terminates with the optimal path in  $\Pi$ , if one exists.*

**Proof** The proof of optimality follows from the admissibility of  $h$  (Lemma 6.3.2) and the absence of reopenings follows from Lemma 6.3.3.

### 6.3.1 A\* for the Time Model

In the time model, the straight line heuristic is not appropriate while searching for the quickest path or for a weighted combination of the quickest and shortest paths. Time plays an important role in this model and a heuristic based on the travel time estimates can well serve this model.

Let us first consider a simpler version of the problem where the graph is queried only for the quickest path. We observe that by a simple extension to the model and by assuming a maximum speed  $\mu$  of the mobile object, a heuristic estimate can be given for the time required to travel a path between two nodes. The heuristic estimate  $h'$  for a node  $u$  can be defined as

$$h'(u) = \frac{1}{\mu} \cdot \|u - t\|_2.$$

The admissibility of  $h'$  follows from the most optimistic estimates both in terms of speed and distance.

Next, we consider the problem when a weighted combination of both the quickest and shortest paths is required i.e., when  $\tau \in (0, 1)$ . Since the value of  $\tau$  is constant throughout the graph, a heuristic estimate consisting of a linear combination of both  $h$  and  $h'$  can be used. We define  $h''$  as the heuristic estimate for a node  $u$  in the time model as

$$\begin{aligned} h''(u) &= \tau \cdot \frac{1}{\mu} \cdot \|u - t\|_2 + (1 - \tau) \cdot \|u - t\|_2 \\ \text{or, } h''(u) &= \|u - t\|_2 \left( \tau \cdot \frac{1}{\mu} + (1 - \tau) \right). \end{aligned} \quad (6.3)$$

The admissibility of  $h''$  can be proved by the following lemma.

**Lemma 6.3.5** *In the compressed graph  $G_c$ ,  $h''$  is admissible.*

**Proof** Since  $\tau$  and  $\mu$  are constant for the entire graph and  $\|u - t\|_2$  never overestimates the actual edge cost,  $h''$  never overestimates the actual path cost.

As with the consistency of  $h$  in the last section, we can prove the consistency of  $h''$  by proving that the edge weights remain non-negative during exploration. Recall that the weight of an edge  $e$  in time model is  $\tau \cdot T_c(e) + (1 - \tau) \cdot d_c((u, v))$ .

**Lemma 6.3.6** *In the compressed graph  $G_c$ ,  $h''$  is consistent, i.e., for all  $(u, v) \in V_c$  we have  $(\tau \cdot T_c(e) + (1 - \tau) \cdot d_c((u, v))) + h''(v) - h''(u) \geq 0$ .*

**Proof** We will prove the above lemma by contradiction. Let us assume that the condition does not hold i.e.,

$$\begin{aligned} (\tau \cdot T_c(e) + (1 - \tau) \cdot d_c((u, v))) + h''(v) - h''(u) &< 0 \\ \text{or } \tau \cdot T_c(e) + (1 - \tau) \cdot d_c((u, v)) &< h''(u) - h''(v). \end{aligned}$$

According to Equation 6.3,  $h''(u) = \|u - t\|_2 \left( \tau \cdot \frac{1}{\mu} + (1 - \tau) \right)$ . We can rewrite the above inequality as:

$$\begin{aligned} \tau \cdot T_c(e) + (1 - \tau) \cdot d_c((u, v)) &< \|u - t\|_2 \left( \tau \cdot \frac{1}{\mu} + (1 - \tau) \right) - \\ &\|v - t\|_2 \left( \tau \cdot \frac{1}{\mu} + (1 - \tau) \right) \end{aligned}$$

We know from Inequality 6.1 that  $d_c((u, v)) \geq \|u - v\|_2$ . Also since the time estimate for traversing an edge with maximum speed never overestimates the

actual time, we can say that  $T_c(e) \geq \frac{1}{\mu}\|u - v\|_2$ . The above inequality can be rewritten as:

$$\begin{aligned} \tau \cdot \frac{1}{\mu}\|u - v\|_2 + (1 - \tau) \cdot \|u - v\|_2 &< \|u - t\|_2 \left( \tau \cdot \frac{1}{\mu} + (1 - \tau) \right) - \\ &\|v - t\|_2 \left( \tau \cdot \frac{1}{\mu} + (1 - \tau) \right) \\ \|u - v\|_2 \left( \tau \cdot \frac{1}{\mu} + (1 - \tau) \right) &< \|u - t\|_2 \left( \tau \cdot \frac{1}{\mu} + (1 - \tau) \right) - \\ &\|v - t\|_2 \left( \tau \cdot \frac{1}{\mu} + (1 - \tau) \right) \\ \|u - v\|_2 &< \|u - t\|_2 - \|v - t\|_2 \\ \|u - v\|_2 + \|v - t\|_2 &< \|u - t\|_2 \end{aligned}$$

This is a **contradiction** to the triangular property in Euclidean space.

**Theorem 6.3.7** *Algorithm 9 with  $h''$  as the heuristic estimate, when applied on the compressed graph  $G_c = (V_c, E_c, d_c, T_c)$  for the shortest distance from a start node  $s$  to a target node  $t$ , terminates with the optimal path in  $\Pi$ , if one exists.*

**Proof** The proof of optimality follows from the admissibility of  $h$  (Lemma 6.3.5) and the absence of reopenings follows from Lemma 6.3.6.

### 6.3.2 A\* for the Absolute Time model

Algorithm 10 shows the A\* algorithm adapted for the absolute time model. It is basically an extension of Dijkstra's algorithm variant for Absolute Time Model with heuristic estimate  $h'$ .

## 6.4 Geometric Pruning

In a search algorithm, the search time can be reduced by information that can be computed before the arrival of the queries. This information can be used to accelerate the search algorithm during frequent shortest path queries. The most effective information that can be computed and memorized are all-pairs shortest paths with the Floyd-Warshall or Johnsons' algorithm [4]. The corresponding shortest paths can be saved in a prefix matrix of  $n \times n$ . This can reduce the query time to a mere linear time backtracking from target to source. But the space required for saving this information is  $O(n^2)$  which is not feasible in our case because of  $n$  being very large.

```

Input      : Graph  $G_c$ , Start Vertex  $s$ , Target Vertex  $t$ , Start time
                $time_s$ , Tolerance  $\epsilon$ 
Output    : Shortest path  $\Pi$ 
1 Create Priority Queue  $Q$ 
2 Insert ( $Q, (s, h(s))$ )
3  $t_{min} [s] \leftarrow 0$ 
4 for all  $v \in V \setminus \{s\}$  do
5   | Insert ( $Q, (v, \infty)$ )
6   |  $t_{min} [v] \leftarrow \infty$ 
7 while  $Q$  not empty do
8   |  $u \leftarrow \text{DeleteMin} (Q)$ 
9   | if  $u = t$  then
10  |   | return  $\Pi$ 
11  |   | for all  $v \in \text{Adjacent} (u)$  do
12  |   |   | Edge  $e \leftarrow (u, v)$ 
13  |   |   |  $gain \leftarrow time_s + t_{min} [u] - T_{start} (e)$ 
14  |   |   | if  $|gain| \leq \epsilon$  then
15  |   |   |   | /* it is feasible to expand the edge  $e$  */
16  |   |   |   |  $time_{new} \leftarrow t_{min} [u] + T_{end} (e) - T_{start} (e) + h'(v) - h'(u)$ 
17  |   |   |   | if  $t_{min} [v] > time_{new}$  then
18  |   |   |   |   |  $t_{min} [v] \leftarrow time_{new} - gain$ 
19  |   |   |   |   | DecreaseKey ( $Q, (v, t_{min} [v])$ )
20  |   |   |   |   |  $\Pi[v] \leftarrow e$ 
20 return  $\Pi$ 

```

**Algorithm 10:** A\* for the Absolute Time Model.

A closer look at our domain reveals that the layout of the graph can be well-exploited for this purpose. A recent research on annotating a graph by geometric containers [29] to guide a search algorithm has shown significant gain in terms of the number of expanded nodes in geometric domain. In this approach, an axis-parallel bounding box is attached with an edge  $e = (u, v)$  that contains all the nodes  $w$  that can be reached on a shortest path from  $u$  starting from  $e$ . These bounding boxes can be computed in  $O(n^2 \log n + nm)$  time and utilize only  $O(n)$  space. While searching for the shortest path between the query nodes  $s$  and  $t$ , only those neighbors  $v$  of the current node  $u$  are visited that contain the target  $t$  in the bounding box associated with the edge  $(u, v)$ . The results have shown a reduction of 90 to 95% in explored nodes in rail networks of different European countries [29].

Algorithm 11 shows the algorithm for computing the containers by running Dijkstra's algorithm for all nodes. A container  $BB$  associated with an edge  $(u, v)$  is *consistent*, if for all shortest paths from  $u$  to  $t$  that start with the edge

```

Input      : Graph  $G_c$ 
Output    : Rectangular Containers  $BB : E \rightarrow V'_c \subseteq V_c$ 
1 for all nodes  $s \in V$  do
2   Create Priority Queue  $Q$ 
3   Insert  $(Q, (s, 0))$ 
4    $\text{dist}[s] \leftarrow 0$ 
5   for all  $v \in V \setminus \{s\}$  do
6     Insert  $(Q, (v, \infty))$ 
7      $\text{dist}[v] \leftarrow \infty$ 
8   while  $Q$  not empty do
9      $u \leftarrow \text{DeleteMin}(Q)$ 
10    for all  $v \in \text{Adjacent}(u)$  do
11      if  $\text{dist}[v] > \text{dist}[u] + d_c((u, v))$  then
12         $\text{dist}[v] \leftarrow \text{dist}[u] + d_c((u, v))$ 
13        DecreaseKey  $(Q, (v, \text{dist}[v]))$ 
14        if  $u = s$  then
15           $A[v] \leftarrow (s, v)$ 
16        else
17           $A[v] \leftarrow A[u]$ 
18    for all nodes  $y \in V \setminus \{s\}$  do
19      Enlarge  $BB[A[y]]$  to contain  $y$ 

```

**Algorithm 11:** Algorithm for the creation of containers.

$(u, v)$ , the target  $t$  is in  $BB[(u, v)]$  [29].

This section discusses the adaptation of this approach for the basic and the time model. This approach is not usable in the absolute time model because of extra parameters in the query tuple.

### 6.4.1 Bounding Box Pruning for the Basic Model

Algorithm 12 shows the pseudo-code for the Dijkstra's algorithm for the basic model utilizing the bounding box pruning information. The main extension is line 12, where before visiting  $v$ , it is checked, whether or not going along the edge  $(u, v)$  can take us to the target or not.

In the presence of consistent containers for all edges, the proof for the correctness of the Algorithm 12 follows from the argument about the preservation of order in which edges are processed in the basic Dijkstra's algorithm and in Algorithm 12 [29].

```

Input      : Graph  $G_c$ , Start Vertex  $s$ , Target Vertex  $t$ 
Output    : Shortest path  $\Pi$ 
1 Create Priority Queue  $Q$ 
2 Insert  $(Q, (s, 0))$ 
3  $\text{dist}[s] \leftarrow 0$ 
4 for all  $v \in V \setminus \{s\}$  do
5   | Insert  $(Q, (v, \infty))$ 
6   |  $\text{dist}[v] \leftarrow \infty$ 
7 while  $Q$  not empty do
8   |  $u \leftarrow \text{DeleteMin}(Q)$ 
9   | if  $u = t$  then
10  |   | /* target reached */
11  |   | return  $\Pi$ 
12  |   | for all  $v \in \text{Adjacent}(u)$  do
13  |   |   | if  $t \in BB[(u, v)]$  then
14  |   |   |   | if  $\text{dist}[v] > \text{dist}[u] + d_c((u, v))$  then
15  |   |   |   |   |  $\text{dist}[v] \leftarrow \text{dist}[u] + d_c((u, v))$ 
16  |   |   |   |   | DecreaseKey  $(Q, (v, \text{dist}[v]))$ 
17  |   |   |   |   |  $\Pi[v] \leftarrow (u, v)$ 
17 return  $\Pi$ 

```

**Algorithm 12:** Bounding box algorithm for the Basic Model.

### 6.4.2 Bounding Box Pruning for the Time Model

Bounding boxes are domain-dependent computations. Bounding boxes calculated for distance queries are not usable for the time queries. Hence, we restrict the value of  $\tau$  to be either 0 or 1 and pre-compute separate containers for both shortest path and quickest paths. Algorithm 12 can be used for this purpose with using time difference instead of  $d_c$  while searching for the quickest path.

### 6.4.3 Bounding Box Pruning with A\*

It follows from the definition of *consistent* container that, if  $S((u, v))$  is the set of nodes that are reachable on the shortest path from  $u$  using the edge  $(u, v)$  then  $S((u, v)) \subseteq BB[(u, v)]$ . That is, there can be nodes in  $BB[(u, v)]$  that do not belong to  $S((u, v))$ . This motivates the use of straight line heuristic to further accelerate the exploration process i.e. combining A\* with bounding box pruning [29]. Algorithm 13 shows the resulting algorithm that combines bounding box approach with A\*.

```

Input      : Graph  $G_c$ , Start Vertex  $s$ , Target Vertex  $t$ , Bounding
              boxes  $BB$ 
Output    : Shortest path  $\Pi$ 
1 Create Priority Queue  $Q$ 
2 Insert ( $Q, (s, h(s))$ )
3  $\text{dist}[s] \leftarrow h(s)$ 
4 for all  $v \in V \setminus \{s\}$  do
5   Insert ( $Q, (v, \infty)$ )
6    $\text{dist}[v] \leftarrow \infty$ 
7 while  $Q$  not empty do
8    $u \leftarrow \text{DeleteMin}(Q)$ 
9   if  $u = t$  then
10    return  $\Pi$ 
11  for all  $v \in \text{Adjacent}(u)$  do
12    if  $t \in BB[(u, v)]$  then
13       $\text{dist}_{new} \leftarrow \text{dist}[u] + d_c((u, v)) + h(v) - h(u)$ 
14      if  $\text{dist}[v] > \text{dist}_{new}$  then
15         $\text{dist}[v] \leftarrow \text{dist}_{new}$ 
16        DecreaseKey ( $Q, (v, \text{dist}[v])$ )
17         $\Pi[v] \leftarrow (u, v)$ 
18 return  $\Pi$ 

```

**Algorithm 13:** Bounding Boxes with A\* algorithm.

## 6.5 Re-initialization of Shortest Distance Values

The distance values  $\text{dist}$  for every node has to be re-initialized for every query. This creates an extra linear time overhead on the running time. In case of frequent queries, this linear time overhead can significantly reduce the performance of the search algorithm. This situation can be avoided by maintaining an array of integers  $\text{times}$  corresponding to every node and a global  $\text{currentTime}$  integer. We initialize all of them to zero in the preprocessing. The  $\text{currentTime}$  is incremented by 1 for every search. If  $\text{times}[u]$  is not equal to the  $\text{currentTime}$ , it implies that the node  $u$  is not yet visited and the  $\text{dist}[u]$  contains the value from the previous run. In this situation, we initialize the  $\text{dist}[u]$  to infinity and copy the  $\text{currentTime}$  value to  $\text{times}[u]$  [29].

	$\#nodes$	$t_{pre}$	$t_{search}$	$\#Exp$
Dijkstra	1,473	0.01	0.01	1,293
A*	1,473	0.01	0.00	243
Dijkstra	1,777	0.01	0.01	1,421
A*	1,777	0.00	0.00	451
Dijkstra	2,481	0.00	0.01	1,667
A*	2,481	0.00	0.01	1,600
Dijkstra	54,267	0.27	0.27	44,009
A*	54,267	0.26	0.20	18,755

Table 6.1: Comparison of Dijkstra and A\*.

## 6.6 Experimental Results

In Table 6.1, we give a comparison between the performance of Dijkstra’s algorithm and A\* for the shortest path searching. Time required for the initialization of data structures is shown in the column  $t_{pre}$ . Note that time values shown as 0.00 actually represent the values that are less than 10 milliseconds. The column  $t_{search}$  reflects the search time for one query consisting of the extreme points of the graph. Total number of expansions are shown in  $\#Exp$  column. The gain of using A\* instead of traditional Dijkstra is 3 times on the average on our data sets. We ran these experiments on the uncompressed graph just to inflate the calculated times since singleton times were not measurable.

	$\#nodes$	$t_{pre}$	$t_{search}$	$\#Exp$	$t'_{search}$	$\#Exp'$
Dijkstra	199	1.87	0.34	6,596	0.60	19,595
A*	199	1.87	0.26	3,135	0.19	7,912
Dijkstra	74	0.52	0.30	2896	0.29	7271
A*	74	0.52	0.28	2762	0.30	5169
Dijkstra	130	1.14	0.49	4144	0.54	12392
A*	130	1.14	0.49	3848	0.56	10060
Dijkstra	4,391	1,299	9.36	101,064	17.18	458,156
A*	4,391	1,299	8.11	65,726	12.88	217,430

Table 6.2: Effect of geometric pruning.

Table 6.2 shows the results of using geometric pruning in Dijkstra and A\* for 200 queries. These results are the averages taken over 10 different episodes each of 200 queries. We compare the time needed by Dijkstra and A\* with pruning information and without pruning information. Column  $t'_{search}$  and  $\#Exp'$  show the time and number of expansions without geometric pruning, respectively. We observe a significant gain in the number of expanded nodes.

## 6.7 Summary

This chapter addressed the problem of the GPS based routing from a static point of view. We presented three models in this regard that offer different functionalities based on the real-world requirements. The discussion then proceeds to the searching in those models using the *de facto* Dijkstra's algorithm. The possibility of accelerating the search using domain-dependent knowledge is then exploited, resulting in different A\* variants for different models. We then established an implicit notation of *on-line vs. off-line* computation of searches. We observed that precomputing some information about the topology and paths in the graphs before the arrival of queries (*off-line*) can significantly reduce the searching time (*on-line*).

In a recent research [30], the bounding box pruning strategy is extended to bidirectional search. It requires the computation of both source and target containers. The target containers can be calculated by reversing all the edges and running Algorithm 11 on them.



## Chapter 7

# Navigation in Dynamic Environment

Consider the scenario when, while following a shortest path, the user detects that because of a road accident or some other reason, there is a traffic jam. A traffic jam implies an increase in travel time for the affected area. In this case a second shortest path from the current position to the destination is needed that also considers the affected area.

In our case this situation is modeled as an increase in the weight of the edges in the affected path. This implies the invalidity of some of the bounding boxes, particularly the ones that contain edges with increased edge weights. This, in turn, restricts the bounding-box searching algorithm to exploit the precomputed information.

We present two models for introducing dynamics in the system. In the first model, due to a disturbance, we assume that some particular edges are directly affected and their weights are increased, while in the second setting, we represent a disturbance as a geometrical object on the graph, that affects all the edges covered by that object. In both of these models we assume that the changes in the graph are temporary in nature, i.e., they disappear after some time.

This chapter starts with an introduction to the terminology. A rigorous treatment of the dynamic models is presented next.

The subsequent chapters present two approaches. We characterize the approaches as *off-line* and *on-line*. In the *off-line* approach, upon the arrival or termination of a disturbance, the graph is updated by increasing or decreasing the weights of the affected edges. The search procedure in this approach has to find the shortest path in the *updated* graph. In the *on-line* approach, the disturbances are not considered until the arrival of a query. During exploration, we dynamically check, whether the next candidate edge for the exploration has

been affected by a disturbance.

## 7.1 Problem Characterization

Eppstein et. al [12] classifies the dynamic graph problems in two types, based on the kind of dynamics allowed. A problem is *fully dynamic* if both insertion and deletion of vertices and edges are allowed. If we are allowed to perform only one kind of operation, that is, either insertion or deletion, that problem is referred to as *partially dynamic*. The *partially dynamic* problems can be further classified as either *incremental* - the edges or vertices can only be inserted - or *decremental* - the edges or vertices can only be deleted.

Let us now define a sub-classification of *partially dynamic* problems. A partially dynamic graph problem is *semi-decremental* (*semi-incremental*) if we *only* allow an increase (decrease) in edge weights. An increase in edge weight basically decreases its probability of being selected as part of a shortest path. Note that this also covers the case when an edge cannot be use at all - by increasing the edge weight to  $+\infty$ . In the current context we are dealing with a *semi-decremental* problem as we allow only the increase in edge weights. The present domain is a temporal one, changes of edge weights are not permanent and appear and disappear in the course of time.

We say that a geometrical constraint representing a disturbance is *soft* if the weight of an edge is increased only by a constant amount. This can be exemplified by a situation, where e.g: only 3 out of 4 lanes of a street are affected. The extreme case, when the weight is increased to  $+\infty$ , hence making the edge completely unusable, is characterized as a *hard* constraint.

## 7.2 Dynamic Models

A dynamic shortest path problem can be defined as a 3-tuple  $D = (G_c, C, q)$ . It aims at answering the shortest path query  $q$  in the graph  $G_c$ , in the presence of set of constraints  $C$ . The query  $q$  is represented as  $q = (s, t, time_s)$  with  $s$  as the start node,  $t$  as the target node and  $time_s$  as the desired starting time when the path is to be traversed.

The set of constraints  $C$  defines disturbances that may arise in the road network in the course of time. At the basic level we define these constraints as affecting a subset of edges by increasing their weights. Each constraint is also characterized by the time after which the constraint is no longer valid. This corresponds to the situation when the traffic flow has returned to normal after a disturbance.

The increase in weight can be accommodated in the graph  $G_c$  by increasing

the values  $d_c(e)$  of edges  $e$ . This restricts us to search the shortest path in the modified graph.

This model can be extended to a more general one that facilitates quickest path searching or a combination of both shortest and quickest paths. The extension involves utilizing a separate weight function  $w$  and a priority parameter  $\tau$  value to give priority to distance or time. The weight of an edge  $e$  is  $w(e) = \tau \times (T_{end}(e) - T_{start}(e)) + (1 - \tau) \times d_c(e)$ . Note that this weight function should be calculated beforehand and remains constant during the computation.

For this case constraints can be defined in terms of an increase in the weight  $w$  of an edge. This leads to a re-formalization of our problem as a 5-tuple  $D = (G_c, w, \tau, C, q)$ .

In the following we elaborate our problem from two different perspectives. In the first one we represent disturbances as affecting the edges directly and increasing their weights. We then present an alternative approach, where a disturbance is represented as affecting a rectangular area superimposed on the graph. All the edges underneath this area are in turn considered to be affected by the disturbance.

### 7.2.1 Individual Edge Dynamics

In this model, we define  $C$  to be a set of constraints where each constraint  $c \in C$  maps an edge to a *set* of changes due to disturbances on that edge. Note that we use the term *set* because one disturbance might lead to other disturbances that affect the edge weights differently. As an example, a car accident may lead to a blocked lane and a resulting traffic jam. Removing the traffic jam might not result in the opening of that affected lane.

Formally we define  $C$  as a set of constraints  $c : E \rightarrow \mathcal{P}\{\mathbb{R} \times Time\}$ , i.e., a constraint is a set of tuples representing the changes in the weight of an edge due to disturbances and the time when the change disappears.

### 7.2.2 Disturbances as Geometrical Objects

Sometimes it is difficult to pinpoint the exact location of an edge, where the disturbance actually is. To accommodate this difficulty we extend our model to allow the definition of constraints on the graph in the form of geometrical objects. For the sake of simplicity, we assume each geometrical object to be an iso-oriented rectangle. One reason for choosing *rectangles* to represent disturbances is the already developed fast computational geometry algorithms for rectangles' manipulation.

These rectangles are considered to be a separate entities from the graph and are situated in a layer on top of the graph.

All the edges underneath a rectangular object are assumed to be affected by the disturbance. The edges affected can be divided into three categories. They can either be situated completely within the rectangle, or intersect its boundary at one or two points.

These disturbances can be formalized in the form of constraints  $c : \Gamma \rightarrow \mathbb{R} \times Time$  where  $\Gamma$  is the set of rectangles corresponding to the affected areas. A rectangle  $\gamma \in \Gamma$  is mapped to an *increase* in the weight of the edges underneath it and a *Time* value after which that constraint is considered as *void*. Note that  $\Gamma$  can also be viewed as a function that maps a  $\gamma$  to a set  $E_\gamma \subseteq E$ , i.e. the set of edges affected by  $\gamma$ .

In fact, combining these concepts about  $\Gamma$ , we can view this problem also as an *Individual Edge Dynamics* problem, where more than one edge have the same *increase* and *termination* time values.

### 7.3 Affects of Disturbances on Containers

Due to a change in the weights, the pre-computed information might become invalid. Some of the bounding boxes may mislead the shortest path algorithm and can result in a path other than the *shortest path*. Re-computing the bounding boxes for every change is expensive in terms of time.

Recently, [30] presented an approach where the authors have obtained a run-time reduction of 2 to 3 times as compared to the naive approach of re-computing all the bounding boxes in case of an update. The authors extend the bounding box pruning to bi-directional search algorithms by computing the source containers as well as the target containers. A source container  $S(u, v)$  of an edge  $(u, v)$  is defined as a bounding box that contains all the nodes  $s$  for which there is a shortest  $s$ - $v$ -path that ends with the edge  $(u, v)$ .

The main idea behind their approach is to enlarge the bounding boxes. In case of an increase in edge weight of an edge  $e = (x, y)$ , the set of potentially affected nodes for  $s \in V$  is  $\text{Pot-Aff}(s) = \{v \in V : d_{new}(s, v) < d_{new}(s, x) + w_{new}(x, y) + d_{new}(y, v)\}$ , where  $d_{new}$  are the new shortest path distances and  $w_{new}$  are the new weights. Running Dijkstra's algorithm only on the graph induced by the nodes in  $\text{Pot-Aff}$  for all nodes  $s \in S(x, y)$  and enlarging the containers results in a new set of consistent target containers. To enlarge the source containers, the same process is done but with reversed edges. A similar process can also be done for the case of decreasing edge weights. We refer the reader to [30] for details on this approach.

The approach presented in [30] has shown good results in the case of train graphs. But for large graphs ( $n > 100,000$ ) and for the domain, where disturbances are frequent, the above mentioned approach is not feasible. Also, the authors assume that the disturbances are permanent. But in our case, where the

disturbances are temporary and disappear after a finite time, even expanding the containers is expensive.

In the light of the observation that for a particular query, only a subset of all the bounding boxes are utilized, with the following chain of reasoning, we develop the notion of *potentially affected* bounding boxes. This characterization can be used to decide whether or not to use the bounding box pruning for a particular query.

**Lemma 7.3.1** *Let  $\Pi$  be a shortest path from  $s$  to  $t$ , and  $\text{out-degree}(s) = 1$  with  $(s, a)$  as the only outgoing edge, then for all  $(u, v) \in \Pi \setminus (s, a)$ , we have  $u, v \in BB((s, a))$ .*

**Proof** We will prove this lemma by contradiction. Assume that there exist an edge  $e_x = (u, v)$  such that  $u, v \in \Pi$  but  $u, v \notin BB((s, a))$ . According to the property of a shortest path, a sub-path of a shortest path is also a shortest path. Hence, the sub-path of  $\Pi$  from  $s$  to  $u$  is also a shortest path and by the definition of consistent containers,  $u \in BB((s, a))$ , which is in contradiction to the assumption.

Similarly, since  $e_x \in \Pi$  is the shortest path from  $u$  to  $v$ , the sub-path of  $\Pi$  from  $s$  to  $v$  is also a shortest path. Again by the definition of consistent containers,  $v \in BB(s)$  - leading to a contradiction to the initial assumption.

Generalizing the above argument to the case where  $\text{out-degree}(s) > 1$  gives the following corollary.

**Corollary 7.3.2** *Let  $\Pi$  be a shortest path from  $s$  to  $t$ , then for all  $(u, v) \in \Pi$ , we have  $u, v \in BB_t(s)$ , where*

$$BB_t(s) = \bigcup_{e \in \text{out-edges}(s) \text{ and } t \in BB(e)} BB(e).$$

Consider the scenario as shown in Figure 7.1. The order of edges' expansion in Dijkstra's algorithm with bounding box pruning to find the shortest path from  $s$  to  $t$  would be  $\langle e_1, e_2, e_3 \rangle$ .

Now, let us assume that because of a disturbance the weight of the edge  $e_3$  is increased by a constant  $c$ . For  $c \leq w(e_4) + w(e_5) - w(e_2) - w(e_3)$ , the resulting path is the shortest path. But if  $c > w(e_4) + w(e_5) - w(e_2) - w(e_3)$ , the resulting path will not be the shortest one.

Since  $t \notin b_4$ , the edge  $e_4$  will not be used during exploration and Dijkstra's algorithm with bounding box pruning will terminate with the incorrect path  $\langle e_1, e_2, e_3 \rangle$  as the shortest path.

The following is an observation we obtain from the example.

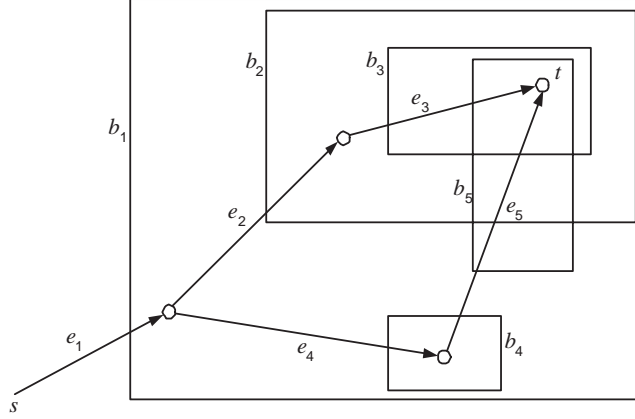


Figure 7.1: Affect of Disturbance on Bounding Boxes

**Observation** Let  $\Pi$  be a shortest path from  $s$  to  $t$ , then in case of an increase in the edge weights of the edges contained inside the bounding boxes  $BB_t(s)$ , it is *possible* that some sub-paths of  $\Pi$  are also affected resulting in  $\Pi$  as not the shortest path.

Based on the above observation, we develop the notion of *potentially inconsistent* bounding boxes.

**Definition 7.3.3** A bounding box  $b$  is potentially inconsistent if there exists an edge  $e \in b$  with  $w(e) > w'(e)$ , where  $w'$  is the weight function in the updated graph.

The converse of the above observation that a shortest path remains to be the shortest, if it is not affected is captured in the following lemma.

**Lemma 7.3.4** Let  $w(\Pi)$  be defined as the weight of the shortest path  $\Pi$  from  $s$  to  $t$  in the graph  $G_c = (V_c, E_c, w)$ . Let  $G'_c$  represents the updated graph with the new weight function  $w'$  such that  $w'(e) \geq w(e)$  for all  $e \in E_c$ . If for all  $e \in \Pi$ , we have  $w(e) = w'(e)$  then  $\Pi$  continues to remain the shortest path from  $s$  to  $t$  in  $G'_c$ .

**Proof** Let  $\mathcal{P}$  be the set of all paths from  $s$  to  $t$  and  $\Pi \in \mathcal{P}$  is the shortest path from  $s$  to  $t$ , i.e.,

$$w(\Pi) \leq w(\pi) \text{ for all } \pi \in \mathcal{P} \setminus \Pi. \quad (7.1)$$

Since  $\Pi$  is not affected,

$$w'(\pi) \geq w(\pi) \text{ for all } \pi \in \mathcal{P} \setminus \Pi. \quad (7.2)$$

Combining the above inequalities, we get  $w(\Pi) \leq w(\pi) \leq w'(\pi)$  for all  $\pi \in \mathcal{P} \setminus \Pi$ .

The following corollary is obtained by a combination of Corollary 7.3.2 and Lemma 7.3.4.

**Corollary 7.3.5** *Let  $\Pi$  be a shortest path from  $s$  to  $t$ . If, for all edges  $(u, v)$ , such that  $u, v \in BB_t(s)$ , we have  $w((u, v)) = w'((u, v))$ , then  $\Pi$  remains to be the shortest path according to the new weight function  $w'$  and hence  $BB_t(s)$  remains to be the set of consistent containers.*

## 7.4 Summary

Dealing with dynamics is an important issue in any navigation system. We presented two models in this chapter for introducing dynamics in our graph. In the first setting, a disturbance due to a traffic jam or some other cause, is reflected in the problem graph as an increase in the weight of the edge corresponding to the affected road. The second model, presents a novel approach, where a disturbance is represented by a geometrical object on super imposed on the graph and affecting all the edges underneath it. Due to the introduction of disturbances, some of the pre-computed information is affected. We presented a concrete characterization of the information that remains *unaffected* due to a disturbance in the graph.



## Chapter 8

# Graph Update - *Off-line* Approach

This chapter presents an approach for dealing with dynamics. We suggest that upon the arrival or termination of a disturbance, the affects are reflected by increasing or decreasing the weights of the affected edges directly and performing the search on the updated graph.

Before actually performing the search, we decide whether to use the pre-computed information for that particular query or not. We exploit the result of Corollary 7.3.5 for this decision. For a query  $q = (s, t, time_s)$ , we check if the  $BB_t(s)$  contains any of the affected edges or not. In case,  $BB_t(s)$  contains any of the affected edge, we declare  $BB_t(s)$  as *potentially inconsistent* (Def. 7.3.3) and the pruning information is not utilized for  $q$ . In the opposite case, we use the guarantee from Corollary 7.3.5 to use the pruning information and obtain a pruned search space for exploration.

In the following, this approach is discussed in the context of the two dynamic models. We present algorithmic issues and geometrical data structures that facilitate the searching of affected edges efficiently.

### 8.1 Graph Update in Individual Edge Dynamics

In this model, the affects of a disturbance are reflected by increasing the weight of the corresponding edge. Recall that a constraint due to a disturbance is defined as  $c(e) = (\delta, t)$ , where  $\delta$  is the amount by which the weight of the edge  $e$  is increased, and  $t$  is the time after which the constraint is considered as *void*.

In order to check whether  $BB_t(s)$  contains an affected edge or not, we need to *window-query* the list of affected edges, i.e., given an axis-parallel rectangle,

**Input** : Weights  $w$ , PQueue  $C$ , Constraint  $c_{new} = (\delta, t)$ , Edge  $e$ ,  
Segment tree  $AffEdges$

**Insert** the tuple  $(e, \delta)$  in  $C$  with priority  $t$

**Insert** the edge  $e$  in  $AffEdges$

$w(e) \leftarrow w(e) + \delta$  /\* Increase the weight of the edge by  $\delta$  \*/

**Algorithm 14:** AddConstraint in Individual Edge Dynamics

search all the edges that are contained in that rectangle. In our case  $BB_t(s)$  are the query rectangles and we need all the affected edges that have *both of their end-points* in the query rectangle. The further operations that we need to perform on the set of constraint is the addition of a new constraint to the set upon the arrival and deletion of the terminated constraint.

The above required operations can best be accomplished by using two different data structures complementing each other. One data structure that allows the addition and deletion of the constraints efficiently on their arrival and termination and a second data structure that allows to perform the *window-query* operation on the affected edges, efficiently.

A priority queue using the termination time value  $t$  as the priority for saving the constraint is a suitable choice for the first data structure. The other constraint informations like the effected edge  $e$  and the increase due to the disturbance  $\delta$  can be saved in the form of a tuple as the node information. This data structure facilitates the efficient removal of constraint on their termination by removing the minimums from the priority queue. For the implementation of the priority queue a (binary) heap data structure is sufficiently fast. It provides the addition and removal of nodes in  $O(\log n)$  time.

For the second data structure, *segment trees* is an appropriate choice. Segment tree is a data structure for saving parallel line segments and with certain enhancements can be used to answer the *window-query*. In their fully dynamic variant, they support *insertion* of new segments in  $O(\log n)$  time and deletions of segments in  $O(\log n \cdot \alpha(i, n))$  time, where  $\alpha(i, n)$  is the extremely slowly growing functional inverse of the Ackermann's function. The window-query operation can be carried out in  $O(\log^2 n + k)$ , where  $k$  is the number of reported segments. Segment tree requires an  $O(\log n)$  space. For the sake of simplicity, we postpone the details of segment trees till Section 8.2.

In the following, the algorithms for the addition and deletion of constraints are formalized in detail. Algorithm 14 adds the constraint  $c_{new}$  on an edge  $e$  in the priority queue of the constraints. It also adds the edge  $e$  in the list of affected edges and increase the weight  $w(e)$  by  $\delta$ . Algorithm 15 removes all the constraints that are terminated by the query time  $time_s$ . A priority queue is represented by PQueue.

Having these data structures and algorithms in hand, we can now formalize

```

Input      : Time  $time_s$ , PQueue  $C$ , Weights  $w$ ,
              Segment tree  $AffEdges$ 
while  $Min(C) \leq time_s$  do
   $(e, \delta) \leftarrow DeleteMin(C)$  /* remove the constraint from  $C$  */
  Remove the edge  $e$  from  $AffEdges$ 
   $w(e) \leftarrow w(e) - \delta$  /* Decrease the weight of the edge by  $\delta$  */

```

**Algorithm 15:** RemoveConstraints in Individual Edge Dynamics

the query algorithm in Algorithm 16. The algorithm first tries to remove all the constraints that might have terminated by the query time. It then search for the validity of the bounding box pruning by *window-querying* the  $AffEdges$  for any affected edge that is contained in  $BB_t(s)$ . An empty  $E_{aff}$  implies the usage of bounding box pruning for the query.

```

Input      : Graph  $G_c$ , Weights  $w$ , Query  $q = (s, t, time_s)$ ,
              PQueue  $C$ , Segment tree  $AffEdges$ 
Output     : ShortestPath  $\Pi$ 
RemoveConstraints( $time_s, C, w, AffEdges$ )
Edges  $E_{aff} \leftarrow \emptyset$ 
for all  $e \in OutEdges(s)$  do
   $E_{aff} = E_{aff} \cup WindowQuery(AffEdges, BB(e))$ 
if  $E_{aff} = \emptyset$  then
   $\Pi \leftarrow DijkstraWithBoundingBox(G_c, w, q)$ 
else
   $\Pi \leftarrow Dijkstra/A*(G_c, w, q)$ 
return  $\Pi$ 

```

**Algorithm 16:** GetShortestPath in Individual Edge Dynamics

The call to the function RemoveConstraints in Algorithm 16 can be removed from the algorithm and made off-line independent to the arrival of the query.

Using a heap implementation for the priority queue, the running time of the function AddConstraint is  $O(\log |C|) + O(|Segmenttree| \cdot I)$ , where  $I$  is the number of insert operations.

**Theorem 8.1.1** *Algorithm 16, when run on the graph  $G_c$  reflecting the constraints  $C$ , terminates with the shortest path in  $\Pi$ , if one exists.*

**Proof** The correctness of the algorithm follows from the guarantee of Corollary 7.3.5 to use the bounding box pruning information in case there are no

affected edges contained inside  $BB_t(s)$ .

## 8.2 Graph Update in Geometrical Objects

In the cases when it is difficult to pinpoint the exact location of the disturbance or when a disturbance affects several paths in a region, it is plausible to define a disturbance in the form of a geometrical object. This geometrical object defines the affected region as the area covered by it. All the edges that are contained inside that object are considered as affected.

Formally, we define disturbances to be objects in a separate layer on top of the graph. For simplicity reasons, we assume these objects to be iso-oriented (axes-parallel) rectangles. We use  $\gamma$  to represent a rectangle and  $\Gamma$  to represent the set of all rectangles. Each rectangle holds two values: the increase on the affected edges due to the disturbance and the duration of that disturbance. The increased weight value due to the disturbance is distributed uniformly on all the edges that are contained inside the rectangle. There can actually be three different kinds of edges that are affected by a disturbance  $\gamma$ :

1. Edges that are completely contained in the object  $\gamma$  i.e.,  
 $E_\gamma^c = \{(u, v) | u \in \gamma \text{ and } v \in \gamma\}$  - referred to as *contained edges*.
2. Edges that are intersecting the object  $\gamma$  at exactly one point i.e.,  
 $E_\gamma^i = \{(u, v) | (u \notin \gamma \text{ and } v \in \gamma) \text{ or } (u \in \gamma \text{ and } v \notin \gamma)\}$  - referred to as *intersecting edges*.
3. Edges that intersect the object  $\gamma$  at exactly two points  $(i, j)$  but having their end points outside of  $\gamma$  i.e.,  $E_\gamma^p = \{(u, v) | u \notin \gamma \text{ and } v \notin \gamma \text{ and } (i, j) = (u, v) \cap \gamma\}$  - referred to as *passing edges*.

Let  $E_\gamma = E_\gamma^c \cup E_\gamma^i \cup E_\gamma^p$ . On the arrival of a disturbance  $\gamma$ ,  $E_\gamma$  has to be found out and the edge weights have to be increased to reflect the current road situation. We discuss two different approaches from two geometrical data structures that facilitate the searching of all three kinds of edges but varies in terms of running time.

- **2D-Range Trees** The first method is to use a geometrical data structure called 2D-range trees. A 2D-range trees is a data structure to query a set of points in Euclidean space for all the points contained inside a rectangular query window. The initial thought was to query for the compressed nodes that are contained inside a query-rectangle and then collect all those edges that are adjacent or incident to those nodes. Since the graph is planar the number of edges is linear in the number of nodes (Euler formula). This

gives a  $O(|V_c| \log |V_c|)$  construction time and  $O(\sqrt{|V_c|} + k)$  time to report  $k$  nodes inside the query-window.

The drawback of this approach is that the *passing edges* cannot be determined because their end points are not in the query-rectangle. They can be reported if we break the compressed edge into original edges and query for the uncompressed points inside the rectangle and find out the passing segments also. But this can *shoot* the running time to  $O(|V| \log |V|)$  for construction and  $O(\sqrt{|V|} + k)$  for searching, which is too much considering  $|V| \gg |V_c|$ .

- **Segment Trees** A segment tree is a geometrical data structure that was mainly designed to answer the *stabbing queries* i.e., to search for all the intervals  $[x, y]$  that contain a query point  $a$ . It can also be casted as a ‘search for all the intervals  $[x, y]$  that  $a$  stabs’. The intervals can be imagined as parallel segments. The construction time for segment trees is  $O(n \log n)$ . The stabbing query can be answered in  $O(\log n + k)$  time, where  $n = |V_c|$  and  $k$  is the number of intersecting intervals reported.

An approach for using segment trees to answer the *Window-Queries* that can also search for the *passing* and *intersecting* edges is presented in [6]. The authors suggest to consider a segment as the diagonal of the smallest enclosing rectangle as shown in Figure 8.1.

Two different segment trees are maintained: one for the vertical edges of the rectangles and one for the horizontal edges. This reduces a window-query operation to the search for all the vertical and horizontal segments that are stabbed by the corners of the query-rectangle. The result is the set of rectangles that are intersected by the query-rectangle.

Once we have those rectangles, in linear time we can determine whether the segments they are enclosing also intersect with the query-rectangle or not. The running time for this method is  $O(\log^2 n + k)$  with an  $O(n \log n)$  space consumption.

Since we are dealing with a semi-incremental partially dynamic problem (Section 7.1) and do not allow any addition or deletion in the edges or node set, the segment tree structure once constructed for the compressed edges can be used through out the computation. This puts the  $O(n \log n)$  construction time in the pre-processing (Chapter 1) time of the system.

Let  $\delta$  denotes the increase due to a disturbance  $\gamma$ . Then  $\delta$  has to be distributed uniformly on the edges in  $E_\gamma$ . But each edge should get only the fraction of amount up to which it is enclosed in  $\gamma$ . This enclosure can be calculated by multiplying the per-unit distance cost of each edge based on  $d_c$  by the length of each edge that is inside  $\gamma$ .

Algorithm 17 formalizes the procedure for calculating the affect of a disturbance  $\gamma$  on a set of edges  $E_\gamma$ . These amounts are denoted by a function  $\delta_\gamma : E_\gamma \rightarrow \mathfrak{R}$ . The function `IntersectionPoints` calculates the intersection

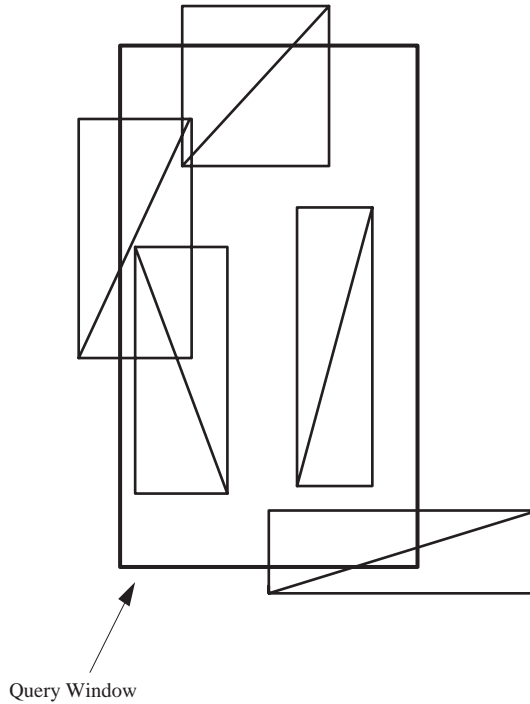


Figure 8.1: Conversion of segments to rectangles

points of an edge  $e$  with the rectangle  $\gamma$ . In case  $e$  does not intersect  $\gamma$  the source and target of  $e$  are returned.

On the arrival of disturbances, the rectangles should be efficiently stored in order to be removed later when their duration is finished and their affects have to be reversed. We suggest to use a priority queue data structure to save  $\Gamma$  with the termination time values as *priority*. In order to keep track of the increase on individual edges we store the individual increase values as individual edge constraints  $c$ . Algorithm 18 formalizes the algorithm for adding a constraint in the form of a rectangle  $\gamma$  to the graph  $G_c$ .

Algorithm 19 removes all the constraints that are terminated by the time  $time_s$ .

The query model is almost the same as in Algorithm 16 except that here we characterize the  $BB_t(s)$  as invalid by looking for their intersection with the rectangles representing the disturbances. This problem is studied in the domain of Computational Geometry as the *rectangle intersection problem* [25] and can be solved in time  $O(n \log n + k)$  using  $O(n \log n)$  space, where  $n = |BB_t(s)| + |\Gamma|$  and  $k$  is the number of reported intersections. The algorithm is based on sweep-

```

Input      : Graph  $G_c$ , Edges  $E_\gamma$ , Real  $\delta$ , Rectangle  $\gamma$ 
Output    :  $\delta_\gamma$ :  $\delta$  values for individual edges affected
for all  $e \in E_\gamma$  do
  |  $x \leftarrow \|\text{Target}(e) - \text{Source}(e)\|_2$ 
  |  $(i, j) \leftarrow \text{IntersectionPoints}(e, \gamma)$ 
  |  $\text{Enclosed}(e) \leftarrow \frac{x}{d_c(e)} \times \|j - i\|_2$  /*Amount of edge  $e$  enclosed in  $\gamma$ */
 $\text{Enclosed}(E_\gamma) \leftarrow \sum_{e \in E_\gamma} \text{Enclosed}(e)$ 
for all  $e \in E_\gamma$  do
  |  $\delta_\gamma(e) \leftarrow \frac{\delta}{\text{Enclosed}(E_\gamma)} \times \text{Enclosed}(e)$ 

```

**Algorithm 17:** Calculation of Increase due to a rectangle  $\gamma$ 

```

Input      : Graph  $G_c = (V_c, E_c, d_c, T_c)$ , Weights  $w$ , PQueue  $C$ ,
              Constraint  $c_{new} = (\delta, t)$ , Rectangle  $\gamma$ , PQueue  $\Gamma$ 
Output    : PQueue  $C$ 
 $E_\gamma \leftarrow \text{WindowQuery}(E_c, \gamma)$ 
 $\delta_\gamma \leftarrow \text{CalculateIncrease}(\delta, E_\gamma)$ 
for all  $e \in E_\gamma$  do
  |  $w(e) \leftarrow w(e) + \delta_\gamma(e)$  /* Increase the edge weight */
  | Constraint  $c \leftarrow (\delta_\gamma(e), t)$ 
  | Insert the tuple  $(e, c)$  in  $C$  with priority  $t$ 
Insert  $(\Gamma, \gamma, t)$  /* Add the  $\gamma$  in the set of rectangles  $\Gamma$  */

```

**Algorithm 18:** AddConstraint in Disturbances as Geometrical Objects

line principle and uses a combination of range and segment trees as the *line-status* data structure. The space complexity can be reduced to  $O(n)$  by using interval trees in place of segment trees.

In fact, we can exploit a special case. We have two different kinds of rectangles and we only want to know if one type intersects with the other or not. This problem is mainly studied as *Red-Blue* rectangle intersection problem [1]. The running time is the same  $O(n \log n + k)$  as for the general problem, where  $n$  is the number of rectangles of both types. We maintain two different sweep structures: one for each type and check for the intersection of rectangles in one structure with the rectangles in the other. Since, we are only interested in deciding if the two types intersect or not, the algorithm can be stopped at the first point of intersection. This reduces the running time to a mere  $O(n \log n)$ .

```

Input      : Time  $time_s$ , Weights  $w$ , PQueue  $C$ , PQueue  $\Gamma$ 
while  $Min(\Gamma) \leq time_s$  do
  |  $DeleteMin(\Gamma)$  /* Remove all the terminated rectangles */
while  $Min(\Gamma) \leq time_s$  do
  |  $(c = (\delta, t), e) \leftarrow DeleteMin(C)$ 
  |  $w(e) \leftarrow w(e) - \delta$ 

```

**Algorithm 19:** RemoveConstraints in Disturbances as Geometrical Objects

### 8.3 Summary

This chapter presented an approach for dealing with dynamics. The graph is updated directly upon the arrival of a disturbance. The search is then done on the updated graph. Before actually performing the search, we decide whether or not to use the bounding box pruning for a particular query. For the Individual edge model, we exploited segment trees to efficiently store the affected edges and to perform *window - query* on them. In case the window-query on the affected edges with the  $BB_t(s)$  as the query rectangles, returns an edge, the pruning information is not used. For the Disturbances as Geometrical Object model, the solution to the problem of *rectangle-intersection* is utilized to decide whether the pruning information is valid for a particular query or not. Using proper data structures, we have managed to obtain feasible run-time bounds in both the models.

## Chapter 9

# Exploration-Time Checking - *On-Line* Approach

We observe two interesting phenomena in a dynamic environment:

1. It is possible that some of the constraints have terminated and no longer be there by the time the mobile object will reach that area.
2. If a shortest path between two points is not affected by a disturbance, it remains to be the shortest one in the affected graph.

These observations suggest a search procedure, where we continue exploring using our search algorithm along with the pre-computed information until we reach an affected edge. If we do not encounter any affected edge and reach the target, we declare the encountered path as the shortest path.

Subsequently, the travel time for the shortest path is maintained along with the shortest distance. When an affected edge  $e = (u, v)$  is encountered in the exploration, we check whether the constraints would be valid by the time the edge is going to be traversed in reality. We named this approach as *Exploration-time checking*. Since it delays the affect of disturbances on the edges until the edge is actually traversed, we characterize this approach as *on-line* as opposed to the *off-line* graph update approach.

This chapter discusses this approach in the context of both of the dynamic models. In Section 9.1, we discuss the general search procedure that exploits the above observations. Section 9.2 discusses the details of transforming the general search procedure in the *individual edge dynamic model*. Section 9.3 discusses this approach for the *geometrical object model*, where a disturbance is defined as a geometrical object on top of the graph and affecting the area underneath it.

## 9.1 General Search Strategy

Let us first formalize the above observations. Let  $time[u]$  denote the time needed to reach the node  $u$  on the shortest path starting from  $s$ . Let  $C(e)$  denote the list of constraints corresponding to the edge  $e = (u, v)$ . We say that a constraint  $c = (\delta, t) \in C(e)$  is *valid* if  $time_s + time[u] < t$ , where  $\delta$  is the weight increase for  $c$  and  $t$  is the time after which  $c$  disappears.

Lemma 7.3.4 formalizes the second observation and gives us a guarantee of the validity of pre-computed information if the shortest path is not affected. Utilizing this guarantee, we can maximize usage of the pre-computed information to prune the search space.

We suggest the following general search procedure based on the above observations:

While searching for a shortest path along with bounding box pruning, and before exploring an edge  $e$ , check whether  $e$  is affected or not.

- *if*  $e$  is affected *then* check whether the constraints would be valid by the time  $e$  would be traversed.
  - *if* constraints are valid *then* declare the search procedure as invalid and use standard Dijkstra or A\*.
  - *else* continue.
- *else* continue.

## 9.2 Exploration-Time Checking in Individual Edge Dynamics Model

Based on the guarantee provided by Lemma 7.3.4, we continue exploring the updated graph until we encounter an edge whose weight is increased due to a disturbance. If we do not encounter such an edge by the time we reach the target node, we declare the searched path as the shortest one. In the other case our search procedure fails and returns *invalid*.

Individual constraints can be saved in a mapping  $C : E \rightarrow \mathcal{P}(\mathbb{R} \times Time)$  that given an edge  $e$ , returns a list of all the constraints on  $e$ . A hash table provides the most suitable data structure for this purpose.

In Algorithm 20, we present the modified fragment of Dijkstra’s algorithm with bounding box pruning that checks for the encounter of an affected edge during exploration. The time when a node  $u$  is going to be traversed using the shortest path is maintained in the variable  $t_{min}[u]$ . It utilizes the function `CalcNewWeight` (Algorithm 21) that calculates the new weight information for an edge based on the constraints associated with the edge and the time when

<b>Input</b>	: Graph $G_c = (V_c, E_c, d_c, T_c)$ , Weights $w$ , Start node $s$ , Target node $t$ , Constraints $C$
<b>Output</b>	: Shortest path $\Pi$ / <b>invalid</b>

```

1 for all  $v \in \text{Adjacent}(u)$  do
2   if  $t \in \text{BB}[(u, v)]$  then
3      $\text{newWeight} \leftarrow \text{CalcNewWeight}(w(u, v), (u, v), t_{\min}[u], C)$ 
4     if  $\text{newWeight} = w(u, v)$  then
5       if  $\text{dist}[v] > \text{dist}[u] + \text{newWeight}$  then
6          $\text{dist}[v] \leftarrow \text{dist}[u] + \text{newWeight}$ 
7          $\text{DecreaseKey}(Q, (v, \text{dist}[v]))$ 
8          $t_{\min}[v] = t_{\min}[u] + T_c((u, v))$ 
9          $\Pi[v] \leftarrow (u, v)$ 
10      else
11        return invalid
12        /* Bounding Box approach turned out to be invalid for this
           query */

```

**Algorithm 20:** Bounding box pruning in the presence of affected edges.

that edge is going to be traversed. A different value of the weight returned from `CalcNewWeight` suggests the possibility of an affected shortest path. In such case, we terminate our search and declare bounding box pruning as **invalid** for that particular query.

Algorithm 22 presents the modified fragment of basic Dijkstra algorithm that search for a shortest path in the updated graph. This algorithm is needed once the search procedure in Algorithm 20 returns **invalid**.

Terminated constraints can be removed by a linear search through the lists and can be invoked when the frequency of queries is minimal. Alternatively, utilizing extra  $O(|C|)$  space, these constraints can be maintained in a priority queue as discussed in the previous chapter. It can decrease the running time for the removal of terminated constraints.

The best case of this approach is when we do not encounter any affected edge. The result is a pruned search space to explore. The worst case is when we were just at one edge distance from the target and the last edge turned out to be an affected one. The search must then be repeated using Dijkstra or A\* algorithm without any pruning information.

```

Input      : real  $w_{old}$ , Edge  $e$ , Time  $currentTime$ , Constraints  $C$ 
Output    : real  $newWeight$ 
real  $newWeight \leftarrow w_{old}$ 
for all  $c_i = (\delta_i, t_i) \in C(e)$  do
  if  $t_i \leq currentTime$  then
     $newWeight \leftarrow newWeight + \delta_i$ 

```

Algorithm 21: CalcNewWeight

```

Input      : Graph  $G_c = (V_c, E_c, d_c, T_c)$ , Weights  $w$ , Start node  $s$ ,
              Target node  $t$ 
Output    : Shortest path  $\Pi$ 
1 for all  $v \in Adjacent(u)$  do
2    $newWeight \leftarrow CalcNewWeight(w(u, v), (u, v), t_{min}[u])$ 
3   if  $dist[v] > dist[u] + newWeight$  then
4      $dist[v] \leftarrow dist[u] + newWeight$ 
5     DecreaseKey( $Q, (v, dist[v])$ )
6      $t_{min}[v] = t_{min}[u] + T_c((u, v))$ 
7      $\Pi[v] \leftarrow (u, v)$ 

```

Algorithm 22: Dijkstra's algorithm in the presence of affected edges.

### 9.3 Exploration-Time Checking in Geometrical Objects

Before we discuss the details of how to incorporate the exploration-time checking in the geometrical object model, we allow a minor change to our model. We say that now the increase  $\delta$  due to a disturbance will not be uniformly distributed but will represent the increase on individual edges contained inside the geometrical object.

In the model, where the disturbances are represented by a geometrical object the condition in Lemma 7.3.4 can be checked by checking at every explored edge  $e$  whether it is affected by a geometrical object or not. We can, in fact, view this problem as a *window-query* problem. Assume that a disturbance rectangle is actually a set of two vertical and two horizontal segments. Also, let us assume that there exist an axis-parallel rectangle  $r$  such that the edge  $e$  is one of the diagonal of  $r$  (see Figure 9.1). We now ask for all the vertical and horizontal segments that intersect  $r$ .

At the first glance, this problem can be solved by using segment trees. But there is a deficiency in segment trees. They are semi-dynamic because they

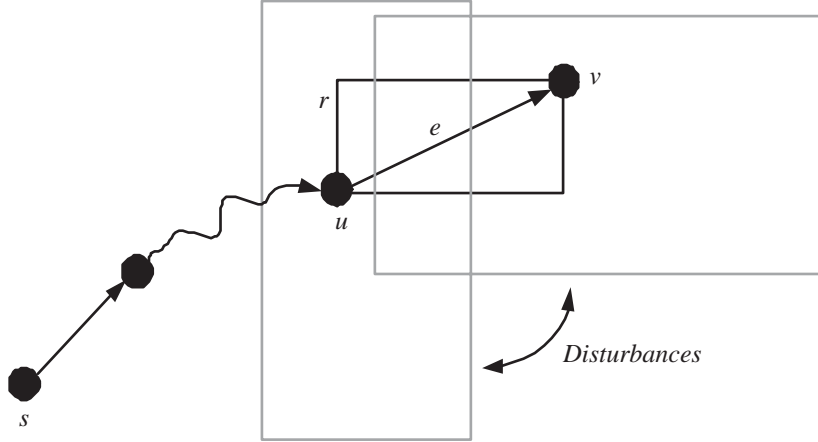


Figure 9.1: Exploration time checking.

should be made from a known universe of values and hence do not allow free insertions and deletions of intervals. In our case, we need a data structure that is fully dynamic, i.e., in which we can insert and delete segments - disturbances come and go with the time.

One of the extension of segment trees to dynamic segment trees is due to [28]. The authors introduced fully dynamic segment trees that allow insertions in  $O(\log n)$  time and deletions in  $O(\log n \cdot \alpha(i, n))$  time, where  $\alpha(i, n)$  is the extremely slowly growing functional inverse of the Ackermann's function. For all practical purposes the value of  $\alpha(i, n)$  is constant. The authors use a *union-copy* structure which is an extension of *union-find* structure with *copy* operation.

The algorithms library LEDA provides an implementation of segment trees that is also fully dynamic. It is based on  $BB[\alpha]$  trees [22] with insertion and deletion in  $O(\log^2 n)$  time.

Having dynamic segment trees in hand, we now use the same procedure as described in the previous chapter for window-query. The rectangle  $r$  is now the query rectangle and we ask for all the horizontal and vertical segments that cross  $r$ . Let  $S_{ver}$  denotes the set of all the vertical segments that intersects  $r$  and  $S_{hor}$  is the set of all horizontal segments that intersects  $r$ . Note that here by the term *intersect* does not only mean that the segments intersect the boundary of  $r$  but a segment having both of its end points in  $r$  is also counted as intersecting  $r$ .

The window-query can be carried out in time  $O(\log^2 |\Gamma| + k)$ , where  $k$  is the number of reported segments and  $\Gamma$  is the set of disturbances [6]. Once we have  $S_{ver}$  and  $S_{hor}$ , we now check whether they actually intersect the edge  $e = (u, v)$  or not and in case they do, we introduce their affect on the edge weight. The

time  $t_{min}[u]$  to reach  $u$  using the shortest path is utilized before introducing the affect of the disturbance to the edge in order to check whether the disturbance would not be terminated by the time the path would be traversed in reality.

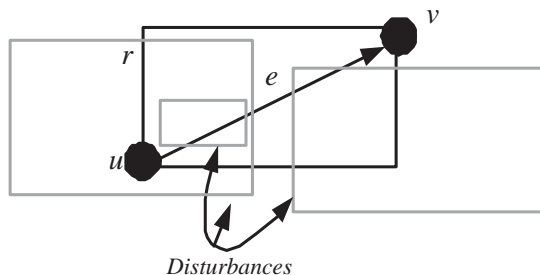


Figure 9.2: Types of disturbances.

Since, it is possible that a disturbance rectangle intersects  $e$  at two points, once through one of its horizontal segment and once through its vertical segment (see Figure 9.2), before actually introducing its affect on the edge weight, we have to check whether we have not already introduced the affect of the disturbance through another segment. Maintaining an array of flags corresponding to disturbances and marking the disturbance entry as 1 once one of its segment has been used to increase the edge weight, provides a sufficiently fast solution to this problem.

But this array has to be re-initialized for every edge which produces a bottleneck in the run-time of the overall method. The re-initialization of this array for every edge can be avoided by maintaining a time counter and increasing its value for every edge encounter. In the array instead of saving 0 or 1, we save the value of the counter. The check, whether a disturbance is already used or not can then be carried out by checking whether the corresponding array entry contains the current value of the counter or not.

We can in fact, re-use the algorithms presented in the last section for the individual edge dynamics model by simply changing the `CalcNewWeight` function to use the approach presented above.

The best case of this approach is present when we do not encounter any of the affected edge till we reach the target node. The running time we obtain in this case is  $O(n \log n + m \cdot (\log^2 |\Gamma| + K))$ , where  $n = |V_c|$ ,  $m = |E_c|$ , and  $K$  is the total number of disturbances reported during exploration. Note that it can be the case that  $K > 0$ , but still we reach the target without encountering any affected edge. The reason is that there might be some disturbances that intersects  $r$  but not  $e$ .

## 9.4 Summary

There were two main observations that motivated to make use of the pre-computed information as much as possible. We observed that a shortest path remains to be the shortest path, if it is not affected by any disturbance. We further observed that it is possible that some of the disturbances might be terminated by the time the shortest path would be traveled in reality. In the light of these observations, in this chapter, we formalized an approach where we explore the pruned search space, utilizing the bounding boxes, until we encounter an edge which is affected by a disturbance. In case we encounter such an edge, we back track to the start node and restart the exploration without using any pre-computed information. This approach is formalized in the context of both of the dynamics model. For the disturbances as geometrical object model, in order to find out whether an edge is affected by a disturbance or not, we utilized the efficient *window-query* method through dynamic segment trees. The correctness of the presented approach in both the models follows immediately from Lemma 7.3.4.



# Chapter 10

## Architecture of GPS-Route

A part of the methodology discussed in this report is implemented in a running system called **GPS-Route**. This chapter presents an overview of the system. We have divided the whole system into three different parts: interface, preprocessor, and shortest path algorithm.

This chapter starts with a description of the general programming technique we have used in the design. Section 10.2 discusses the interface component along with the different kinds of interfaces that can be used. Preprocessor is discussed in Section 10.3 and the component that deals with the management of different shortest path algorithms is discussed in Section 10.4.

### 10.1 General design

The whole system is implemented using **C++** with **GCC-2.95** as the compiler. For a stable implementation of computational geometry algorithms, we have used **LEDA** (Library for Efficient Data Structures and Algorithms) version 3.6.1.

Speed and extensibility were the main motivations behind the design. **C++** provides two methods that facilitate an extensible design: abstract classes and **C++** templates. We adapted the template-based design to decrease the running time of the system.

For the sake of coherence, we designed common interfaces for individual components. This gives the advantage that based on the future needs and research, new components implementing the interfaces can be easily integrated in the system.

The main component of the system can be initialized as a class that takes three sub-components as the template parameters:

Figure 10.1: CGI-based light-weight interface of GPS-Route

```
GPSRoute< Interface,
        Preprocessor,
        ShortestPathAlgorithm
>
```

## 10.2 Interface

All the interactions between the user and the system are done through a separate component called Interface. An Interface can be of three different types: Standard, VEGA, and Network. These interfaces are discussed in the following sub-sections.

### 10.2.1 Standard Interface

The standard interface is a text-based console input/output interface. It takes a file name for the GPS trace file and query points as latitudes and longitudes of source and target.

We also designed a CGI(Common Gateway Interface) based wrapper on top



Figure 10.2: Visualization of a trace using VEGA.

of textual interface. It is implemented using `Perl` and can be accessed through a world wide web. The CGI-based interface is also most suitable for the PDA's or other mobile devices where memory and downloading time is of significant importance. Figure 10.2.1 displays a screen-shot of the web page showing the CGI-based interface.

### 10.2.2 VEGA Interface

For the visualization purposes, we have adapted an already existing interface called VEGA. VEGA stands for Visualization of Efficient Geometric Algorithms. As the name suggests, it was originally designed for the visualization of the working of geometric algorithms. It is a client-server based system. The algorithms that have to be visualized stay on the server while a Java-Applet based web-interface shows the visualization of the algorithms. It provides a VCR-like panel to navigate between different steps of an algorithm and an easy-to-use interface for scaling and coloring different objects. Algorithms can be implemented in

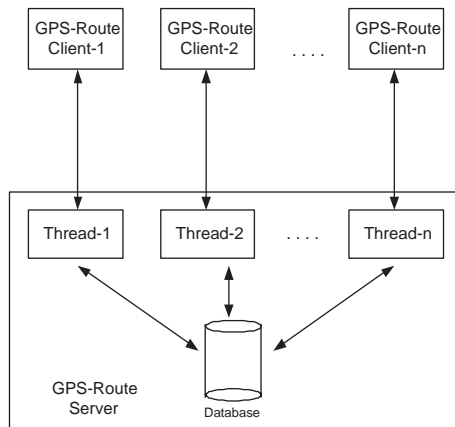


Figure 10.3: Network Interface

any language. The algorithms just need to implement a textual protocol with which VEGA server can control their execution.

We use VEGA for the visualization of the maps and the shortest paths between the query points. In Figure 10.2 the visualization of a graph, in gray, along with the shortest path, in black, from the extreme right point to the extreme left point is shown. Unfortunately, because of the restriction of maximum text-size in a Java text box, the VEGA client cannot be used to provide a trace file as input. Currently, we use VEGA only for the visualization of already existing traces on the server side. For the future, we have planned to provide another CGI-based front-end that takes a trace file and upload it on the server. The uploaded trace file can then be used for visualization through VEGA.

### 10.2.3 Network Interface

A Network interface consists of a GPS-Route server and its self-executable client. The client takes a file as input and transfers it to a server through TCP/IP protocols. The server working on thread architecture, generates a new thread for every client. The traces are saved on the server after preprocessing. These traces can then be queried for the shortest paths.

This interface is also best suited for dynamics and would be able to keep track of the disturbances. An individual client would be able to upload the information about the disturbances. The shortest path can then be searched in the presence of these disturbances.

This interface is only in its design stage and not implemented yet. The need for this interface arises because both CGI and VEGA do not provide support

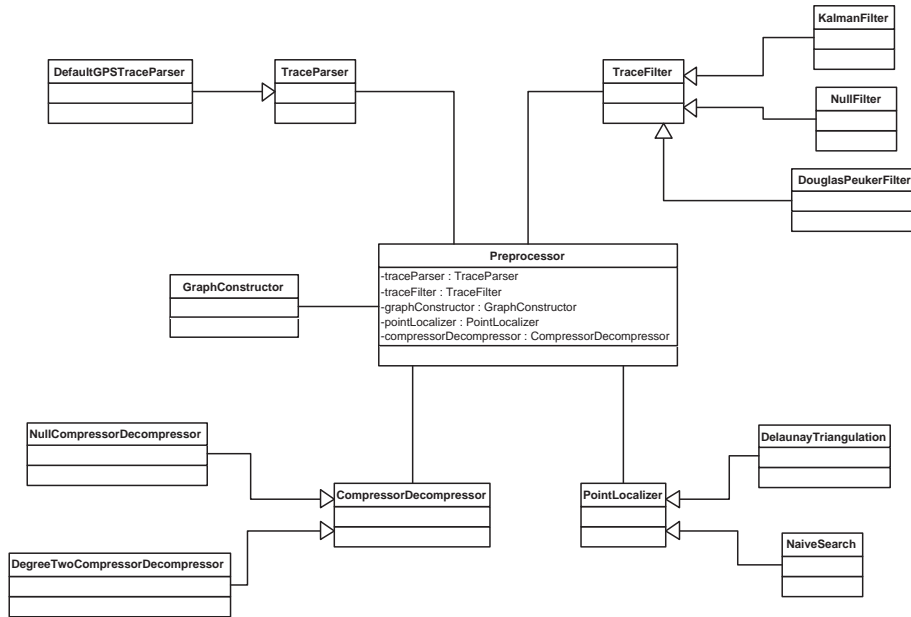


Figure 10.4: Class diagram of Preprocessor

for saving the sessions. Sessions are necessary in order to save the preprocessed information on the server and query it multiple times. Figure 10.3 shows a block diagram of the Network based interface.

## 10.3 Preprocessor

The preprocessor component is responsible for the preprocessing of traces. It itself consists of 5 sub-components:

- Trace parser: to parse the trace file.
- Trace filter: to filter/round the traces - Douglas-Peucker algorithm or Kalman filter.
- Graph constructor: to construct graph from traces - line-sweep algorithm.
- Point localizer: to answer the nearest neighbor queries - Delaunay Triangulation or naive search.
- Graph compressor: to compress a graph to fewer number of nodes.

A preprocessor is implemented as a class that takes these five components as template parameters. A typical initialization is as follows.

```
Preprocessor < DefaultGPSTraceParser,
              DouglasPeuckerTraceFilter,
              GraphConstructor,
              DTPointLocalizer,
              DegreeTwoCompressorDecompressor
              >
```

Each individual component implements the functions defined in its corresponding abstract class, e.g: we implemented an abstract class called `PointLocalizer` that contains the signature of an unimplemented method `Point getClosestPoint(Point)`. All the sub-classes of `PointLocalizer` must implement this method. In the `Preprocessor` class only this method is used to communicate with the `PointLocalizer`. The advantage of this approach is that any other point localization structure that implements the `getClosestPoint` method can be used as a template argument to the `Preprocessor` class.

## 10.4 Shortest-Path-Algorithm

For the implementation of shortest path algorithms especially the bounding box pruning, we have adapted the already-designed routines [29]. The authors have employed Mixin-based programming for their implementation. Mixin-based programming allows to provide a super-class as the template parameter, e.g.,

```
template <class Super>
class Mixin : public Super{
    /* mixin body */
}
```

Mixin-based programming facilitates an elegant layer-based design of a component. The implemented routines for the shortest path algorithms are carefully designed and provides also facility to collect the information of a *run*, e.g., the number of expanded nodes for a search.

A typical initialization of the mixin-based class to use the bounding box pruning information is as follows:

```
BoundingBoxDijkstra <
  SelectionDijkstra <
    CoordinateDijkstra <
      PathDijkstra <
        TargetDijkstra <
```

```
CountingDijkstra <  
Dijkstra> > > > >
```

For the sake of coherency, we designed an abstract class called `SPAlgorithm` that provides the signatures of necessary methods. To adapt the existing implementation of bounding box pruning algorithm, we developed a wrapper class on it that implements the methods of `SPAlgorithm`. The resulting class can then be used as a template argument to the main `GPSRoute` class.

## 10.5 Summary

This chapter presented a brief overview of the design of `GPS-Route`. Speed and extensibility were the main motivations behind the design. We divided the whole functionality into three separate components to obtain an efficient, elegant and easily maintainable design. The interface component is responsible for all the interactions with the system. The Preprocessor component is responsible for the preprocessing of provided traces ranging from the filtering to the construction of graph and construction of point localization structure.



# Bibliography

- [1] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems (extended abstract). In *Symposium on Discrete Algorithms*, pages 685–694, 1998.
- [2] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *Transactions on Computing*, 28:643–647, 1979.
- [3] F. Chin, J. Snoeyink, and C.-A. Wang. Finding the medial axis of a simple polygon in linear time. In *International Symposium Algorithms Computation (ISAAC)*, Lecture Notes in Computer Science, pages 382–391. Springer, 1995.
- [4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
- [5] Peter H. Dana. The global positioning system overview. <http://www.colorado.edu/geography/gcraft/notes/gps/gps.html>, 2000.
- [6] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 1997.
- [7] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.
- [8] D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points. *The Canadian Cartographer*, 10:112–122, 1973.
- [9] S. Edelkamp. Memory limitation in artificial intelligence. In J. F. Sibeyn P. Sanders, U. Meyer, editor, *Memory Hierarchies*, volume 2625 of *Lecture Notes in Computer Science*, pages 233–250. Springer, 2003.
- [10] S. Edelkamp, S. Jabbar, and T. Willhalm. Accelerating heuristic search in spatial domains. In *17. Workshop on New Results in Planning, Scheduling and Design*, pages 1–20, 2003.

- [11] S. Edelkamp, S. Jabbar, and T. Willhalm. Geometric travel planning. In *IEEE International Conference on Intelligent Transportation Systems (ITS)*, pages 964–969, 2003.
- [12] D. Eppstein, Z. Galil, and G. Italiano. Dynamic graph algorithms. *CRC Handbook of Algorithms and Theory of Computation*, 1997.
- [13] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7:381–413, 1992.
- [14] S. Gutmann. Markov-Kalman localization for mobile robots. In *In International Conference on Pattern Recognition (ICPR)*, 2002.
- [15] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.
- [16] J. Hershberger and J. Snoeyink. An  $O(n \log n)$  implementation of the Douglas-Peucker algorithm for line simplification. *ACM Computational Geometry*, pages 383–384, 1994.
- [17] Ch. A. Hipke and S. Schuierer. Vega—a user-centered approach to the distributed visualization of geometric algorithms. In *Computer Graphics, Visualization and Interactive Digital Media (WSCG)*, pages 110–117, 1998.
- [18] C. Icking, R. Klein, P. Koellner, and L. Ma. Java applets for the dynamic visualization of voronoi diagrams. In *Computer Science in Perspective: Essays Dedicated to Thomas Ottmann*, volume 2598 of *LNCS*, pages 191–205. Springer, 2002.
- [19] R. E. Kalman. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering, Transactions of the American Society of Mechanical Engineers*, 82(1):35–45, 1960.
- [20] L. J. Levy. The Kalman filter: Navigation’s integration workhorse. *GPS World*, 8(9):65–71, 1997.
- [21] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *European Symposium on Algorithms (ESA)*, 2002.
- [22] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
- [23] K. Mulmuley. A fast planar partition algorithm. *Journal of Symbolic Computation*, 3–4(10):253–280, 1990.
- [24] J. Pearl. *Heuristics*. Addison-Wesley, 1985.
- [25] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer, 1985.

- [26] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 1995.
- [27] P. Su and R. L. S. Drysdale. A comparison of sequential Delaunay triangulation algorithms. In *Symposium on Computational Geometry*, pages 61–70, 1995.
- [28] M. J. van Kreveld and M. H. Overmars. Union-copy structures and dynamic segment trees. *Journal of the ACM*, 40(3):635–652, 1997.
- [29] D. Wagner and T. Willhalm. Geometric speed-up techniques for finding shortest paths in large sparse graphs. In Giuseppe Di Battista and Uri Zwick, editors, *Proc. 11th European Symposium on Algorithms (ESA 2003)*, volume 2832 of *LNCS*, pages 776–787. Springer, 2003.
- [30] D. Wagner, T. Willhalm, and C. Zaroliagis. Dynamic shortest path containers. In Alberto Marchetti-Spaccamela, editor, *Proc. Algorithmic Methods and Models for Optimization of Railways 2003*, Electronic Notes in Theoretical Computer Science, 2003. To appear.
- [31] A. Wilson, editor. *Galileo: The European Program for Global Navigation Services*. 2003.