Auto-completion Algorithms for Geocoding Systems

Master Thesis

Vlad Mînzatu

Auto-completion Algorithms for Geocoding Systems

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Vlad Mînzatu
born in Roman, Romania

Algorithmics Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

TomTom International B.V.
Oosterdoksstraat 114, 1011 DK
Amsterdam, the Netherlands
`www.tomtom.com`

Auto-completion Algorithms for Geocoding Systems

Author: Vlad Mînzatu
Student id: 4117816
Email: v.minzatu@student.tudelft.nl

Abstract

This thesis presents solutions to the problem of implementing auto-completion functionality, which is currently lacking from TomTom geocoding systems. Auto-completion is a highly desirable feature enabling users to perform their task more effectively, by providing suggestions for completing their queries as they start to type.

Implementing such functionality in the specific context of geocoding systems raises several constraints and requirements not dealt with in related literature. After identifying all the requirements, this thesis will present the overall approach and the algorithms used to meet all of them, including a novel algorithm for offering location biased query completion suggestions.

The thesis will end with conclusions and ideas for future work, but not before an experimental analysis which reveals some interesting characteristics of the system and provides guidelines on getting the best performance for the system by properly adjusting the data that it operates on.

Thesis Committee:

University supervisor:	Prof. Dr. C. Witteveen, Faculty EEMCS, TU Delft
Company supervisor:	A. Sutherland, TomTom International B.V.
Committee Member:	Dr. F.A. Kuipers, Faculty EEMCS, TU Delft
Committee Member:	Dr. T.B. Klos, Faculty EEMCS, TU Delft

Preface

This thesis is the result of the research I have done in implementing auto-completion functionality for geocoding systems as part of my internship within TomTom International B.V. in Amsterdam.

I started this internship in July 2011, initially without a research topic. Only after working in the Software Development team developing the Andorra Geocoding System for about the first two months of my internship, it was suggested that auto-completion functionality would be a highly desired addition to the geocoder, so I started to research existing approaches to implement such functionality, while still working on current tasks for the Andorra geocoder, in preparation for its launch.

Developing the algorithms presented in this thesis was a gradual process, starting from desired properties derived from the functionality of some of the most advanced existing solutions on one hand, and existing approaches to addressing the problem, which only covered part of these desired properties, on the other. In the end, I believe a suitable meeting point was reached: the resulting algorithms draw inspiration from existing approaches, but also offer extensions that enable the successful implementation of all the desired properties.

I have several people to thank for their help and support. First, I would like to thank my University supervisor, Prof. Dr. Cees Witteveen, for the constant feedback and guidance in the process of writing this thesis.

I would also like to thank all the people that I have had the pleasure of working with during my internship in TomTom for their support during my research, as well as for the pleasant and very useful experience of working on a project as interesting as the Andorra geocoder. Here I would like to extend special thanks to Martine Woudstra for her insightful notes on my thesis early on. And last, but definitely not least, I would like to thank Jason Griffin, who offered me the internship within TomTom, and thus the chance to gain useful practical experience as well as to research an interesting topic for my master thesis.

Vlad Mînzatu
Amsterdam, the Netherlands
April 13, 2012

Contents

Chapter 1

Introduction

This chapter will introduce the basic notions that create the context of this thesis project and will present the problem statement from a business point of view. We will start with an introduction to TomTom and the company's main products. We will then discuss the Andorra Geocoding System and its role within the company. This will create a proper foundation for an exact formulation of our problem statement speaking strictly from the perspective of TomTom's business need. Future chapters will deal with translating this into a research problem and discussing the proposed solutions.

1.1 TomTom and the Andorra Geocoder

TomTom was founded in 1991 by Peter-Frans Pauwels, Pieter Geelen, Harold Goddijn and Corinne Vigreux who are all currently still working within TomTom. Having studies in Business and Computer Science completed at the University of Amsterdam, Peter-Frans Pauwels founded, along with his former university colleague Pieter Geelen, a company called Palmtop Software, with the initial aim of building general software solutions for mobile devices. Harold Goddijn, who also studied economics at the University of Amsterdam and Corinne Vigreux who has studies in International Affairs, soon joined the company, which was later renamed to TomTom.

Until 1996, the company developed a number of business-to-business applications for mobile devices such as bar-code reading, meter reading and order-entry systems, before shifting focus to developing consumer software products for personal digital assistant devices (PDAs). By 1998, TomTom was established as a market leader in PDA software, creating a number of consumer applications for PDAs, such as the EnRoute (later renamed RoutePlanner) and Citymaps navigation applications.

In 2001, as more accurate GPS satellite readings became available, TomTom looked towards in-car navigation as a major opportunity for innovation. The company's first navigation (software) product for PDAs, the TomTom Navigator, was launched in 2002. In 2004 the Navigator targeted for PalmOS was launched, based on a cross-platform navigation engine still used in current products. The company's first stand-alone portable navigation device (PND), the TomTom GO, was introduced in March 2004 and marked a turning point in TomTom’s story. It featured a 3.5” 320x240 touchscreen, a 200 MHz CPU, 32 MB of RAM and an integrated SD reader, at a very competitive price (at the time) of £499. The maps provided with the device came from Tele Atlas, a Netherlands-based company specialized in delivering digital maps and other dynamic content for navigation and location-based services. The TomTom GO met a need for a portable fit-for-purpose navigation device that was simple to use, affordable and worked better than any other navigation solution on the market and effectively defined a new category of consumer electronics: the PND. Its success was immediate, and by the end of the year of its launch, sales of the GO device formed 60% of the company’s revenue. In 2007, TomTom took over their map data provider, Tele Atlas, after a bidding war with United States-based rival Garmin. The final accepted offer was worth €2.9 billion.

Although TomTom has become nearly synonymous to the PND in the company’s recent history, due to its success in this active market segment, the company’s product offering is much wider and also includes in-dash infotainment systems, fleet management solutions, maps and real-time services, including the award winning HD Traffic, which makes the most up-to-date traffic information available, in order to optimize routing. Branching out to new solutions is necessary, as the PND market is currently in decline. However, it is expected to endure for a long time and may never disappear. TomTom aims to slow the decline and lengthen its life by improving the user experience and through innovation. In-dash infotainment systems, on the other hand, will be a growth area. Such solutions are analogous to the PND, but are integrated with the car that is equipped with them. It will take a long time to develop these markets, because cars have long development and replacement cycles compared to consumer electronics. TomTom already currently provides in-dash navigation solutions for several car companies, such as Renault and Mazda.

Since 2004, TomTom has sold over 55 million PNDs and since 2009 over 2 million in dash navigation systems. TomTom maps cover over 100 countries, reaching more than 3 billion people. These figures establish TomTom as a world leading supplier of navigation products and services, and with the new products under development, TomTom aims to maintain this position.

Traditionally, TomTom’s biggest rival is considered the American GPS solutions provider Garmin, mainly because of the competition between the two companies on the PND market. However, the product offerings of the two companies have significant non-overlapping areas as well. For instance TomTom also offers in-dash navigation solutions for the automotive industry. On the other hand, Garmin also has solutions for the aviation industry. Different TomTom products have to compete with products coming from different competitors. Whereas Garmin is the main competitor on the PND marked, TomTom’s RoutePlanner (which can be found at `http://routes.tomtom.com/`) competes with similar services provided by Google Maps and Bing Maps. RoutePlanner has the advantage of benefiting from up to date traffic information, whereas Google Maps has better support for Points of Interest (i.e. finding locations by name, rather than address, where applicable) and includes auto-completion functionality.

Headquartered in Amsterdam, TomTom has over 3,500 employees in offices located in 30 countries. There are four offices in the Netherlands: two in Amsterdam (the headquarters and customer care center), one in Eindhoven (Automotive) and one in Amersfoort (iLocal).

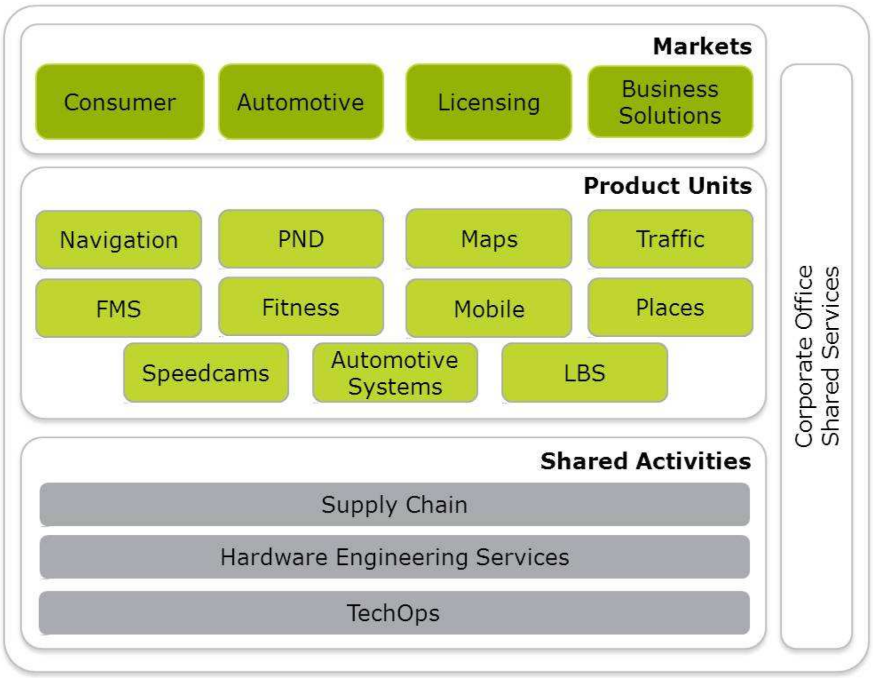


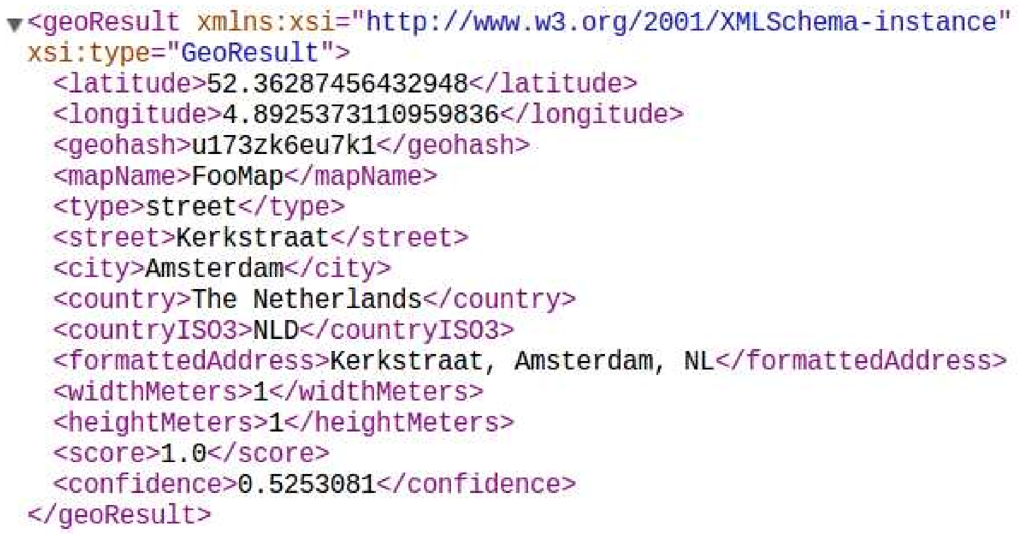
Figure 1.1: TomTom's organizational chart

Across the world TomTom offices are located in Austria, Belgium, Denmark, France, Germany, Hungary, Italy, Poland, Russia, Spain, Sweden, Switzerland, UK, USA, Mexico, Canada, China, India, Indonesia, Japan, Korea, Malaysia, Singapore, South Africa, Taiwan, Thailand, Turkey and Australia.

Figure 1.1 shows TomTom's current organizational chart (as of January 2012), highlighting the company's departments.

TomTom comprises four business units, as indicated by this figure: Consumer, Automotive, Business Solutions and Licensing. These can be seen in the Markets section in the figure, and correspond to the markets currently targeted by TomTom. The products to support TomTom's business units are developed in the corresponding product units shown in the chart. Most names are self-explanatory. The work of the different business units is distributed across multiple TomTom offices worldwide, even at the level of individual teams. Hence, distributing work on a specific project across multiple countries or even continents is very common within TomTom.

The specific product that this thesis is concerned with, named Andorra, is a *geocoding system*, developed under LBS (for Location-Based Services) Product Unit. A geocoding system is a software system used for finding geographic coordinates (latitude and longitude) using other geographic data, most notably text-form addresses consisting of street names, city names, postcodes, or combinations thereof, which are much easier to interpret by humans. Andorra is currently used by Business Solutions to enable fleet management businesses to do geocoding. Andorra will soon also be available on demand to a larger num-



```
▾<geoResult xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:type="GeoResult">
   <latitude>52.36287456432948</latitude>
   <longitude>4.8925373110959836</longitude>
   <geohash>u173zk6eu7k1</geohash>
   <mapName>FooMap</mapName>
   <type>street</type>
   <street>Kerkstraat</street>
   <city>Amsterdam</city>
   <country>The Netherlands</country>
   <countryISO3>NLD</countryISO3>
   <formattedAddress>Kerkstraat, Amsterdam, NL</formattedAddress>
   <widthMeters>1</widthMeters>
   <heightMeters>1</heightMeters>
   <score>1.0</score>
   <confidence>0.5253081</confidence>
 </geoResult>
```

Figure 1.2: XML response given by Andorra for a geocoding query

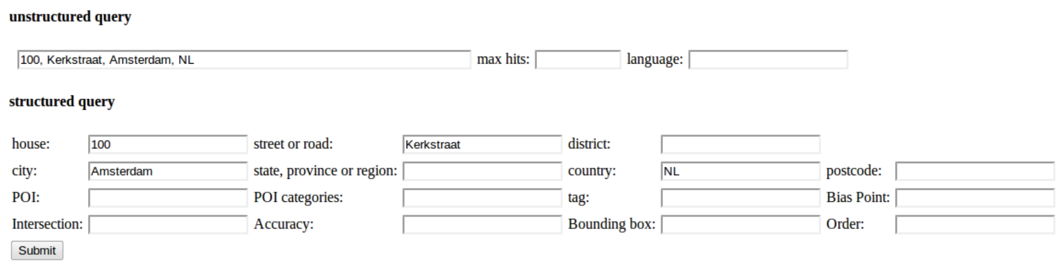
ber of clients, through its inclusion in the Consumer business unit. Geocoding functionality is essential to a large number of TomTom applications, such as the online RoutePlanner or mobile applications that meet different needs. In the following we will give a brief introduction into the main constraints and challenges that need to be tackled in building a geocoding system.

The set of all addresses that can be geocoded to is finite, albeit very large. Andorra, for instance, needs be able to geocode to nearly 1.2 million settlements and over 68 million street-level addresses. Moreover, the addresses to which the system needs to be able to geocode have a very strict and well-defined structure.

The input to a geocoding system will be referred to as a *query*. An example of a typical query that a geocoding system needs to handle is ”Kerkstraat, Amsterdam”. The response for such a query, regardless of the way it is displayed, consists of the latitude and longitude of the location to which the query is geocoded, along with other geographical information that is associated to that location and that the geocoding system supports. Figure 1.2 shows an example output in XML format from TomTom’s Andorra Geocoder for the above mentioned query.

This example shows the typical structure of a returned address for a particular geocoding query. All returned addresses have such a structure, and all the data needs to be derived from a user query containing a combination of city name, street name, postcode or other geographical data. We also note that although the displayed result can take many forms (one option would be to display only the *formattedAddress* field above to the user), the underlying structure of the result always resembles what is depicted in Figure 1.2.

We also note that ambiguity is not a characteristic of the underlying data. All possible latitude-longitude pairs that exist in the geocoder data source uniquely define all locations that can be geocoded to, and the underlying data provides the information associated to



unstructured query

100, Kerkstraat, Amsterdam, NL	max hits:		language:	

structured query

house:	100	street or road:	Kerkstraat	district:			
city:	Amsterdam	state, province or region:		country:	NL	postcode:	
POI:		POI categories:		tag:		Bias Point:	
Intersection:		Accuracy:		Bounding box:		Order:	

Submit

Figure 1.3: The query input interface of the Andorra geocoder

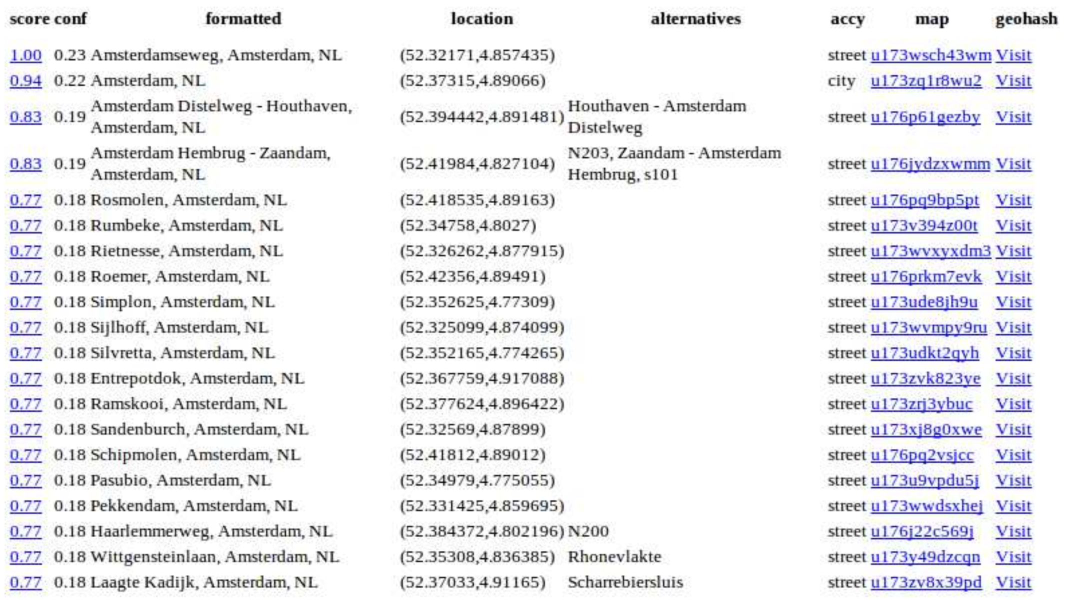
each such location. However, ambiguity is something a geocoding system may have to deal with. The following aspects are among the most important to be considered in order for the geocoder to provide the desired behavior:

- *The use of synonyms*: As far as the geocoding system is concerned, the underlying data defines the truth about the real world. But it is fairly common for the underlying data to contain names such as ”Glenwood Ave”, for example. It is up to the geocoding system to offer its users the flexibility of being able to retrieve such a street with a query for ”Glenwood Avenue”, which is not the correct street name according to the data, but can be assumed to refer to the same thing.

- *Ambiguous queries*: Some queries are more specific than others. For instance, a query simply stating ”Amsterdam” as a city name could be referring to Amsterdam, The Netherlands, as well as Amsterdam in the state of New York, USA. Depending on the available data, it is up to the geocoding system to define a certain preference between the two in the case of this ambiguous query whenever possible and whenever it is considered fit to do so. For example, Amsterdam in The Netherlands should be seen as the more likely desired result, as it is a major European capital. On the other hand, if extra knowledge is available, stating that the user is located in the proximity of Amsterdam, NY, USA, the balance may be tipped the other way.

Furthermore, queries are generally split into two classes: *structured* queries and *unstructured* or *free-text* queries. Figure 1.3 shows part of the user interface used in the development of Andorra - namely, the part that enables input to be provided to the geocoder - with a query for ”100 Kerkstraat, Amsterdam, NL”, in both structured and unstructured form.

Note that the input to the geocoding system need only be one of the two types of queries and structured queries are preferred, as they don’t have the same potential for ambiguity that unstructured queries do. That is because although most address formats are similar regardless of the country, when providing free-text queries users should be free to both change the order of the different elements of the address provided, as well as skip over certain elements in the address. Hence the main difficulty in handling free-text queries: figuring out what address element each input token stands for. This task is further complicated by the fact that many geographical elements (most notably streets) have names consisting of multiple tokens.



score	conf	formatted	location	alternatives	accy	map	geohash
1.00	0.23	Amsterdamseweg, Amsterdam, NL	(52.32171,4.857435)		street	u173wsch43wm	Visit
0.94	0.22	Amsterdam, NL	(52.37315,4.89066)		city	u173zq1r8wu2	Visit
0.83	0.19	Amsterdam Distelweg - Houthaven, Amsterdam, NL	(52.394442,4.891481)	Houthaven - Amsterdam Distelweg	street	u176p61gezby	Visit
0.83	0.19	Amsterdam Hembrug - Zaandam, Amsterdam, NL	(52.41984,4.827104)	N203, Zaandam - Amsterdam Hembrug, s101	street	u176jydzxwmm	Visit
0.77	0.18	Rosmolen, Amsterdam, NL	(52.418535,4.89163)		street	u176pq9bp5pt	Visit
0.77	0.18	Rumbeke, Amsterdam, NL	(52.34758,4.8027)		street	u173v394z00t	Visit
0.77	0.18	Rietnesse, Amsterdam, NL	(52.326262,4.877915)		street	u173wvxyxdm3	Visit
0.77	0.18	Roemer, Amsterdam, NL	(52.42356,4.89491)		street	u176prkm7evk	Visit
0.77	0.18	Simplon, Amsterdam, NL	(52.352625,4.77309)		street	u173ude8jh9u	Visit
0.77	0.18	Sijlhoff, Amsterdam, NL	(52.325099,4.874099)		street	u173wvmpy9ru	Visit
0.77	0.18	Silvretta, Amsterdam, NL	(52.352165,4.774265)		street	u173udkt2qyh	Visit
0.77	0.18	Entrepotdok, Amsterdam, NL	(52.367759,4.917088)		street	u173zvk823ye	Visit
0.77	0.18	Ramskooi, Amsterdam, NL	(52.377624,4.896422)		street	u173zrj3ybuc	Visit
0.77	0.18	Sandenburch, Amsterdam, NL	(52.32569,4.87899)		street	u173xj8g0xwe	Visit
0.77	0.18	Schipmolen, Amsterdam, NL	(52.41812,4.89012)		street	u176pq2vsjcc	Visit
0.77	0.18	Pasubio, Amsterdam, NL	(52.34979,4.775055)		street	u173u9vpdu5j	Visit
0.77	0.18	Pekkendam, Amsterdam, NL	(52.331425,4.859695)		street	u173wwdsxhej	Visit
0.77	0.18	Haarlemmerweg, Amsterdam, NL	(52.384372,4.802196)	N200	street	u176j22c569j	Visit
0.77	0.18	Wittgensteinlaan, Amsterdam, NL	(52.35308,4.836385)	Rhonevlakte	street	u173y49dzcqn	Visit
0.77	0.18	Laagte Kadijk, Amsterdam, NL	(52.37033,4.91165)	Scharrebiersluis	street	u173zv8x39pd	Visit

Figure 1.4: Response from Andorra for an unstructured query for ”Amsterdam”

Andorra does not officially support free-text queries at the time of this writing, but it is planned to offer such support in the near future. A basic form of unstructured query handling is implemented. The functionality currently supported matches an input query against a complete address string in order to find the one that is the closest match to the entered string by looking at tokens common to both. This offers particularly poor performance in the case of ambiguous queries. Consider for example an unstructured query simply stating ’Amsterdam’. The result for such a query is shown in Figure 1.4 as currently provided by Andorra.

As we can see in this figure, although the second suggestion seems like the most likely match for the given query, the first suggestion that is made is a street in Amsterdam, whose name also matches (depending on the matcher used) the name Amsterdam. This may not happen on many examples. However, a high number of irrelevant suggestions are also made: basically all the streets in Amsterdam are returned, the set only being limited in this case by the maximum number of addresses that we chose to retrieve. This can be expected to happen for any query containing a city name and no street name. Ideally, a system for dealing with unstructured queries would transform such inputs into structured queries, by best interpreting what each token stands for. In the example above, the input string should be interpreted as representing a city. This would avoid both problems identified here. As adding proper support for unstructured queries is work in progress, we will assume that such functionality is in place. Moreover, it is required that unstructured queries perform best when the input is nicely formatted, i.e. the query is *well structured*. For our purposes, we will define this to mean one of two possible formats: either *”street name, city name, country name”* or *”city name, country name”*.

1.2 Problem statement

The goal of this thesis is to research effective techniques of building a real-time auto-completion engine for free-text queries for the Andorra geocoder. Auto-completion algorithms aim to offer search query suggestions in real-time, as the user is typing. This functionality is currently lacking from TomTom geocoders, but could be desirable for a number of applications that rely on the geocoding service. The advantages of implementing auto-completion functionality into the geocoding system include:

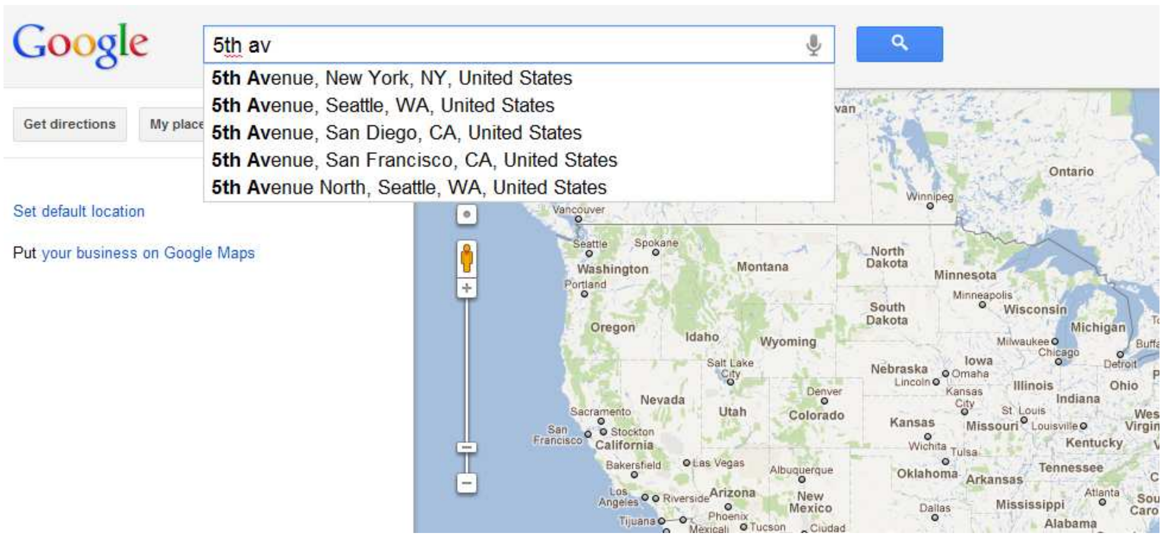
- *better user experience*: As the user is typing, providing useful suggestions is an effective way of helping the user perform their task more efficiently.
- *improved accuracy*: Spelling mistakes are very common due to a series of factors. The situation only gets worse when the user has to formulate a query with strict constraints, quite possibly in a language that the user does not know, as is often the case with geocoding systems. The right suggestion could be very useful for someone typing, as it could save the person the trouble of having to type many related queries before getting the name right, or having to look up the exact spelling of the name somewhere else. For instance, more or less surprisingly, ”Manhattan” is one of the most misspelled location names in America, according to `http://www.epodunk.com/top10/misspelled/`.

The problem of providing auto-completion suggestions has multiple variations. For instance, depending on the type of suggestions that need to be offered, auto-completion may need to suggest common natural language expressions, frequently observed user queries, or suggestions from a fixed set, possibly depending on context. This last type of auto-completion system is common in tasks such as suggesting terms from a dictionary or completion suggestions for source code in IDEs, whereas the former two are commonly used in general search engines, such as those designed to search the entire Web. Our requirements resemble the ones addressed by auto-completion engines that work on a fixed set of suggestions. That is, the fixed set of addresses that we need to be able to geocode to represents the set of all the possible suggestions we need to support.

Another important characteristic of auto-completion systems has to do with whether or not support for suggestion ranking is required. Even when ranking needs to be supported there can be variations in the requirements. While auto-completion strategies for IDEs may only need to specify a ranking between suggestions based on the context and the type of matched token, an auto-completion engine for geocoding systems should be able to deal with arbitrary distributions over supported queries.

Because the geocoder is a tool that sits at the base of a large number of functionalities supported by TomTom products, the auto-completion system has a large number of potential stakeholders who use geocoding and who want to offer the before mentioned benefits to their customers. This includes the online RoutePlanner (`http://routes.tomtom.com/`) and other not yet released products that offer various services to users of mobile applications.

Auto-completion functionality is commonly incorporated in many search systems, in-



5th Avenue, New York, NY, United States
5th Avenue, Seattle, WA, United States
5th Avenue, San Diego, CA, United States
5th Avenue, San Francisco, CA, United States
5th Avenue North, Seattle, WA, United States

Figure 1.5: The auto-completion functionality offered by Google Maps

cluding geocoding systems. Concerning the competing products, auto-completion functionality is offered by Google Maps, but not by Bing Maps, at the time of this writing. The best known example of the use of auto-completion system for geocoding is Google Maps' functionality, illustrated in Figure 1.5.

In the figure we can see that as the user has started typing a query, the software system suggests ways to expand the already typed string to a complete address that the user is likely to be intending to type. The string typed by the user shall be referred to as a *prefix string*. There is a double challenge in building an auto-completion suggestions engine: on one hand, an efficient framework for providing suggestions based on user-entered prefix strings is needed. On the other hand, it is desired that the best use is made of domain-specific knowledge and system usage information, in order to ensure the relevance of the suggestions made.

Therefore, TomTom's main requirements for an auto-completion system, that this thesis should address, are as follows:

- *Main Requirement 1*: The auto-completion system should be able to make query completion suggestions with real-time performance.

- *Main Requirement 2*: The auto-completion system should offer support for arbitrary distributions over the supported query suggestions that it can make.

Note that we have named these requirements (the names being Main Requirement 1 and Main Requirement 2), so that we may refer to them throughout this thesis. Main Requirement 1 may seem ambiguous right now, but we will see in the following chapter how this translates to a technical requirement. For now we only specify that the user needs to have the perception that the system is reacting immediately to an alteration of the provided query

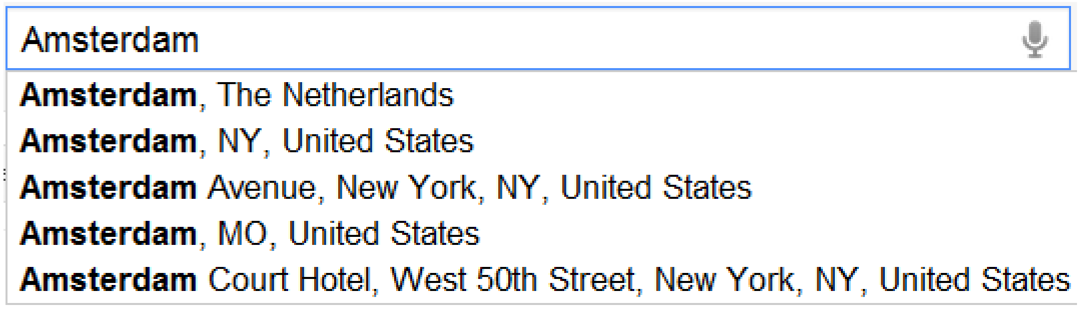

Amsterdam
Amsterdam, The Netherlands
Amsterdam, NY, United States
Amsterdam Avenue, New York, NY, United States
Amsterdam, MO, United States
Amsterdam Court Hotel, West 50th Street, New York, NY, United States

Figure 1.6: Query completion suggestions for the prefix ”Amsterdam” as offered by Google Maps

prefix. Main Requirement 2 simply states that the system should accept input consisting of the supported suggestions, with arbitrary weights associated to them, representing their relative priorities, and then make suggestions according to these priorities.

Thus, the problem statement consists of researching techniques that can be used to add such functionality to TomTom's Andorra geocoder. Hence, the geocoder should offer clients the possibility of efficiently retrieving a list of suggestions for a provided prefix, ordered according to relevance.

Because we are dealing with the problem of auto-completion in the context of a system that retrieves addresses, we also require that suggestions be presented in a format that includes the complete hierarchy of geographical data describing an address, as we can see in Figure 1.6. The reason for this is that the completion suggestion making process is -in general, at least- completely separate from the actual geocoding. Thus, as explained earlier, it is desirable to provide the free-text query processor with an input that it can best interpret. We will aim for a general enough implementation so that it shouldn't matter how we define this property, but to be going on with, we will consider that only well structured suggestions, as defined previously, should be made. This cannot be seen as an added requirement to the auto-completion system in itself, however, since the data provided to the auto-completion system as supported suggestions should simply represent such address strings, and it is the job of the system making use of the auto-completion system to provide this data.

Figure 1.6 also shows an example of meeting Main requirement 2. Source data, as well as user logs could be used to assign priorities to suggestions, but regardless, it is useful to derive information that allows the system to decide to promote, for example, a street in New York (”Amsterdam Avenue”) as a more likely candidate than a town named ”Amsterdam” in the state of Missouri for a given prefix.

In addition to these requirements that are inherent to an auto-completion system as described above, next we identify some extra requirements that are specific to offering auto-completion functionality for geocoding systems. These extra requirements were derived from observing such functionality as it is offered by competing products and following discussions with potential users of the geocoding service from other teams inside TomTom.

One such feature is illustrated in Figure 1.7: the system should also offer suggestions

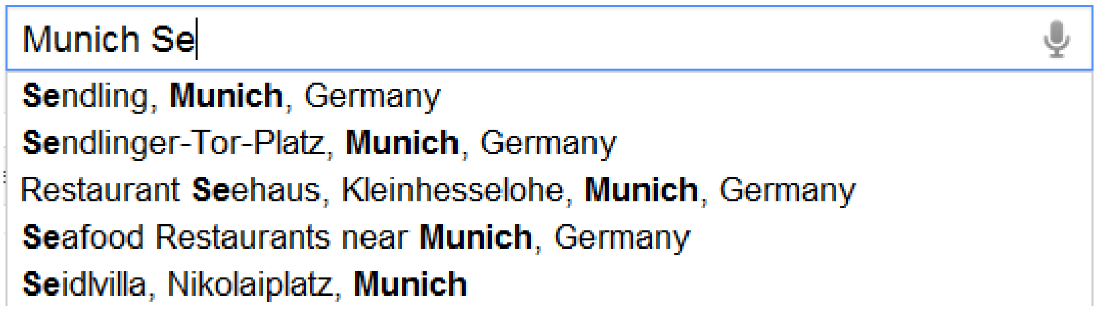

Munich Se
Sendling, Munich, Germany
Sendlinger-Tor-Platz, Munich, Germany
Restaurant Seehaus, Kleinhesselohe, Munich, Germany
Seafood Restaurants near Munich, Germany
Seidlvilla, Nikolaiplatz, Munich

Figure 1.7: Well-structured query completion suggestions for prefixes not obeying the well structured address format

of which the partial query typed is a substring which is not a prefix of the well structured query. This is particularly interesting for geocoding systems. One important argument for the necessity of supporting this feature is that we are dealing with a setting in which it is quite likely that the user is very unfamiliar with the query that they are trying to formulate. Thus, when searching for ”Sendlinger-Tor-Platz, Munich, Germany”, a user who is unfamiliar with the spelling could first type the city name in order to restrict the search scope. It is, therefore, desirable in this case that a query prefix of ”Munich, Se”, for example, should still produce useful suggestions in the well structured format described earlier. This can be seen to be the case in the example of Figure 1.7. Thus, we can formulate the following optional requirement:

- *Optional Requirement 1*: The auto-completion system should offer suggestions that are well structured, even if the supplied prefix does not obey this structure.

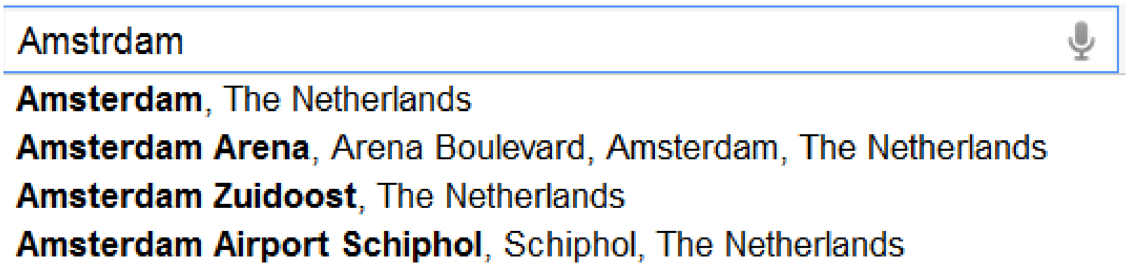
Another desirable property is the ability of the system to offer suggestions for prefixes that don’t perfectly match prefixes in the set of supported queries. There are two cases to be considered here:

- On one hand, we could simply be dealing with a typing error by the user. Such a scenario is depicted in Figure 1.8. Here, the mistyped name ”Amstrdam” still gets the useful suggestions for queries containing the name ”Amsterdam”. Such functionality is also desirable for the auto-completion system that we are developing for the Andorra geocoder.

- Another type of inexact matching is required in the case of special characters or groups of characters. This is language-specific behavior. An example of such matching is presented in Figure 1.9. The German language character ’ß’ should be accepted both as-is, as well as substituted by the character group ’ss’.

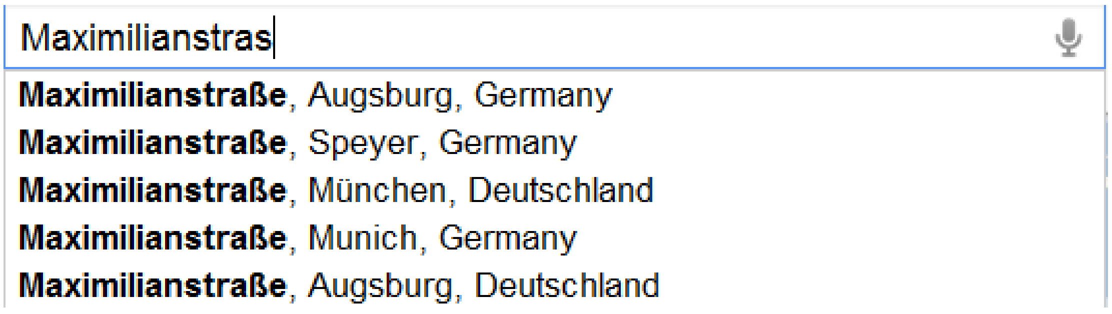
Thus, we can formulate the following optional requirement:

- *Optional Requirement 2*: The auto-completion system should offer suggestions for mistyped prefixes that do not match any suggestion prefix.


Amstrdam

Amsterdam, The Netherlands
Amsterdam Arena, Arena Boulevard, Amsterdam, The Netherlands
Amsterdam Zuidoost, The Netherlands
Amsterdam Airport Schiphol, Schiphol, The Netherlands

Figure 1.8: Query completion suggestions for the mistyped prefix ”Amstrdam”


Maximilianstras

Maximilianstraße, Augsburg, Germany
Maximilianstraße, Speyer, Germany
Maximilianstraße, München, Deutschland
Maximilianstraße, Munich, Germany
Maximilianstraße, Augsburg, Deutschland

Figure 1.9: Query completion suggestions for alternative spelling of the same name

It is also desirable to implement support for a *location bias* in the suggestion making process. That is, the priorities can be altered depending on the region of the map that the user is currently looking at, which implies the need to quickly alter the distribution of matched queries. For the example in Figure 1.9, if the user were zooming into the region of the map where the city Munich is located, or if we had knowledge that the user is located somewhere in Munich as they are typing a query, we would prefer to use this knowledge to promote the street in Munich as the first suggestion.

Thus we derive the following optional requirement:

- *Optional Requirement 3*: The auto-completion system should allow for making suggestions taking into account both the suggestion weights, as well as the user location, if available.

The main challenge in meeting this requirement is incorporating location-aware logic into the suggestion making process, preferably without having to dramatically alter the approach to making suggestions in a location-unaware context, and preferably at no high computational cost compared to the standard suggestion-making process.

Remark 1.1. Despite the name, the optional requirements mentioned here are highly desirable. The auto-completion system that we are describing in this thesis should be able to meet as many of these requirements as possible. ■

Thus, the result of this thesis project should consist of a description of algorithmic techniques that can be used to meet the main requirements identified in this chapter, and which ideally offer extensions to meet all the optional requirements as well. Also, a proof of concept implementation should be made, in order to demonstrate the described algorithms.

1.3 Thesis outline

In the following chapter we will translate the business requirements identified here into a research question, and we will look at how this problem is approached in existing literature. We will also look at the available software packages that could address our requirements and see to which extent they meet our demands. Based on this, we will outline the overall approach that we will take, motivating our choices using existing conclusions in literature and knowledge of the particularities of addressing auto-completion in the context of geocoding. We will then introduce some definitions and properties of the theoretical model that we will use as support for our implementation in order to identify its capabilities and its applicability to our problem.

The third chapter will describe the actual algorithms that we will use to address all the requirements identified earlier. We will start with a description of the algorithms for meeting the main requirements for an auto-completion system, and then identify ways to extend these algorithms to accommodate all of the remaining (optional) requirements. The main challenge comes from making sure that the added functionality is done in a way that can be closely integrated with the overall approach and without a high performance penalty.

Then, in Chapter 4 we will explain the experimental setup used to test whether, to which extent and with what implications these requirements can be met by the proposed algorithms. We will devise experiments in order to verify that the assumptions we made in developing our algorithms were true, and we will place particular focus on the impact that each added functionality has on the performance of the suggestion making process. We will also test a number of hypotheses about the impact on performance that different choices in the usage of the system may have.

The main conclusions will be drawn in Chapter 5. We will also take the opportunity there to outline the main ideas for future work, based on the experimental results from Chapter 4.

The architecture of the actual implementation will be discussed in Appendix A. The various components will be described along with their roles and how they relate to the algorithmic presentation in this thesis.

Chapter 2

Methods for auto-completion

This chapter will discuss related work done to address the business requirement introduced in Chapter 1 and formulate the technical requirements for the system that we are developing. We will first identify geocoding systems as a particular class of information retrieval systems and look at what it means to adapt techniques for dealing with auto-completion in information retrieval systems to the specific context of geocoding.

We will then analyze the handling of the auto-completion problem in literature. We will also look at existing software packages and derive the overall approach that we will take based on the conclusions that can be drawn from related work. With these conclusions, as well as the requirements identified in the previous chapter in mind, we will introduce the theoretical model that we will use to solve our problem: the *Probabilistic Prefix Tree (PPT)*. We will look at the literature dealing with PPTs, in order to identify their properties and applicability to our problem. Subsequent chapters will deal with the actual implementation and analysis of our algorithms based on PPTs.

2.1 Information retrieval systems and geocoding systems

We begin with a discussion of the broader field of information retrieval, because, as we will see, geocoding systems are a particular type of information retrieval systems, and most of the literature dealing with the problem of auto-completion is focused on this more general context. As a particular type of information retrieval system, a geocoder can make use of similar approaches, but also introduces some extra constraints to be considered.

Information retrieval is the task of searching within a given set of documents, for exactly those that satisfy a certain information need [4]. The search is done based on a query formulated by the user of the information retrieval system such that the underlying information need is best represented. An information retrieval system then has the task of retrieving the documents that are relevant for the (most likely) underlying information need, based only on the given query.

Information retrieval systems are very widespread and intensely used nowadays in order to perform numerous common search tasks, ranging from searching through personal e-mails, searching for news articles on a news website to searching the entire Web for a particular piece of information. The best known example of an information retrieval system is Google's search engine, which handles over one billion search requests every day.

Information retrieval systems are often used for search over unstructured documents (usually containing plain text). An example of information retrieval through structured data is when querying a relational database. Searching through a set of documents having a certain structure (i.e. certain fields), but without many constraints on the field contents is called semistructured information retrieval [4].

We stated that the task of an information retrieval system is to retrieve documents that are relevant to a particular information need. However, the system does not have access to the information need, but rather to a textual representation of it (the query) provided by the user. The task of searching is thus by definition not only given in terms of the query supplied as input, but also in terms of the relation between the query and the likely underlying information need.

In order to better express this relationship, domain specific knowledge should be used. For example, in an information retrieval system where the documents are known to represent scientific papers, we may want to use the knowledge about the structure of a typical paper in order to give more importance to finding a certain piece of information in the abstract, compared to the other sections, for example. If this is known to be in accordance with what users expect, then we have used the extra knowledge about the documents in order to improve the accuracy of the information retrieval system. Similarly, a geocoding system expects that a query represents an address, and thus tries to map the tokens of a given query to fields of an address, such as street name, city name, etc. Using this extra knowledge correctly is very important in getting significantly improved performance from the geocoding system, as explained in the previous chapter, when we discussed the free-text query support for Andorra.

Another approach commonly used in practice to improve the accuracy of such systems is *machine learning*. Machine learning can be seen as the process that enables us to solve a particular problem when either specifying the problem statement in a rigorous way is not possible, or finding an algorithm to directly address the problem proves to be a very complex, if at all possible, task [2]. Some examples of tasks that fall under this category are spam filtering [1], face recognition [11], making personalized recommendations of news articles [15] and many more. The way to tackle these problems is by building a simpler model that enables us to solve the problem in a satisfactory way, for example a parametrized function computing the probability that a certain user is interested in a certain news article. The model is configured - that is, its parameters are adjusted - through the use of data. This is the actual ”learning” task. This approach relies on the belief that there is a process that explains the data we observe, but we do not know the details of it. The idea is to use a machine in order to ”learn” to approximate that process.

User feedback is commonly used with machine learning algorithms, not only to assess the performance of an information retrieval system, but also (as a natural extension) to improve or personalize the system behavior, either in general or with respect to specific users [7]. Measures commonly used for user feedback include the amount of time the user spends on a page, clickthrough data ([15, 7]), and even subjects' eye movements [8].

Many of the approaches used in information retrieval systems, including those addressing the auto-completion problem can be applicable to geocoding systems as well. For instance, user feedback can be useful in a geocoding system because it could allow the system to infer preference relations among different locations to which it can do geocoding, depending on the query and the knowledge of past usage of the system. Thus, an auto-completion system for geocoding can make use of implicit feedback to ”learn” to make better suggestions. To support this it is important that we construct a solution that allows for a great degree of flexibility when configuring the ranking among suggestions made, especially since user preferences are known to change over time.

On the other hand, as a particular class of information retrieval systems, geocoders have properties that set them apart in this class. Knowledge of these particularities is important when adapting search or suggestion making techniques used in other search applications. Such knowledge can help us make simplifying assumptions, but also introduces extra constraints and requirements. For instance, a geocoding system does not have to account for the infinite productivity of natural languages because the addresses that are supported form a fixed set, leaving no room for generalization. On the other hand, as we specified in the first chapter, geocoding systems are by definition concerned with location, and as such they could benefit greatly from knowing where a certain request is coming from. For instance a user typing an address on a mobile device is very likely to be looking for locations in their immediate vicinity. Thus, making good use of the knowledge of the user’s location is crucial in order to produce relevant suggestions, at least in some applications.

But before addressing the requirements specific to geocoding systems, we will first look in the next section at how the auto-completion problem is addressed in literature, in the related, but more general context of information retrieval systems. We will then look at available software packages that implement such algorithms. We will end the section by identifying the shortcomings of these approaches and motivating the introduction of a new approach, based on the model of a Probabilistic Prefix Tree.

2.2 Literature and software libraries for auto-completion

As mentioned in the previous section, the auto-completion problem is studied in literature mainly in the general context of information retrieval systems. We will survey these approaches in this section and identify their applicability to our problem, and look at the possible extensions that need to be made to accommodate the extra requirements that come with supporting auto-completion for geocoding systems.

The requirements are that such algorithms are not only accurate, but also efficient. In order to get the perception of real-time interaction, a response time of no more than 100 ms is required [12].

Algorithms for providing query completion suggestions take as input a string which is assumed to represent the prefix of some query that the user is formulating and output a list of possible queries that could be derived from that prefix and that the user is most likely to have started typing as they were entering the prefix. This problem of offering suggestions from a fixed set of possibilities has been studied in [22, 10]. We will look at those approaches in this section. Other variations of the the problem of auto-completion have also been studied in literature. For instance [9] looks at the problem of making completion suggestions by taking into account the context in which the to-be-completed word has been typed. Such considerations do not apply to our problem, however.

A problem similar to ours - that of displaying strings from a fixed set - is studied in [22]. The focus of the paper is on incorporating error-tolerant matching of user prefix queries. Thus, this paper addresses Optional Requirement 2 (see previous chapter) in addition to our main requirements. As a distance measure, the well known Levenshtein distance is used [5]. The Levenshtein distance between two strings is defined as the minimum number of edits needed to transform one string into the other. Three different edits are considered: deletion of a character, replacement of a character by another or inserting a new character at some location in the string.

Example 2.1. The Levenshtein distance between the strings 'oslo' and 'snow' is 3. That is, we can obtain the string 'snow' from 'oslo' through the following sequence of transformations: **o**slo $\xrightarrow{delete}$ slo ; s**l**o $\xrightarrow{replace}$ s**n**o and sno $\xrightarrow{insert}$ sno**w**. ■

The well known dynamic programming algorithm for computing the Levenshtein distance is based on the following recursive formula for computing the distance between the prefix of the first string ending at index i and the prefix of the second string ending at index j, where δ_{ij} is 0 if the characters at index i and j in the two strings respectively are the same and 1 otherwise:

$$D[i,j] = min(D[i-1,j]+1, D[i,j-1]+1, D[i-1,j-1]+\delta_{ij})$$

One approach to providing auto-completion is to use an n-gram based algorithm. An n-gram is a contiguous sequence of characters of length n inside a string.

Example 2.2. Consider the string "abcd". The 1-grams, also called *unigrams*, of this string are "a", "b", "c", "d". The 2-grams, also called *bigrams*, are "ab", "bc", "cd". And generally, a string of length m has $m-n+1$ n-grams if $m \geq n$. ■

The intuition is that two strings that are very similar should have a high overlap in their n-gram sets. To make this more precise, the edit distance between two strings s and r is at most k if the intersection between their n-gram sets is at least $(max(|r|,|s|)-n+1)-n\cdot k$ [22]. This property can be used to enable error-tolerant string matching, based on set-similarity. These algorithms are considered the state of the art in offline edit distance matching.

For the task of online query completion, an alternative approach using a *trie*-based algorithm is proposed in [22], and shown to outperform the n-gram algorithm. A trie, or *prefix-tree*, is a data structure that supports fast search for a string withing a given set. Tries can also be used to match keys in associative data structures where the keys are strings.

Example 2.3. An example trie is shown in Figure 2.1. Each of the terminal nodes along each path represent one supported string, in this case those belonging to the set {tex, ted, tod, ad} ■

Example 2.3 shows the main idea behind the use of tries: reduce the storage space requirement and parsing time for all strings by sharing prefixes. While the space reduction

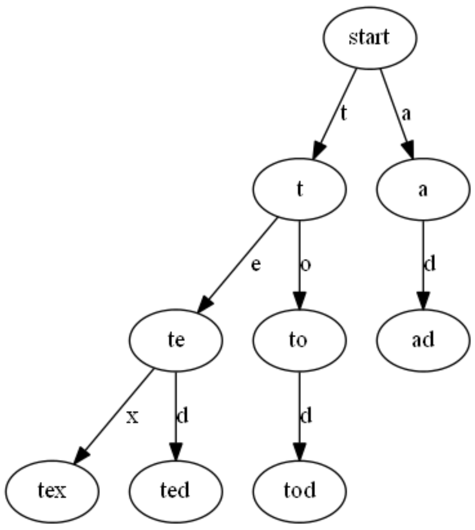


Figure 2.1: Trie

is dependent on the degree to which the supported strings share prefixes and on the representation of the links between nodes, the main advantage is that it now only takes $O(m)$ time to determine whether or not a string belongs to the set, where m is the length of the string we are searching for, as each character needs to have a corresponding transition from the current state. The same complexity is obtained for adding a string to a trie, through a process that is very similar to parsing.

Example 2.4. Given the trie in Figure 2.1, parsing the string 'tex' can be done by taking the the edge labeled 't' from the start node, then the edge labeled 'e', and finally the edge labeled 'x' which is the last character in the searched string and leads to a leaf. Hence the string is successfully parsed. On the other hand, parsing the string 'tall' would fail as soon as we find no edge labeled 'a' after parsing 't'. Adding a new string to the trie is done through a similar process, with the only difference that whenever edges with the required label are not found, they are simply added. ■

As example 2.4 shows, whenever parsing a string using a trie, or adding a string to it, the maximum number of operations is bounded by the length of the string to be parsed or added. Note that the main advantage of this is that this run-time does not depend in any way on the total number of strings that the trie is able to parse.

The $O(m)$ parsing time is implementation specific, however. To achieve it, it is assumed that the next character can be looked up in constant time. This can be achieved by using an array to associate a new edge to characters where necessary. This approach can be too inefficient in terms of space requirements if the supported alphabet is very large, especially since the number of transitions emitting from each node should be expected to drop as we go deeper down the tree.

The way error-tolerance is implemented to work with this approach is by adapting the well-known dynamic programming algorithm for computing the Levenshtein distance mentioned earlier, introducing the notion of an *active node*, representing a node that can be expanded to generate suggestions, based on the given prefix and some threshold for the maximum tolerated edit distance.

The algorithm to achieve this is described in [22] and we also present its pseudocode here, as Algorithm 1. The algorithm maintains a set of active nodes, *valid*, initialized to contain just the start node and all nodes reachable from it within the maximum provided edit distance, *maxDistance*. Then, iteratively, as each character is consumed, the set of active nodes is replaced by a new set, starting from the current active nodes. If an active node was reachable within edit distance less than the maximum threshold, then it is maintained in the set with an increased distance (this corresponds to removing the currently read character). The node that follows down the path of the consumed character is added and the distance is not increased (this corresponds to leaving that character as it is). Finally all other nodes that follow edges otherwise labeled are added to the set, with increased distance (this corresponds to replacing the currently parsed character with another character). Then the set thus formed becomes the new set of active nodes, and the process is repeated with the next character in the input $prefix$. Note that if the distance has already reached the maximum allowed threshold, only the second of the operations mentioned above can be performed (the one that does not require further increasing the distance). Anything that falls outside the cases mentioned here will not result in active nodes being added to the set. After parsing the whole string, we can expand all active nodes, as they all correspond to adequate suggestions for the typed prefix, given the error-tolerance limit being used.

Example 2.5. Figure 2.2 shows how the set of active nodes evolves while parsing the string 'tax' with the trie in Figure 2.1 and a maximum edit distance threshold of 1. The highlighted nodes are active, and next to each active node we write the minimum distance of reaching that node, while consuming the prefix up to a certain point. ■

Referring back to the requirements that we introduced in Chapter 1, the previously mentioned approaches do not meet several of them. Although the performance of the trie-based approach is shown to meet real-time requirements, thus meeting Main Requirement 1, setting priorities among the supported strings is not possible in this implementation. Instead, the authors suggest making use of a static score for the suggestions made. In order to avoid sorting a large number of possible suggestions according to this score, the authors suggest pre-computing the top-k suggestions associated to each node in the trie. This is a possible solution to meeting Main Requirement 2, but this approach is not very flexible. We would prefer to be able to alter suggestion scores at run-time without having to recompute the top-k suggestions associated to each node. We also expect that such lists would not be very useful if we want to alter suggestion rankings based on the user location (to meet Optional Requirement 3), since in that case the lists are not fixed between different queries. Also, suggestions are only made for matching prefixes, which does not meet Optional Requirement 1. Another shortcoming is that there is no support for a location bias in the suggestion making process, since location is not an important part of the system under consideration in the paper.

Algorithm 1: GetValidStates(prefix, maxDistance)

```
input  : The query prefix, prefix, and the maximum admissible edit distance,
         maxDistance
output: All active states, valid corresponding to prefix
valid ⟵ [];
valid.add(start,0);
foreach node reachable from start within distance $i \leq$ maxDistance do
    valid.add(node,i);
foreach character c in the prefix, in order do
    new_valid ⟵ [];
    foreach node node in valid, in queue order do
        if valid[node].distance + 1 $\leq$ maxDistance then
            new_valid.add(node,valid[node].distance + 1);
        if node has child node' through c then
            new_valid.add(node',valid[node].distance);
        l ⟵ max(new_valid[node].distance,valid[node].distance);
        if l + 1 $\leq$ maxDistance then
            foreach child node' of node do
                new_valid.add(node',l+1);
    valid ⟵ new_valid;
return valid;
```

Another trie-based approach to auto-completion is presented in [10]. The focus here as well is on achieving error-tolerant auto-completion, but unlike the approach presented earlier, here a database of corrected queries is used to learn to make corrections by training a transformation model. A transformation model is defined by decomposing a transformation from the intended (corrected) query c to the mistyped query q into substring transformation units.

Example 2.6. The paper gives as an example the transformation $britney \rightarrow britny$ into the substring transformation (or *transfeme*) units $\{br \rightarrow br, i \rightarrow i, t \rightarrow i, t \rightarrow t, ney \rightarrow ny\}$ ■

The advantage of this approach is that the transformation model learns to make such corrections from data, so with sufficient data, error-correction should be expected to perform better, as it is more likely to prioritize correcting mistakes that are more common. This transformation model is used along with a trie that represents the supported queries which can be suggested by the system. Probabilities are also assigned to the edges in the trie representing the likelihoods of following those edges from the current node. Because we are lacking a database of corrected queries used against Andorra, we cannot use a similar approach to error-correction. Instead, we will prefer adapting the edit distance approach discussed in [22] to our algorithms. We will, however, also be looking to implement a probabilistic approach in order to support suggestion ranking, but we have to come up with an approach that can be extended to also meet Optional Requirement 1 and Optional Require-

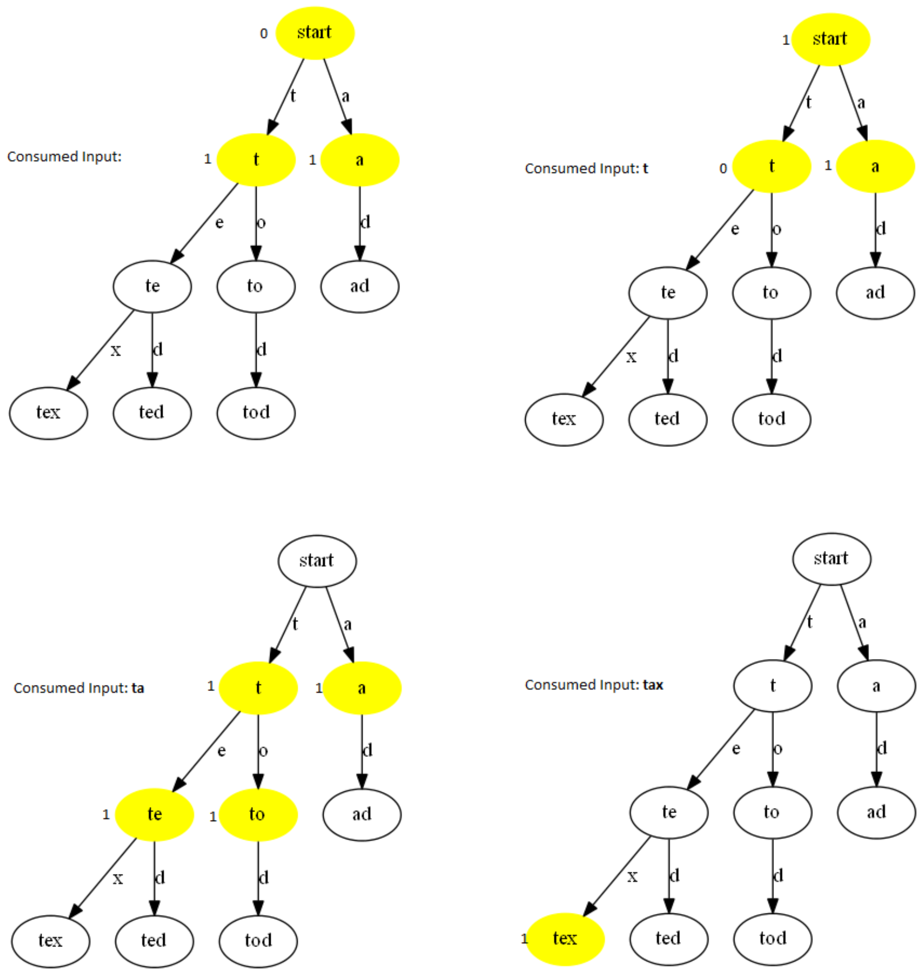


Figure 2.2: Edit-distance based error-tolerant parsing

ment 3, which are not addressed in [10].

Next we will introduce some software libraries and tools that can be used for providing auto-completion functionality. The algorithms implemented by these tools are similar to the ones discussed in literature, with some differences and enhancements as we will see. We will evaluate their suitability in addressing our problem and we will end by motivating the choice to develop a new solution from the ground up.

- *Sphinx* (`http://www.sphinxsearch.com/`) is a full-text search engine, used for searching words in indexed documents, similar to *Apache Lucene* and *Solr*. Hence, it is essentially an information retrieval system that works with SQL databases, NoSQL storage and simple files. Such technologies can be used for example in implementing *n*-gram-based solutions like the ones described earlier, by indexing documents consisting of the *n*-grams of the supported queries. Such an approach does present some shortcomings, though. As stated previously, these approaches are the state of the art in offline edit distance matching, but are shown to be outperformed by trie-based edit distance matching solutions for the purpose of online auto-completion. Alternatively, an index based on supported prefixes can be created, but this would result in a potentially very large index, and despite the moderate memory requirements, querying an index may prove too slow for auto-completion if the index size is too big. Also, such an approach does not offer the level of flexibility we would like in order to easily adjust the distribution of queries. Also, there is no included support for implementing a location bias, and implementing this functionality as a post-search technique could prove too computationally expensive for a service that aims to deliver real-time performance.

- *LingPipe* (`http://alias-i.com/lingpipe/`) is an open source suite of Java libraries for the analysis of human language. It also offers auto-completion functionality using a trie implementation over a set of strings, which seems to be the recommended approach in the present literature as well. The library implementation's functionality is similar to the trie-based algorithm discussed earlier, but also supports ranking among the suggestions that can be made, as it takes as input a set of strings along with weights associated to each of them, and produces completions for given prefixes, ranked according to these weights, thus meeting both of our main requirements. Edit distance based error-tolerance is also implemented. However, the implementation lacks the flexibility to configure probabilities for the supported queries in real-time, as it uses a scoring scheme based on string similarity and the pre-computed counts, or weights. Another reason for not being able to use this solution as-is is the lack of support for matching substrings with arbitrary start index within the supported query pattern (i.e. this approach does not meet Optional Requirement 1). Thus, well structured queries, as they were defined in the previous chapter, can only be provided if the user offers a prefix of that well-structured query, and not for a prefix not starting with the smallest element in the address. Also, the implementation is not location-aware, so Optional Requirement 3 is also not met.

These shortcomings, along with the unexploited potential of using properties particular to geocoding systems lead us to favor developing an in-house implementation over the general solutions presented so far. In order to implement auto-completion for Andorra, we will make use of the conclusions from the mentioned literature and opt for a prefix-tree based approach. However, we need to adapt this approach in order to meet the requirements identified in Chapter 1. Given the conclusions in the existing literature, we expect that a prefix-tree is a good choice for real-time suggestion making. However, we also need to account for the ranking of suggestions made in a highly configurable way. In addition to these main requirements, we should aim for a solution that allows us to easily integrate the other optional requirements: substring matching, error-tolerance and location bias.

The need to support arbitrary distributions over the suggestions that can be made leads to the idea of using a probabilistic approach. We would like to combine this with the benefits in terms of speed offered by prefix trees. A data structure that meets these requirements is the Probabilistic Prefix Tree, which is a particular class of Probabilistic Automata. We will use the Probabilistic Prefix Tree as the core model for developing our algorithms. Several modifications will be made to this model in order to meet all our requirements. We will describe all these modifications in the next chapter, as we present our algorithms, but first, in the following section, we will present a survey of the current literature on these models in order to get a better understanding of their known properties and thus to establish their applicability in solving our problem. The algorithms for implementing the desired functionality will be described and analyzed in the next chapter, with an experimental evaluation to follow in Chapter 4.

2.3 Probabilistic Finite Automata and their properties

The goal of this section is to introduce the theoretical concepts necessary in analyzing the algorithms that we will develop to address the problem that this thesis is concerned with. As prefix-trees form the basis of our approach, it is important to study the properties of these structures that have been identified in previous work. Our implementation will be based on a *Probabilistic Finite Automaton (PFA)*. We will look at the definition and properties of such structures. Automata are extensively used to model language, but they have also been used successfully to model physical systems [14], in tasks ranging from increasing sustainability of existing systems by learning efficient behavior [23] to giving systems the capability to self-diagnose [3].

We will now define Probabilistic Automata (PA). Definitions of these structures can also be found in [18] and [17], along with more detailed descriptions and comparisons to other similar concepts. We shall present the general definitions here and quickly turn our attention to the class of PAs of particular interest to us.

In order to study PAs we must first introduce the notion of *stochastic language* [18]. Let Σ be a finite alphabet, and $\Sigma^\star$ the set of all strings over Σ, including the empty string λ, and let Σ^n (and $\Sigma^{\leq n}$) represent the set of words of length n (and the set of words of length no greater than n respectively). A *language* is defined as a subset of $\Sigma^\star$.

Example 2.7. Let $\Sigma = \{a, b, c\}$. Then $\Sigma^\star$ is the set $\{\lambda, a, b, c, aa, ab, ac, ba, ...\}$. Any subset of $\Sigma^\star$ defines a language. An example of a language is $\Sigma^{\leq 2} = \{\lambda, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc\}$. ■

Definition 2.1. A *stochastic language* ψ is a probability distribution over $\Sigma^\star$.

Alternatively, we can define a *semi-distribution* ([17]) as follows:

Definition 2.2. A *semi-distribution* over $\Sigma^\star$ is a function $\psi : \Sigma^\star \to [0,1]$ satisfying $\sum_{u\in\Sigma^\star} \psi(u) \leq 1$.

Thus, we can give the following alternative definition for a stochastic language:

Definition 2.3. A *stochastic language* ψ *over* $\Sigma^\star$ is a semi-distribution over $\Sigma^\star$ such that $\Sigma_{u\in\Sigma^\star}\psi(u) = 1$.

Example 2.8. Given $\Sigma = \{a,b,c\}$, the function $\psi : \Sigma^\star \to [0,1]$ defined by $\psi(\lambda) = \psi(a) = \psi(b) = \psi(c) = 0.25$ and $\psi(u) = 0$, $\forall u \in \Sigma^\star \setminus \{\lambda,a,b,c\}$ is a stochastic language over $\Sigma^\star$. ■

We will denote $Pr_\psi(x)$ the probability associated to a string $x \in \Sigma^\star$ under the distribution ψ. Note that according to the previous definition, the distribution must verify $\Sigma_{x\in\Sigma^\star} Pr_\psi(x) = 1$. Assuming that the distribution is modeled by a machine A, the probability of x according to the probability distribution defined by A is denoted $Pr_A(x)$, and the distribution modeled by A will be denoted ψ_A. If L is a language over Σ, and ψ is a distribution over $\Sigma^\star$, $Pr_\psi(L) = \Sigma_{x\in L} Pr_\psi(x)$.

In the context of geocoding, the finite alphabet Σ is the set of all characters contained in the address strings corresponding to locations that can be geocoded to, and the language is the subset of $\Sigma^\star$ consisting of valid strings that represent addresses. An easy way to conform to the definition of stochastic languages would be to assign probability 0 (through the function ψ) to all strings except the ones representing addresses, and assign equal probabilities to all address strings. Thus, if there are n address strings in total, probability $\frac{1}{n}$ can be assigned to each of them. Alternatively, though, we would like to assign probabilities depending on the relevance of each string: assigning higher probabilities to large cities, compared to small roads in villages, for instance.

Definition 2.4. A *sample* S is a multiset of strings from $\Sigma^\star$.

We will denote the *size* of the sample by $|S|$, and the number of distinct strings in S by $\|S\|$. Thus, the empirical distribution associated with S will be denoted by ψ_S, i.e. $Pr_{\psi_S}(x) = \frac{|\{x\}\cap S|}{|S|}$. We will use such samples to generate the distributions of address strings supported by the auto-completion system.

Next we will first define a semi-PA([17]). We will then be able to define a PA in terms semi-PAs by imposing a simple condition.

Definition 2.5. A *semi-PA* is a 5-tuple $A =< \Sigma, Q, \delta, \gamma, \tau >$, where:

- Σ is the finite alphabet
- Q is a finite set of states
- $\delta : Q \times \Sigma \times Q \to [0,1]$ defines transition probabilities
- $\gamma : Q \to [0,1]$ defines the probability of a state being an initial state
- $\tau : Q \to [0,1]$ defines the probability of a state being a final state

and the following constraints are satisfied:

- $\sum_{q\in Q}\gamma(q) = 1$

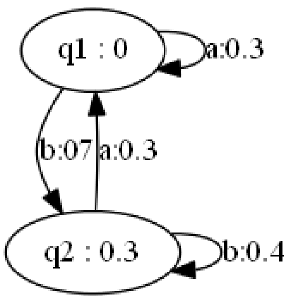


Figure 2.3: Semi-PA with multiple initial states

- $\forall q \in Q$, we have $\tau(q) + \sum_{a\in\Sigma}\sum_{q'\in Q}\delta(q,a,q') = 1$

The function δ can be interpreted as assigning a probability to each transition from one state to another. The first constraint in the definition above states that the total initial state probability across the states of the automaton should be 1. Hence there is no notion of the automaton not generating any string in $\Sigma^\star$. It may, however, generate the empty string with a certain probability in case we have states q with both $\gamma(q) > 0$ and $\tau(q) > 0$. The second constraint in the definition is similar to the first, stating that the probabilities of a state being final or leading to any transition add up to 1, and hence any other option is excluded. These two constraints ensure that a semi-PA defines a semi-distribution over $\Sigma^\star$.

Example 2.9. Figure 2.3 shows an example of a semi-PA. Here, $Q = \{q1, q2\}$, $\Sigma = \{a, b\}$, and we assume $\gamma(q1) = 0.6$ and $\gamma(q2) = 0.4$. We also assume $\tau(q1) = 0$, $\tau(q2) = 0.3$ (as indicated on the states themselves). The δ probability values are written on the corresponding edges, along with the edge labels. It is easy to see that the properties of semi-PAs are verified in this example.■

Definition 2.6. A state q is said to be an *initial state* if $\gamma(q) > 0$ and it is said to be a final state if $\tau(q) > 0$.

Proposition 2.1. *Any semi-PA is equivalent to a semi-PA with one initial state.*

Proof. A constructive proof of this proposition is presented in [17]. □

Example 2.10. Returning to the example in Figure 2.4, we note that it contained two initial states: $q1$ with $\gamma(q1) = 0.6$ and $q2$ with $\gamma(q2) = 0.4$. The idea behind converting this semi-PA to one with one initial state is to add a state $q0$ with this designated purpose of being the single initial state in the semi-PA. The different fields are updated accordingly, such that a semi-PA generating the same semi-distribution over $\Sigma^\star$ is obtained. These changes are shown in figure 2.5.

In the new semi-PA there is only one initial state $q0$ having $\gamma'(q0) = 1$ and $\gamma'(q1)$ and $\gamma'(q2)$ are both set to 0. Nothing else changes for $q1$ and $q2$. The probability that $q0$ is a final state is set to $\tau'(q0) = \sum_{q\in Q}\gamma(q)\cdot\tau(q)$, which is basically to account for $q0$ stealing the

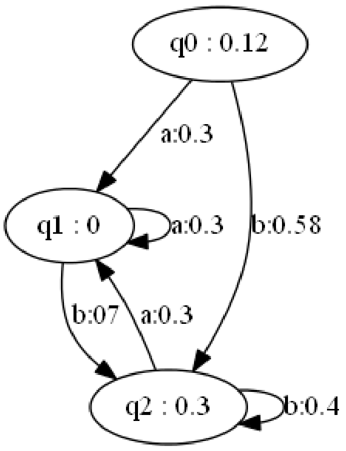


Figure 2.4: Semi-PA with one initial state q_0

role of initial state from all states that previously had a non-zero probability of being initial states. The new δ values for $q0$ are set according to the following formula: $\delta'(q0, a, q) = \sum_{q' \in Q} \gamma(q') \cdot \delta(q', a, q)$. It is proven in [17] that this semi-PA is equivalent to the PA in figure 2.4. ■

Next, we will define the probability of generating a word u, which we will denote $P_A(u)$.

Definition 2.7. The function $P_A : \Sigma^\star \to [0, 1]$ is defined as follows:

$$P_A(u) = \sum_{q,q' \in Q} \gamma(q)\delta(q, u, q')\tau(q')$$

We can extend the definition of this function to subsets U of $\Sigma^\star$ as follows:

$$P_A(U) = \sum_{u \in U} P_A(u)$$

Definition 2.8. Let A be a semi-PA. Then A is a PA if P_A is a distribution over $\Sigma^\star$.

What we will be using in our implementation is a particular type of PA, known as *deterministic probabilistic finite-state automata (DPA)*, defined as follows:

Definition 2.9. A Probabilistic Automaton $A =< \Sigma, Q, \delta, \gamma, \tau >$ is a *Deterministic PA* if $\forall q \in Q, \forall a \in \Sigma, |q' : (q, a, q') \in \delta| \leq 1$.

DPAs present some advantages over PAs:

- Parsing is easier, since only one path is followed.
- Some intractable problems (such as finding the most probable string, or comparing two distributions) become tractable.
- There are a number of positive learning results for DPA that do not hold for PA.

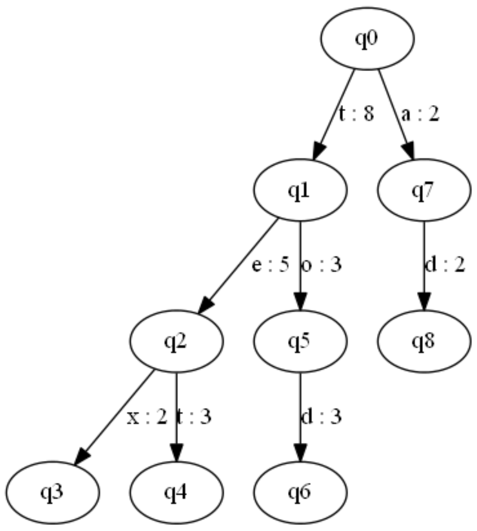


Figure 2.5: Probabilistic Prefix Tree

The particular case of DPA that we will be studying is the *Probabilistic Prefix Tree (PPT)*, where the underlying graph is a tree, rooted at the single initial state $q0$, and which only accepts strings in the sample.

Definition 2.10. A *Probabilistic Prefix Tree* is a deterministic PA with one initial state q_0, satisfying the following added constraint:

- The underlying undirected graph corresponding to all transitions is a tree

Note that this is similar to the trie introduced in the previous chapter. As described in [16], we will construct the PPT such that each transition has a probability which is proportional to the number of times it is used while generating it. Given any finite sample S, a PPT can be easily constructed which generates the empirical distribution ψ_S.

There are some issues to consider when implementing a PA. One issue is that the codomain of functions δ and τ is restricted from $\mathbb{R}^+$ to a subset of $\mathbb{Q}^+$.

Example 2.11. Figure 2.6 gives a representation of a Probabilistic Prefix Tree. Note that we will be using integer values to denote weights associated to each edge. The weight associated to some edge $(q1, a, q2)$ will be denoted by $w(q1, a, q2)$, where $q1, q2 \in Q$ and $a \in \Sigma$. Thus, $\delta(q1, a, q2) = \frac{w(q1,a,q2)}{\sum_{q' \in Q, x \in \Sigma} w(q1,x,q')}$. ■

Note that the weights decrease along each path. This is because in the construction of a Probabilistic Prefix Tree from a sample, each edge that is added to a state other than $q0$ has a corresponding preceding edge leading into that state. Thus, it is also easy to see that the properties of Probabilistic Automata are verified.

Another important issue is that if we want to compute the complexity of the algorithms using PAs, we need a way of expressing the size of a PA. Thus, since the number of bits needed to represent the symbols in the alphabet or the weights is fixed, a correct measure of

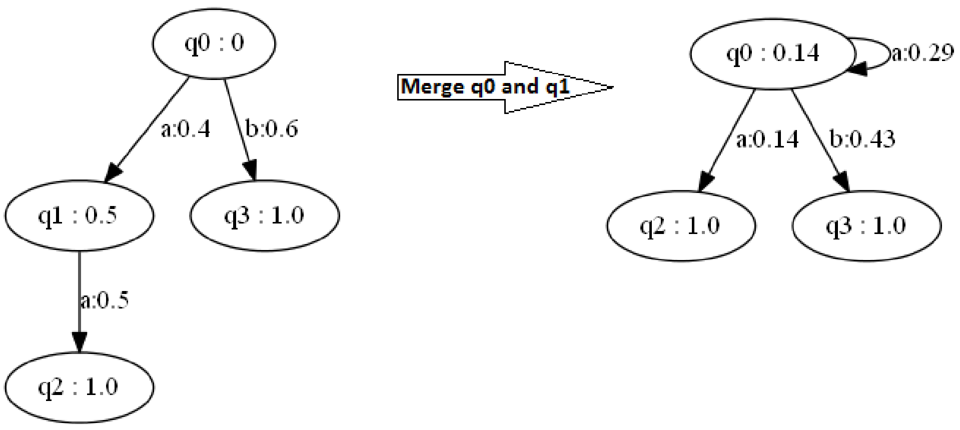


Figure 2.6: State merging

the size of PAs is the sum of the number of states and the size of the alphabet ($|\Sigma|$). Given that the alphabet that we will be working with is fixed, we will only be concerned with reducing the number of states where possible.

Approaches to reducing or minimizing the size of PAs are presented in [16, 19, 20, 13]. These approaches make use of state merging to generalize the learning sample and reduce the size of the automaton. We will define state merging in the context of Probabilistic Automata as follows:

Definition 2.11. *State merging* is the process of obtaining an automaton *A'* from an automaton *A* by combining two or more of the states of *A* into a single state, such that the language generated by A is a subset of the language generated by A'.

Example 2.12. Figure 2.7 shows an example of state merging. As we can see this generalizes the language generated by the automaton in such a way that in the new automaton, generating the empty string is possible with probability 0.14 and generating the string *ab* is also possible, whereas these strings were not part of the language generated by the initial automaton. ■

When performing state merging it should be ensured that the generalization is kept within certain bounds. For example, the Kullback-Leibler divergence measure between two distributions can be used to measure the generalization quality during merging. This divergence measure between two automata A and A' is defined as follows:

Definition 2.12. The *Kullback-Leibler divergence measure* between the distributions generated by two automata A and A' is given by the formula:

$$D(A\|A') = \sum_{x\in\Sigma^\star} P_A(x) \log \frac{P_A(x)}{P_{A'}(x)}.$$

An algorithm for minimizing PAs through state merging using the Kullback-Leibler divergence measure to control generalization is the ALERGIA algorithm [19]. The algorithm starts by constructing a prefix tree based on a sample form a certain language, where each transition has a probability according to the number of times it is traversed when constructing the prefix tree. The algorithm then merges compatible states, where compatibility is established based on the similarity of suffixes generated from those states. The problem with the approach is that there is no way to globally control the generalization from the training sample. The MDI algorithm presented in [16] aims to fix this by considering a new solution compatible with the training data if the divergence increment relative to the size reduction is under a certain threshold.

As we've mentioned, these algorithms capture not only strings from the sample, but also strings that were not part of the sample (but which are likely part of the language from which the sample was extracted). This was also illustrated through an example (Figure 2.7). Clearly, although performing state merging in this fashion could lead to significant state count reduction, with a controlled divergence from the initial language, this is not what we want for our problem, as the set of addresses that we operate on does not allow for generalization of this kind. Thus, we are constrained to using the prefix tree generated from the sample - which has zero-divergence from the training sample - at the cost of increased automaton size. We will, however, look at the alternatives to compressing probabilistic prefix trees, by only merging state pairs (q_1, q_2) where there is one and only one transition from q_1 to q_2, where $\tau(q_1) = 0$, as this is an operation that preserves the distribution in the training sample. Other results concerning the learnability of Probabilistic Automata can be found in [6, 17].

In the following, we will describe the task of parsing a string using a Probabilistic Automaton. Of course, the discussion also applies to Probabilistic Prefix Trees. This is an important operation which determines the probability associated to a certain string according to the distribution modeled by the automaton. As we will see in the next chapter, suggestion making can also be seen as an extension to parsing.

Given the string $x = x_0 x_1 \ldots x_{k-1}$, let $(q_0, x_0, q_1, x_1, q_2, \ldots, q_{k-1}, x_{k-1}, q_k)$ be a path for x in A, i.e. there is a sequence of transitions $(q_0, x_0, q_1), \ldots (q_{k-1}, x_{k-1}, q_k) \in \delta$

Definition 2.13. *Parsing* a string x using a PA A means computing $Pr_A(x) = \sum_{\theta \in \Theta_A(x)} Pr_A(\theta)$, where $\Theta_A(x)$ is the set of all paths for string x in A and $Pr_A(\theta) = \gamma(q_0) \cdot (\prod_{j=1}^{k} \delta(q_{j-1}, x_{j-1}, q_j)) \cdot \tau(q_k)$.

As noted earlier, this task is greatly simplified in the case of Probabilistic Prefix Trees, since the path for string x is unique, therefore $Pr_A(x) = Pr_A(\theta)$, where θ is this unique path.

Also, when there is only one initial state q_0 with $\gamma(q_0) = 1$, the formula for parsing a string x simplifies to $Pr_A(x) = \prod_{j=1}^{k}(\delta(q_{j-1}, x_{j-1}, q_j)) \cdot \tau(q_k)$, since there is only one initial state, and the path associated to string x is unique. This means that a parsing a string x can be done in $O(|x|)$ time, and thus the complexity of parsing a string in a PPT does not depend on the number of states.

Example 2.13. Referring back to the prefix tree in Figure 2.5, computing the probability for a string (i.e. parsing that string) can be done according to the formula in definition 2.13, but it is worth noting that when using weights this way, the value of parsing any particular string is equal to the weight of that string divided by the total sum of weights in the initial state, so all intermediate transition probabilities need not be computed. Hence, the benefit of defining arbitrary distributions over the supported queries in the prefix tree comes at virtually no extra cost for the operations supported by it. ■

2.4 Summary

Our choice of a Probabilistic Prefix Tree as the structure on which to base our implementation was motivated by the prospect of a double advantage of computational efficiency in parsing and suggestion making (owing to the prefix tree structure) and the ability to very flexibly configure the distribution of supported suggestions (owing to the probabilistic nature of the structure).

We have already seen in this chapter how a Probabilistic Prefix Tree can be defined to represent a given distribution over a fixed set of suggestions, and how it can be used to parse a string to compute its probability of occurrence in the given set. In the next chapter we will describe the algorithms that we can build starting from this Probabilistic Prefix Tree structure in order to support the functionality we require. In the following chapters we will also see that in practice the use of a probabilistic, rather than a regular, prefix tree also has the potential to make the suggestion making process far more efficient, without having to resort to tricks such as pre-computing suggestion lists for each state, which greatly limits the flexibility of suggestion making.

Chapter 3

The auto-completion algorithms

In this chapter we will present and analyze the algorithms used to address the requirements identified in the previous chapters. First, in section 3.1, we will introduce an extension to the Probabilistic Prefix Tree that we will be using in our implementation: the Probabilistic Radix Tree. The intuitive idea behind this data structure is to merge the state pairs (q_1, q_2) in the prefix tree where there is only one edge leaving q_1, to q_2, whenever possible. We expect that this approach will result in a considerable reduction of the memory requirements as well as improved speed in making suggestions. We expect the gains to be considerable, given the types of suggestions we are making (address strings), where long suffixes can be compacted most of the time.

In Section 3.2, we will further extend this data structure to support storing $key : weight$ pairs within each state. This will enable us to use the radix tree for indexing keys in an associative data structure, rather than representing the actual set of possible suggestions. Section 3.3 describes the procedure for building such a Probabilistic Radix Tree from data. Apart from allowing for more flexibility when parsing prefixes, this choice will form the basis for supporting substring matching in a way that we hope is scalable in terms of run-time performance, and it will also enable us to store any piece of data we need, associated to a particular suggestion. As we will see in section 3.6, this will also allow us to come up with an algorithm for location biased suggestion making that requires little modification and little overhead compared to the algorithm without location bias support (which will be described in Section 3.4).

The algorithm for supporting error-tolerant prefix matching will also need to be adapted for use with Probabilistic Radix Trees. We will discuss it in Section 3.5. After describing all the algorithms and our expectations in relation to them here, the next chapter will deal with the experimental analysis of these algorithms, focusing on verifying our intuitions with respect to the algorithms, as well as trying to answer other questions to do with their applicability, such as scalability and the effect of possible decisions that need to be made when using these algorithms.

3.1 The Probabilistic Radix Tree

One observation we can make is that given the structure of our suggestion strings, unless two addresses are exactly the same, after merging prefixes, the country name will be spelled out over a number of states equal to the number of characters in the name. One approach to fixing this is to switch to a *Probabilistic Radix Tree* implementation, instead of a Probabilistic Prefix Tree. The idea (also illustrated in Figure 3.1) is to merge those states that have only one outgoing edge with the state that follows along the path of that edge. Equivalently, this can be seen as removing the states with one outgoing edge and replacing them with the corresponding successor, and merging the incoming edge with the outgoing edge by setting the label of the incoming edge to the concatenation of the labels of the two merged edges.

Formally, a Probabilistic Radix Tree can be defined in a way that is very similar to the Probabilistic Prefix Tree, but with an added function with the role of explicitly representing edge labels, and some added constraints:

Definition 3.1. A *Probabilistic Radix Tree* is a deterministic PA with one initial state q_0, with the added function:

- $\alpha : Q \times \Sigma \to \Sigma^\star$, which represents edge labels

and satisfying the following added constraints:

- The underlying undirected graph corresponding to all transitions is a tree
- $\forall q \in Q \setminus \{q_0\}$, we have $(|\{q' \in Q : \delta(q,a,q') > 0 \textit{ for any } a \in \Sigma\}| = 1) \longrightarrow (\tau(q) > 0)$

This definition adds two things to that of the Probabilistic Prefix Tree (see Definition 2.10): the function α and the second constraint. The added function $\alpha(q,a)$ associates a *transition string* to the unique edge leaving q that is associated to a, in case it exists. The second constraint describes a property that applies to all states except for the root of the tree and says that for each such state, if it has exactly one outgoing edge, then it must be final with some probability greater than 0. Equivalently, if the probability of one such state being final is 0, then it must either have none, or more than one outgoing edge, but never exactly one.

To see why this corresponds to our intuition for turning a Probabilistic Prefix Tree into Probabilistic Radix Tree, note that this last constraint in the definition means that we cannot have states with just one outgoing edge and which are not final. Those are exactly the states that we would want to remove by merging their incoming edge (which exists and is unique by the first constraint in the definition and the fact that we are not considering the root here) with the unique outgoing edge. States that are final should not be removed even if they have exactly one outgoing edge, because otherwise the information about the strings that could be successfully parsed until reaching them would be lost.

Example 3.1. In the radix tree in Figure 3.1, let q_0 be the initial state, or root of the tree. The two edges leaving this start state have α values ”berlin” and ”potsdamer$platz$berlin”. That is, edges for which $\delta(q_0,b,q') > 0$ and $\delta(q_0,p,q'') > 0$ have labels given by $\alpha(q_0,b)$ and

$\alpha(q_0, p)$ respectively, where q' and $q'' \in Q$. Also, here, only the leaves are considered final. As we can see, all other states are either the root or have multiple outgoing edges (or both), and hence cannot be considered for merging. ■

Using the definition of the Probabilistic Radix Tree, we present in Algorithm 2 a procedure for obtaining a Probabilistic Radix Tree from a Probabilistic Prefix Tree. The correctness of this algorithm follows immediately by noticing that it closely follows the constraints in the definition of a Probabilistic Radix Tree.

Algorithm 2: Transform(q)

```
input  : A state $q$, in a Probabilistic Prefix Tree
output: -
effect : Merges $q$ with the state that it has a transition to, if there is a unique state
         with this property
if $|\{q' \in Q : \delta(q,a,q') > 0$ for any $a \in \Sigma\}| = 0$ then
    return
if $|\{q' \in Q : \delta(q,a,q') > 0$ for any $a \in \Sigma\}| = 1$ and $\tau(q) = 0$ then
    if $q$ is the root then
        Transform(q');
    else
        Let $p$ be the direct ancestor of $q$ through symbol $c$;
        $\alpha(p,c) \longleftarrow \alpha(p,c) \cdot a$;
        for the unique q' and $a$ in this case do
            $\tau(q) \longleftarrow \tau(q')$;
            for all $b \in \Sigma$ such that $\delta(q',b,q'') > 0$ for some $q'' \in Q$ do
                $\delta(q,b,q'') \longleftarrow \delta(q',b,q'')$;
                Transform(q);
else
    for all states $q'$ such that $\delta(q,a,q') > 0$ for some $a \in \Sigma$ do
        Transform(q');
```

This algorithm operates recursively and should be invoked with the root of the Probabilistic Prefix Tree as a parameter. The first case that is handled is the stopping condition. That is, if there is no edge leaving the current state, there is no pair of edges to merge. This corresponds to reaching the leaves of the tree. Otherwise, if the current state has only one outgoing edge, we merge its unique incoming edge with this unique outgoing edge, replacing the state with the state that this latter edge leads to. Note that this is only done if $\tau(q) = 0$. This ensures the probability of the single outgoing edge is 1 (by Definition 2.5) and the only information that needs to be maintained while merging is the label of the edge being removed.

However, if the current state is the root of the tree, we try to apply this procedure on its unique descendant, as the root is the only state that has no incoming edge. The α value that is being modified is that corresponding to the parent of the current state that is being

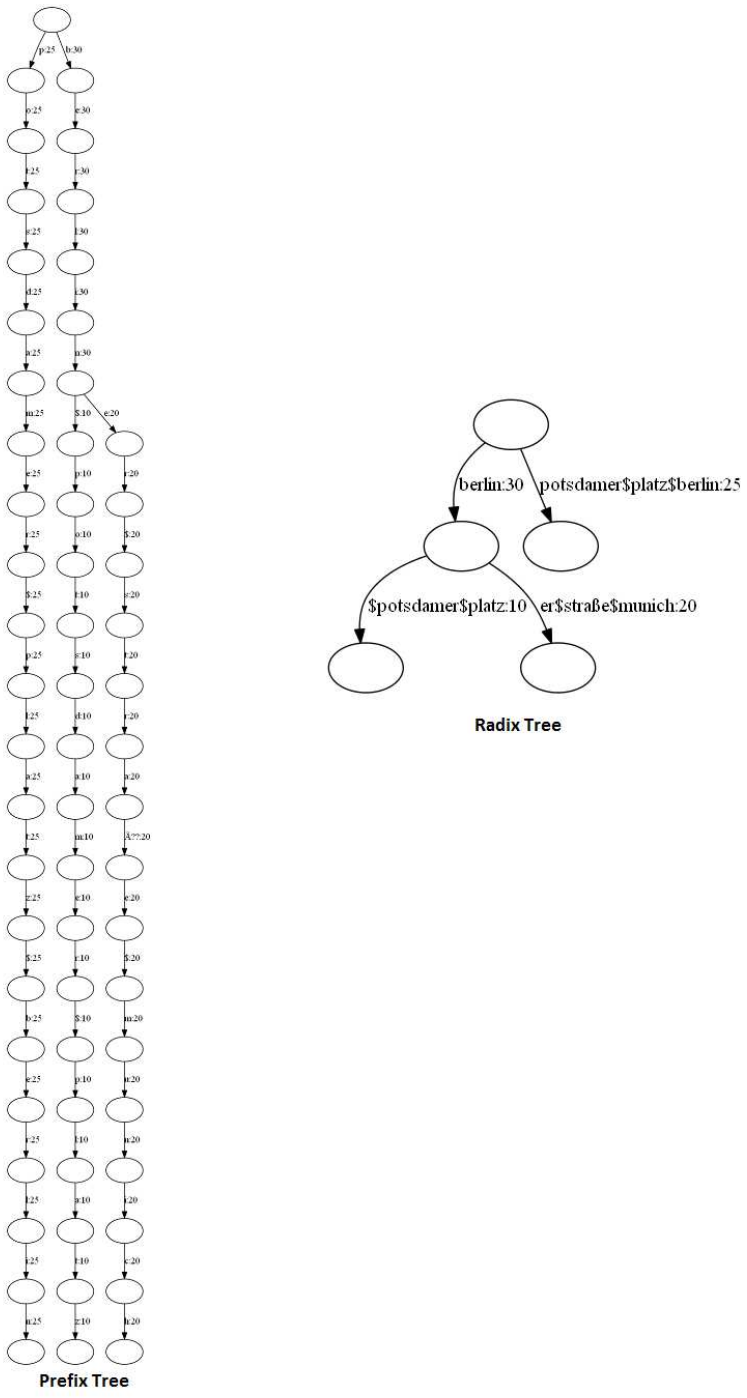


Figure 3.1: Transforming a Probabilistic Prefix Tree to a Probabilistic Radix Tree

replaced. We are essentially adding the symbol corresponding to the outgoing edge of the state that we are removing (through merging) to the label of the state leading to the current state. Hence, any string that could be parsed in the initial prefix tree, can be parsed in the resulting radix tree, as the complete information regarding the path labels is preserved through the function α. Note also that in case states are merged, the recursive call is made on the same state. Intuitively, this corresponds to the state consuming outgoing edges (and maintaining the information about the consumed edge labels in the α function corresponding to the parent of the current state) as long as they are unique. When this process cannot be continued, recursive calls are made on the states to which the current state has defined transitions, in order to apply the process all the way down the tree, until the recursive calls bottom out at the leaves level. The effects of running this procedure on a a Probabilistic Prefix Tree, in the form of the resulting Probabilistic Radix Tree, are shown in Figure 3.1.

Example 3.2. In Figure 3.1, calling Algorithm 2 with the root of the Probabilistic Prefix Tree as a parameter will result in calling the algorithm recursively for its two descendants, as the root has more than one of them. For the left descendant, the condition for merging would be met in all in successive recursive calls on this state until all the symbols in the path down to the leaf are added to the transition string from the root to the current state. In the end, this transition string will be $potsdamer\$platz\$berlin$ and the current state will be final and have no more outgoing edges, so recursive calls bottom out.

On the other branch, a similar process will take place until the state with two outgoing edges is met. Note that this state is now a direct descendant of the root, by following the edge now labeled *berlin*. According to the way Algorithm 2 works, since the current state now has two outgoing edges, the algorithm is called recursively for each of them, and merging is continued as before down each path, until the leaves are reached. ■

To sum it up, if the conditions mentioned above for merging are met, the current state is essentially replaced by its only descendant, and the label of the edge leading to the current state is updated in order to preserve the labeling of the complete path leading to its descendant. To see that the probabilities associated to supported terms are not affected, note that by Algorithm 2, we are only removing edges with probability 1 associated to them. Since this is the multiplicative identity, and by the way terms are parsed (see Definition 2.13), we can conclude that removing such edges does not affect the probabilities of the supported terms.

The use of radix trees as opposed to prefix trees can be expected to work well for many vocabularies, but it should be a particularly useful approach in the context of geocoding, where the patterns that we are trying to predict constitute addresses. To get an intuition as to why this can be expected in the context of geocoding, consider all the street-level addresses in Amsterdam. All will have the form "*street-name*, Amsterdam, The Netherlands". For all distinct street names, the whole string "Amsterdam, The Netherlands" could label a single transition, binding only two states, instead of generating 25 states and the transitions between them.

Remark 3.1. As such labels will exist for all addresses in Amsterdam, this leads to the idea of further reducing the number of states required, through the use of a data structure called a *Directed Acyclic Word Graph*.

Definition 3.2. A *Directed Acyclic Word Graph* is a directed acyclic graph G=(V,E) for which we define a function $s : E \rightarrow \Sigma$ which assigns a symbol to each edge.

Note that apart from the lack of weights or probabilities assigned to each edge, unlike a Probabilistic Prefix Tree or Radix Tree, the Directed Acyclic Word Graph also does not impose that the underlying (induced) undirected graph be a tree. The construction of a Directed Acyclic Word Graph from a given set of strings is described in [21]. The idea behind this data structure is to merge not only prefixes, but also common suffixes of the strings over which we support searching.

Figure 3.2 shows the construction of a Directed Acyclic Word Graph starting from a prefix tree representing the strings {tap,taps,top,tops}, with the character '$' used to mark the end of strings.

This data structure can be used to represent large vocabularies in a very compact manner. Although this could once again lead to significant state count reduction in our context, such a representation lacks the power of representing a probability distribution over a given vocabulary, and thus cannot be used to solve the problem of offering ranked auto-completion suggestions. Therefore, we will construct our algorithms around the Probabilistic Radix Tree. ■

Although it is perhaps conceptually simplest to understand radix trees as they can be obtained from prefix trees, as depicted in Figure 3.1 and explained in Algorithm 2 - by collapsing multiple states in the prefix tree into a single state where possible and merging edges by preserving information in their labels - we will not be constructing radix trees from prefix trees, as this would mean losing the main advantage of using radix trees in the first place: reduced memory requirements.

In order to avoid the high memory requirements of representing our data using prefix trees, we will present all our algorithms in terms of radix trees, and we will start with the construction of radix trees directly from sample data consisting of addresses to be suggested, without having to first represent the strings as a Probabilistic Prefix Tree.

Note that from an implementation perspective, reducing the number of states comes at the cost of storing the strings associated to each transition, that is, the function α (see Definition 3.1) also needs to be represented with each state. This is not necessary for a Probabilistic Prefix Tree. Thus, representing a single state of the radix tree would require more memory compared to the prefix tree. However, we expect that the state count reduction will far outweigh the extra storage needed with each state and the overall memory requirement should drop significantly.

But before explaining the construction of a Probabilistic Radix Tree from data, in the following section we will describe how the Probabilistic Radix Tree can be used to store keys to suggestions and how this can help us meet the requirements formulated in Chapter 1.

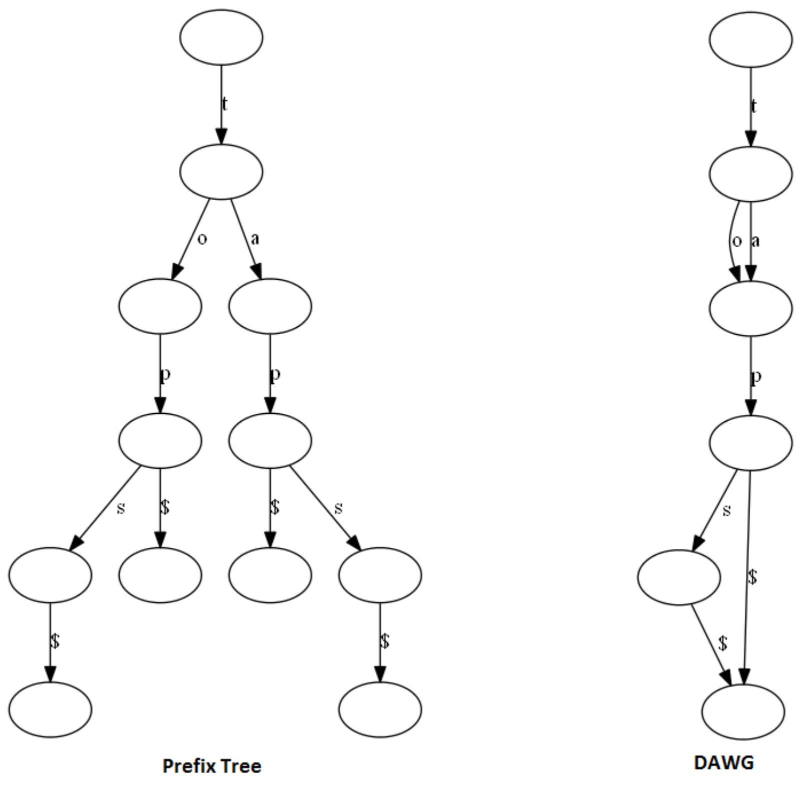


Figure 3.2: Transforming a Probabilistic Prefix Tree to a Directed Acyclic Word Graph

3.2 Using Probabilistic Radix Trees to index suggestions

In this section we describe how a Probabilistic Radix Tree can be used to parse keys in an associative data structure. The definition of the Probabilistic Radix Tree as presented in the previous section does not change, but we add the following implementation detail: each state will store a possibly empty set of (*key* : *weight*) pairs, where $\sum_{(key:weight)\in q} weight = \tau(q)$, with the convention that if the set is empty, then $\tau(q) = 0$.

Example 3.3. The use of a radix tree in order to index an associative data structure is depicted in Figure 3.3. As in the previous chapter, we have used integer weights instead of probability values to annotate the states and transitions. It is easy from these to derive the actual probability values, based on the radix tree definition. Each state stores a (possibly empty) set of references to associations (in our context a completion suggestion for an address) and their associated weights. This decouples the parsed string from the actual suggestion string returned to the user. Thus some flexibility is ensured when parsing prefixes. For example, the radix tree in Figure 3.3 contains strings that are all lower case and any non-empty sequence of symbols considered token separators is replaced by the single placeholder symbol: $. If user-typed prefixes are also normalized according to these rules prior to parsing, this ensures case insensitive parsing with some freedom in choosing how to separate tokens (e.g. spaces and commas can be used interchangeably).

This separation between matched strings and actual suggestion strings is also the key to offering suggestions in a well structured format. For instance, as seen in the radix tree in Figure 3.3, both ”Potsdamer Platz, Berlin” as well as ”Berlin, Potsdamer Platz” can be supported by the radix tree and have the same key associated to them, so regardless of the way the input is given, the suggestion will be ”Potsdamer Platz, Berlin”, which is a nicely formatted address string. This comes at the cost of adding multiple strings in the radix tree for each association. On the other hand, because the parsing performance only depends on the length of the string to be parsed, as we’ve seen in the previous chapter, the prefix matching performance should not suffer as a result, and we expect that the suggestion making performance also won’t suffer too much from this choice.

But perhaps the main advantage of our approach is that the weight of the association in each case is free to differ. As we can see in the example, parsing ”Potsdamer Platz, Berlin” gives us this suggestion with a weight of 25. Parsing ”Berlin, Potsdamer Platz”, on the other hand, gives us the same suggestion, but with a weight of 10. This makes a difference when we have to make suggestions and several parsing options need to be explored in order to choose the most likely one, as higher weight suggestions will be prioritized. Thus, we are giving the associations for badly structured queries a lower weight. As an example of why we may want this, consider a user typing ”Berlin” in the search box. Apart from the suggestion for ”Potsdamer Platz, Berlin” (because we’re also supporting improperly formatted address queries) we could also offer ”Berliner Straße, Munich” as a suggestion. Note that parsing ”Potsdamer Platz, Berlin” gives us a higher weight (25) than ”Berliner Straße, Munich” (20). This could be because the first street is somehow considered more important, or we know people search for it more often, or maybe simply because it is situated in a capital city. However, because we store different strings to be parsed in the radix tree for the same suggestion, we are free to associate a lower weight to the first suggestion when it is retrieved by parsing ”Berlin, Potsdamer Platz” compared to when it is retrieved by parsing ”Potsdamer Platz, Berlin”. That is, we assume the user is in fact typing a prefix of a well formatted address. This is not a hard constraint, however. We may choose not to obey this rule depending on the difference in the weights of the two streets or the user location. ■

The choice of an associative data structure also presents another significant advantage for geocoding applications: since the radix tree need only store a pointer to an auto-completion suggestion, the suggestion could consist of the textual representation of a complete address, as well as other data, such as a complete specification of all the fields in the suggestion. This can help improve search results in case a user chooses a suggestion and does not edit it, as the associated suggestion can be used as a structured query, thus eliminating the need to rely on the free-text query support of the geocoder, and producing better accuracy. Moreover, the service managing these suggestions can be completely decoupled from the radix tree or even be implemented to run on a different machine, thus offering more flexibility in resource management. But perhaps the most important benefit of using an associative data structure - which enables us to keep extra information about the suggestions made - will be highlighted when we introduce our strategy for implementing a location bias.

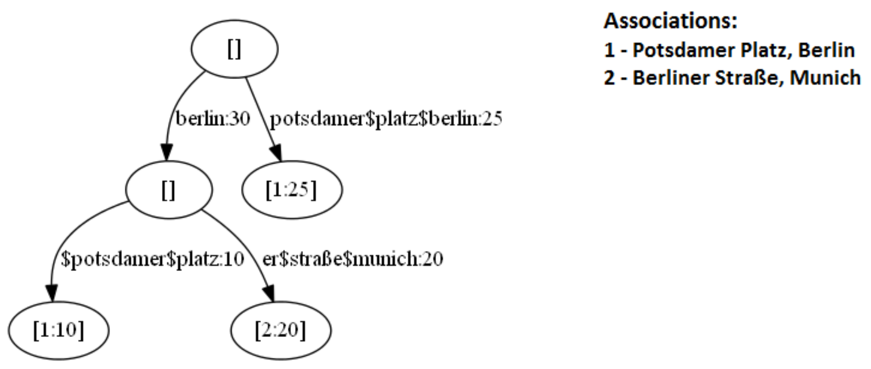


Figure 3.3: Using a Probabilistic Radix Tree to parse keys in an associative data structure

For this, we assume that each suggestion key comes with a way of retrieving the latitude and longitude of the location of that suggestion.

3.3 Probabilistic Radix Tree construction

As stated in the previous sections, we would like to construct a Probabilistic Radix Tree from data, without having to build a Probabilistic Prefix Tree first. Thus, instead of merging states of a Probabilistic Prefix Tree as we did in Section 3.1, we would prefer to use as few states as possible from the start, labeling edges between them with strings as long as possible, and only add new states when necessary.

For this, we would like to define a method which we will call $AddTerm(q, term, k, w)$ to add the string *term* such that it can be parsed starting from state q in order to retrieve suggestion k with an associated weight w.

Example 3.4. Figure 3.4 shows the construction of a Probabilistic Radix Tree through successive calls to the *AddTerm* method described above. We start with just the root of the tree, and the first term that is added to it is *potsdamer$platz$berlin*. For this, we only need to add one state, as there is a single path in our tree that leads to a suggestion, from the root to the state which now stores suggestion key 1, with a corresponding weight of 25. Then the terms *berlin$potsdamer$platz* and *berliner$straße$munich* are added, making sure at each step that the result is equivalent to creating a Probabilistic Prefix Tree and then merging states wherever possible, as described in Algorithm 2. ■

To describe the algorithm for constructing the Probabilistic Radix Tree from data, we first describe the representation of a single state q as consisting of the following fields and their relation to the definition of a radix tree:

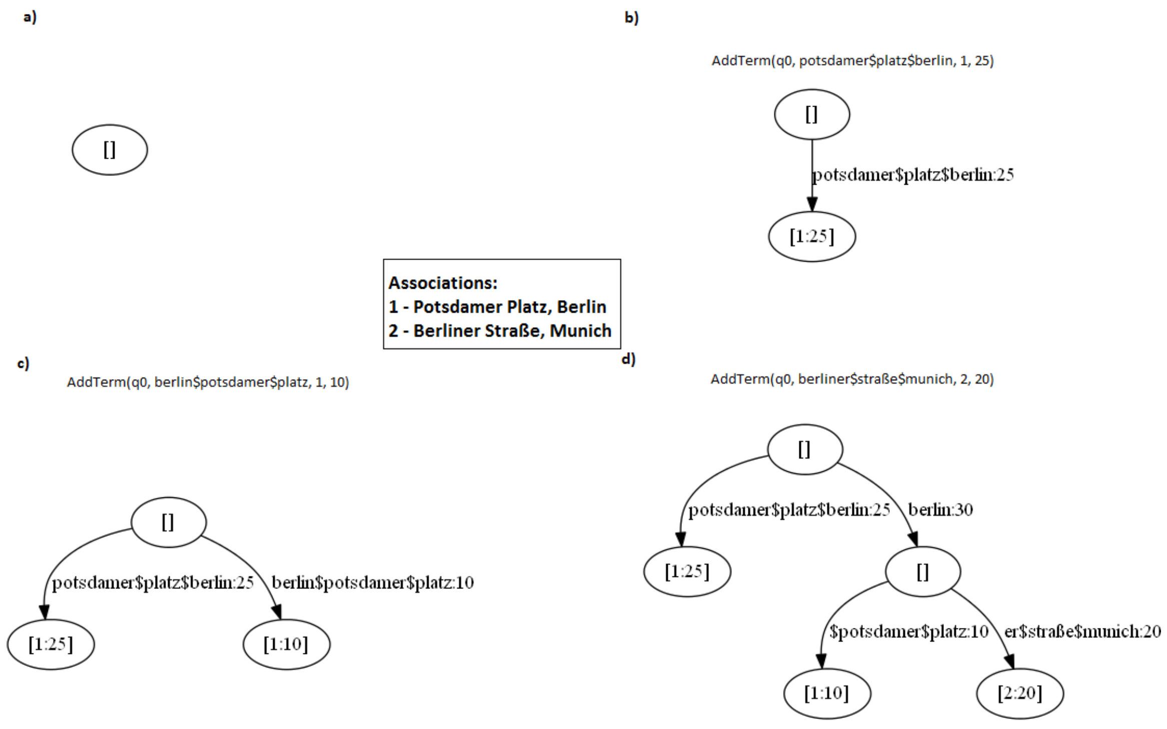

a)
[]
b)
AddTerm(q0, potsdamer$platz$berlin, 1, 25)
[]
potsdamer$platz$berlin:25
[1:25]
Associations:
1 - Potsdamer Platz, Berlin
2 - Berliner Straße, Munich
c)
AddTerm(q0, berlin$potsdamer$platz, 1, 10)
[]
potsdamer$platz$berlin:25
berlin$potsdamer$platz:10
[1:25]
[1:10]
d)
AddTerm(q0, berliner$straße$munich, 2, 20)
[]
potsdamer$platz$berlin:25
berlin:30
[1:25]
[]
$potsdamer$platz:10
er$straße$munich:20
[1:10]
[2:20]

Figure 3.4: Constructing a Probabilistic Radix Tree from data

- *transition_string*$[c]$, which stores the label on the edge associated to character c leaving the current state. This corresponds to the values of $s \in \Sigma^\star$, such that $\alpha(q,c) = s$. Hence, *transition_string*$[c]$ will always be a string of length at least 1, and will always start with c.

- *transition_count*$[c]$, which stores the number of times the transition which is labeled *transition_string*$[c]$ is used while generating the radix tree. This value, divided by the *total_count* (see below), is the actual value of $\delta(q,c,q')$ for the unique $q' \in Q$ for which $\delta(q,c,q') > 0$, according to the properties of radix trees.

- *transition_state*$[c]$, which stores a reference to the state at the end of the transition arrow labeled *transition_string*$[c]$, corresponds to those states $q' \in Q$ for which $\delta(q,c,q') > 0$.

- *key_count*$[k]$, which corresponds to the number of times the current state is associated with suggestion k (as a final state). Hence $\tau(q) = \frac{\sum_k key\_count[k]}{total\_count}$ (see below the definition of *total_count*).

- *total_count*, which is simply the sum of transition counts and key counts across all defined transition characters and key values for state q.

Example 3.5. For the radix tree constructed in Figure 3.4, if we consider the start state, it will have $transition\_string[b] = berlin$ and $transition\_string[p] = potsdamer\$platz\$berlin$, with the *transition_state* fields pointing to the corresponding states, and $transition\_count[b] = 30$, while $transition\_count[p] = 25$, for a $total\_count = 55$. There are no *key_count* values associated to the start state, however the terminal states in this case have the following key counts: $key\_count[1] = 10$, $key\_count[2] = 20$ and $key\_count[1] = 25$ ■

Note that this ensures we are building a Probabilistic Radix Tree as it was defined in this chapter. To see this, note that the required constraint $\tau(q) + \sum_{a \in \Sigma} \sum_{q' \in Q} \delta(q, a, q') = 1$ is verified for all states by the definition of *total_count*. Moreover, we assume a construction of a radix tree in which there is exactly one state -which we will denote q_0- designated as initial state, for which $\gamma(q_0) = 1$ and assume that $\gamma(q) = 0$ for all other states q, so the relation $\sum_{q \in Q} \gamma(q) = 1$ is trivially verified. In fact, assuming this implicit definition of γ, we will not represent it explicitly. Instead, we just assume that the construction of a radix tree must start with the single initial state, which must always exist.

Next we mention two functions that will be used in our algorithm for constructing radix trees from data. We will skip the actual implementations of these functions, as they are trivial:

- *AddAssociation(q,k)* - adds an association key k to state q and adjusts the fields of q to a new consistent state. That is, the $key\_count[k]$ is incremented, or initialized to 1 if it did not exist, and the *total_count* of state q is also incremented by 1. We will consider that this function is implemented such that it takes $O(1)$ time to complete.

- *AddTransition(q,s,q')* - adds a transition labeled s from state q to state q', where $s \in \Sigma^\star$. This is a utility function that is not concerned with maintaining the radix tree structure, rather it just adds the requested transition updating the fields describing the two states appropriately. We only mention this function here because we will make use of it in the radix tree construction. Hence, all it does is update $transition\_string[s[1]]$ to s and $transition\_state[s[1]]$ to q', while $transition\_count[s[1]]$ and *total_count* are each incremented by 1. This function will also be assumed $O(1)$-time computable. We will assume that we also have a variation of this function: *AddTransition(q,s,q',w)*, which is equivalent to calling *AddTransition(q,s,q')* w times.

The function that we will define and use to construct the Probabilistic Radix Tree is *AddTerm(q,s,k)*, where the term to be added is denoted s, where $s \in \Sigma^\star$. This will be the same as calling the function *AddTerm(q,s,k,w)* discussed earlier, with $w = 1$. Generalizing for arbitrary values of w is trivial. The result of calling this function is that starting from q, the string s can be parsed in the radix tree and the resulting state should have suggestion k associated to it.

Constructing a Probabilistic Radix Tree from a data set consisting of address strings s will be done through successive calls to *AddTerm(q_0,s,k)*, where q_0 is the designated start state of the Probabilistic Radix Tree.

Because we are working with a radix tree, the *AddTerm(q,s,k)* function is considerably more complex than the similar function for prefix trees. The situations that need to be considered by this function are depicted in Figure 3.5, and Algorithm 3 implements this function. The first case to consider is when there are no outgoing edges from state q whose labels start with $s[1]$. In this case, a new state needs to be created, which will hold association k and the edge from q to this new state is labeled s. Once this new state is created, this can be accomplished by calling the utility function *AddTransition(q,s,q')*, described earlier.

The other easy case is when there is an edge from q to some state q' whose label not only starts with the same character as s, but is exactly s. In this case, the different fields of state q are updated to reflect the addition of extra weight on that transition (this means incrementing $transition\_count[s[1]]$ and $total\_count$) and suggestion k is added to q', either incrementing $key\_count[k]$ or initializing it to 1, depending on whether or not it had already been initialized.

The remaining cases are more complex. Since we know that we do not fall under the first case, because our algorithm would have ended there, we can conclude that there is a transition s' from q to some state q' such that $s[1] = s'[1]$. Moreover, by this point we can safely assume that $s \neq s'$, because that would have been handled in the case we just described. Therefore we can write $s = vw$ and $s' = vt$ with $w, t \in \Sigma^\star$ and $v \in \Sigma^+$, such that $w \neq t$ and w and t do not share a prefix. Note that this implies that we cannot have $t = \lambda$ and $w = \lambda$ at the same time. The remaining cases are as follows:

- $t = \lambda$. This basically means that the transition string is a prefix of the term to be added. The action taken here is represented in Figure 3.5, under case 3. We can safely consume the prefix of the term that is added which is equal to the transition string and add the remaining piece of the term to q' through a recursive call.

- $w = \lambda$. This is basically the case when the term to be added is a prefix of the transition string between q and q'. The solution in this case is shown in Figure 3.5 under case 4: we create a new state q'' in order to store the new suggestion k at the end of the string that is associated to it (which ends after v) and making sure we still have a path to q' for exactly the same strings that used to lead us to q', i.e. those passing through q and ending after seeing vt from there.

- $w \neq \lambda$ and $t \neq \lambda$. The solution in this case is shown under the last case in Figure 3.5. This solution is similar to that for the previous case, except the new state q'' that is added between q and q' does not store suggestion k any more. Instead it adds a new transition labeled w to yet another new state $q^{(3)}$ which stores suggestion k. The transition labeled t to q' is also added like before, in order to maintain the information needed to parse strings previously added to the radix tree.

Algorithm 3 is a rather straightforward implementation of exactly the cases described here and depicted in Figure 3.5. Constructing a Probabilistic Radix Tree can be accomplished through successive calls to $AddTerm(q_0, s, k)$ for the different terms that need to be supported $s \in \Sigma^\star$, where q_0 is the root of the radix tree.

By the discussion of the cases above, and the observation that these cases sum up all possible scenarios that can be encountered when adding a string s to the radix tree state, we can conclude that calling $AddTerm(q, s, k)$ ensures that we are able to parse the string s

Algorithm 3: AddTerm(q,s,k)

```
input  : A state q, a string s and an associated suggestion k
output: -
Result: State q will support parsing the string s, resulting in the association k
if transition_state[s[1]] = null then
    q’ ⟵ new_state;
    AddTransition(q,s,q’);
    return;
s’ ⟵ q.transition_string[s[1]];
index ⟵ longestCommonPrefixIndex(s,s’);
v ⟵ s[1..index];
w ⟵ s[index + 1..s.length];
t ⟵ s’[index + 1..s’.length];
q’ ⟵ q.transition_state[s[1]];
if $(w = \lambda) \wedge (t = \lambda)$ then
    q.transition_count[s[1]] ⟵ q.transition_count[s[1]] + 1;
    q.total_count ⟵ q.total_count + 1;
    AddAssociation(q’,k);
    return;
if $t = \lambda$ then
    q.transition_count[s[1]] ⟵ q.transition_count[s[1]] + 1;
    q.total_count ⟵ q.total_count + 1;
    AddTerm(q’,w,k);
    return;
if $w = \lambda$ then
    q” ⟵ new_state;
    q.transition_count[s[1]] ⟵ q.transition_count[s[1]] + 1;
    q.total_count ⟵ q.total_count + 1;
    q.transition_string[s[1]] ⟵ v;
    q.transition_state[s[1]] ⟵ q”;
    AddAssociation(q”,k);
    AddTransition(q”,t,q’,q.transition_count[s[1]] - 1);
    else
        q” ⟵ new_state;
        q.transition_count[s[1]] ⟵ q.transition_count[s[1]] + 1;
        q.total_count ⟵ q.total_count + 1;
        q.transition_string[s[1]] ⟵ v;
        q.transition_state[s[1]] ⟵ q”;
        AddAssociation(q”,k);
        AddTransition(q”,t,q’,q.transition_count[s[1]] - 1);
        AddTerm(q”,w, k);
```

from q in order to retrieve suggestion k, as established by the following theorem.

Theorem 3.1. *Calling $AddTerm(q, s, k)$ ensures that string s can be parsed from state q in order to retrieve suggestion k with a weight equal to the number of times it has been added, without breaking this property for previously added strings.*

Proof. We can prove this by induction. Since we do not have to deal with empty strings, we can take $|s| = 1$ as our base case. Note that only cases 1, 2 and 4 from Figure 3.5 are applicable to this situation. From the previous discussion of Algorithm 3, in each such case we create ways of reaching a state storing suggestion k at the end of a path labeled s from q without losing previously existing information in the tree.

Now, assuming this property holds for all strings s', with $1 \leq |s'| < n$, we consider a string s having length n, where $n > 1$. Again, it is easy to see from the discussion of Algorithm 3 that in cases 1, 2 and 4 we add s so that it can be parsed from q in order to retrieve k with a weight equal to the number of times it has been added, and maintaining previously existing information in the tree. In case 3, a prefix of s already gives us a way to retrieve an existing state q'. Since this prefix is not empty, when removing it, we will have a string of length less than n to add to that state. By our induction hypothesis, by first parsing the prefix of s to retrieve state q', we will be able to parse the remainder of s in order to retrieve suggestion k with a weight equal to the number of times it has been added. In case 5, we need to break up an existing edge with a label that shares only a prefix smaller than the length of either string with s. In that case, a new state is added at the breaking point, and edges are added and weights are updated in order to maintain all the information, as discussed in the cases above. Coupled with the observation that these cases list all possibilities that we have to face when adding a string s to state q, we can conclude that the desired properties are maintained throughout the construction of a Probabilistic Radix Tree through successive calls to $AddTerm$. □

Another important question is whether or not we have obtained a result equivalent to running state merging on a Probabilistic Prefix Tree. That is, is the Radix Tree built through successive calls to $AddTerm$ minimal, in the sense that no further state merging as described in Algorithm 2 can be performed?

Theorem 3.2. *Given a Probabilistic Radix Tree as it results through repeated calls to $AddTerm(q_0, s, k)$, merging of any two states as described in Algorithm 2 is impossible.*

Proof. To see this, note that we build the Probabilistic Radix Tree, starting from just the designated start state, which can never be merged by Algorithm 2. The function $AddTerm$ only creates states that either have multiple outgoing edges or have associations stored within them, and thus a value $\tau > 0$. By the description of Algorithm 2, such states would never be considered for merging with descendants. Therefore, we have generated a Radix Tree from data, that is already minimal, in the sense that its state count cannot be further reduced by applying the merging procedure in Algorithm 2. □

Theorems 3.1 and 3.2 ensure that we are able to use Algorithm 3 to build a Probabilistic Radix Tree directly from data, without having to first build a Probabilistic Prefix Tree

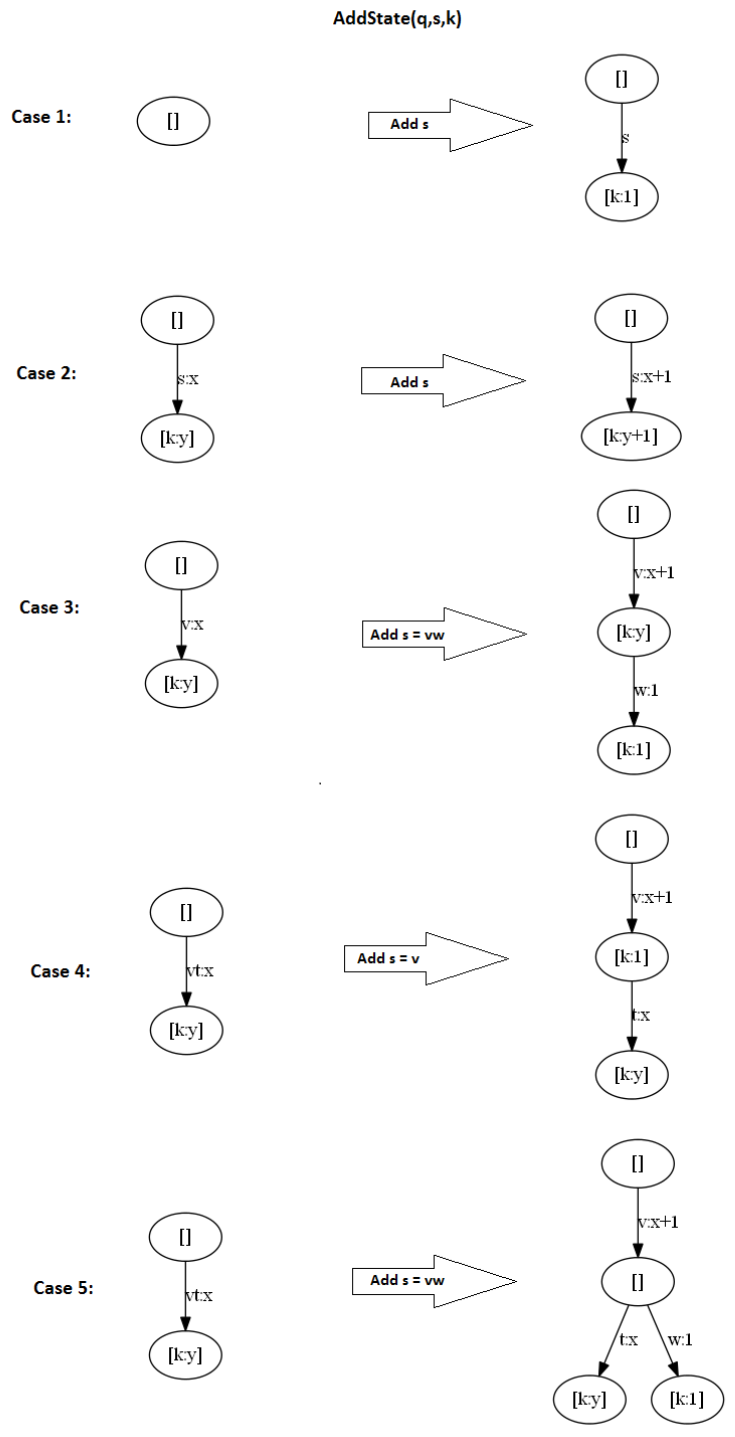


Figure 3.5: The cases encountered in adding a term to be parsed form a state in the Probabilistic Radix Tree

and whose state count cannot be further reduced by merging states. Note that to do this, we needed a considerably more complicated procedure than the equivalent procedure for adding terms to a Probabilistic Prefix Tree, but this extra effort should pay off, and not only through reduced memory requirements. Although the task of adding terms to a radix tree in a way that maintains the integrity of the data structure as it is defined was more complicated compared to the equivalent operation on a prefix tree, the actual computational complexity of adding a term s as a supported term to the radix tree is still $O(|s|)$ time if we consider the time spent updating the data structures describing the states computable in constant time. This is because any recursive call is made after parsing a portion of the string which is later never visited again, and all other control paths that do not make recursive calls are computable in a constant amount of time. Moreover, depending on the actual implementation, if updating the data structures does turn out to be a rather costly procedure, the radix tree may actually be built faster than a prefix tree, because there may be far fewer states to update along the path taken to add s to the radix tree.

Note that to make this discussion simpler, we assumed that the radix tree sets counts depending only on the number of occurrences of a string in a given sample. The actual implementation should support addition of weighted strings, as in Figure 3.4, but this can be achieved through a minor change in the algorithm.

Now that we have seen how to construct a Probabilistic Radix Tree from a given set of suggestions and we have described the structure of such Radix Trees, we can define the algorithms that implement the functionality that we require from the auto-completion system. Hence, in the following sections we will look at the following algorithms on a given Probabilistic Radix Tree:

- *GetState(sate,prefix)*, which retrieves a state of the Probabilistic Radix Tree, obtained from parsing the supplied string *prefix*, starting from the given state, *state*. This algorithm will be introduced because it is useful for suggestion making, as it returns the subtree of possible suggestions corresponding to a given prefix string.

- *GetStates(prefix,distance)*, which is the algorithm for retrieving the set of states that can be obtained from parsing any of the strings within edit-distance *distance* from the supplied prefix string. This algorithm will enable us to offer error-tolerant prefix matching and suggestion making.

- *GetSuggestions(prefix,n)*, which is the algorithm used for getting a ranked list of the top *n* suggestions for a supplied *prefix* string. This algorithm will rely on the *GetState* algorithm mentioned earlier to offer suggestions for a supplied prefix, and on the *GetStates* algorithm when error-tolerance is required for parsing the supplied prefix.

- *GetSuggestions(prefix, n, lat, long)*, which is a variation on the suggestion-making algorithm, which also allows for taking a location-bias into consideration, thus providing an altered list of top-n suggestions, with a relevance measure being a function of both the weight of the suggestion and its proximity to the supplied bias point (the lat-long parameter pair).

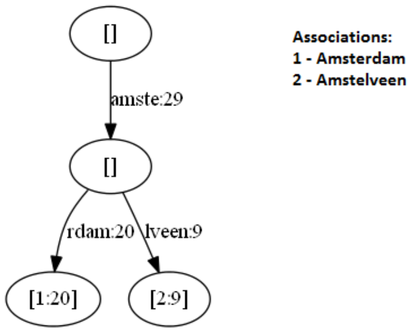


Figure 3.6: A Radix Tree used for suggesting ’Amsterdam’ and ’Amstelveen’

3.4 Making suggestions

In this section we will discuss the algorithm for retrieving a ranked list of the top-n suggestions for a supplied prefix. This algorithm is the first to directly address requirements from those introduced in Chapter 1. That is, this algorithm implements the two main requirements and Optional Requirement 1. This latter requirement is not directly addressed by the algorithm, but it is met if the radix tree that the algorithm operates on has been properly configured to offer such support, as described in the previous sections. In order to discuss the suggestion making algorithm, we start by describing a convenience procedure, *GetState(state,prefix)*, mentioned in the previous section, which returns the state obtained by parsing the string *prefix*, given a certain start state as its first parameter.

Algorithm 4 describes a recursive function that achieves this. In order to parse a prefix on the whole tree, the start state of the tree should be supplied as the *state* parameter. Note that the algorithm may return a state that the supplied prefix does not reach, but which it can be extended to reach. This behavior is suitable for the task of suggestion making.

Example 3.6. Consider Figure 3.6. If we try to get the state for string ”amster”, then by Algorithm 4, we will end up with a state having ”Amsterdam” as an associated suggestion, although we did not enter ”amsterdam”, which is the only string sufficient to reach that state. However, if we are interested in making suggestions for extending the entered string ”amster”, it is clear that this state is a good candidate, along with any state that may follow it down the radix tree. ■

We can now finally introduce the algorithm that we will use to produce a list of ranked suggestions for an input prefix string. Algorithm 5 achieves this by making use of a priority queue Q to store entries which can be either states or association keys. The implementation is such that the top element in the queue (and the first one to be retrieved when calling $Q.top()$) is the one with the largest weight. Unless otherwise stated, it should be assumed that whenever we refer to a priority queue, we are using a max-priority queue. Hence the first n association keys that come out of the priority queue as a result of the algorithm execution will be our top-n suggestions. As long as there are states that are more promising, they are expanded (all their associated suggestions and states that follow from them are added to the priority queue). The correctness of this algorithm is established by Theorem 3.3.

Algorithm 4: GetState(sate,prefix)

```
input  : A current state state, and a prefix string prefix
output: The state that parsing the input prefix leads to, from the given node
if prefix.length = 0 then
    return state
newState ⟵ transition_state[prefix[1]];
if newState = null then
    return null
transitionString ⟵ state.transition_string[prefix[1]];
if prefix = transitionString then
    return newState
else
    index ⟵ longestCommonPrefixIndex(prefix,transitionString);
    if index = transitionString.length then
        GetState(newNode,prefix[index+1..prefix.length]);
    else
        if index = prefix.length then
            return newNode
        else
            return null
```

Theorem 3.3. *The suggestions returned by Algorithm 5 are the top n suggestions that can be retrieved from the subtree rooted at the state obtained by the supplied prefix.*

Proof. Based on the radix tree construction, the weight of any association is added to all the weights of all transitions leading to it, and thus to the *total_weight* member of any state traversed along the path to adding that association. Hence, if the algorithm prefers a certain suggestion to some state at a certain iteration of the *while* loop, then that state cannot lead to a better suggestion, so it is safe to report that suggestion as part of the desired result, ahead of any suggestion that may be found by expanding the state.

□

Example 3.7. Consider the radix tree that we built in Figure 3.4. If we want to retrieve the (single) top suggestion for the prefix 'be', the function *GetState(startState,'be')* would return the state at the end of the edge labeled 'berlin:30', and this state would be added to the queue. As this state is expanded, we next add to the queue the state at the end of the edge labeled 'potsdamer$platz' with a weight of 10 and the state at the end of the edge labeled 'er$straße$munich' with a weight of 20. As the entry with the highest weight currently in the queue, this last state is extracted next, and expanded such that suggestion 2 is added to the queue with a weight of 20. As this weight is still the highest in the queue, it is extracted next, and since it is a suggestion, it is added to the final result, which also ends the loop, since we were only looking to make one top suggestion. Thus, we have come up with a result without expanding all states, since we had enough suggestions and we knew expanding more states would not lead to any better suggestions. In this simple example, the savings were modest, as we only avoided expanding one extra state. In significantly larger trees, the savings should also be significantly higher.■

Algorithm 5: GetSuggestions(prefix,n)

```
input  : A prefix string prefix and a maximum number of suggestions n
output: The list of top n suggestions taking into account the suggestion weights
result ⟵ [];
Q ⟵ [];
state ⟵ getState(startState,prefix);
Q.add(state,state.total_count);
while Q.notEmpty() ∧ result.size < n do
    current ⟵ Q.top();
    if current.isState() then
        foreach suggestion k associated to current do
            Q.add(k,current.key_count[k]);
        foreach transition character c from current do
            Q.add(current.transition_state[c],current.transition_count[c]);
    else
        result.add(current);
return result;
```

The upper bound on the run-time of Algorithm 5 is $O(k \log k)$, where k is the total number of states and suggestions in the radix tree. This corresponds to the situation in which we have to visit all states before we can return a list of the top n suggestions. However, because we can stop expanding states as soon as we have retrieved enough suggestions, this upper bound is potentially over-pessimistic, depending on the weights associated to the supported strings. We will have much more to say about this in the following chapter.

Note also, that this is exactly how we would implement this algorithm for a prefix tree as well. The distinction between the two tree structures is irrelevant for Algorithm 5. Thus, given the expected state count reduction, this algorithm should be considerably faster on a radix tree. Although we paid for the compactness of our representation by having to design more complicated procedures when building the radix tree, there is no more price to pay when making suggestions, so this is where the extra effort should pay off.

3.5 Implementing error tolerance

In this section we will look at an alternative implementation of the GetState procedure (Algorithm 4), which will enable us to parse the supplied prefix with some degree of tolerance to error. Thus, we will offer an solution for meeting Optional Requirement 2, which we introduced in the first chapter. Because there may be multiple states that can be expanded for a given prefix given a certain tolerance to error, we will change the algorithm name to *GetStates* and its implicit return type is changed as well: the algorithm now returns a set of states, rather than a single state.

We implement error tolerant prefix matching for the radix tree using an approach based on the Levenshtein distance between two strings. In this respect it is similar to the approach presented in Chapter 2 for regular tries, but the implementation differs as a result of having to accommodate for edges labeled with strings, as opposed to single characters. This function can then simply be used instead of the call to *GetState* within the algorithm for suggestion making, in order to allow retrieving suggestions with some tolerance to error when parsing the supplied prefix string. This procedure is shown in Algorithm 6.

A regular FIFO queue Q is used, for storing states in the radix tree along with the *index* to which the $prefix$ has been consumed in reaching this state, and the (edit) $distance$ needed in order to achieve this, but only if the edit distance does not exceed the $distance$ parameter to the function. The queue initially consists of just the start state, with the associated prefix and index set to 0.

On each iteration of the while loop, we examine all transitions from the current state

Algorithm 6: GetStates(prefix,distance)

```
input  : A prefix string prefix, a value distance for the maximal tolerated edit distance
output: A set of active states that are viable candidates to be expanded for
        suggestion making
result ⟵ [];
Q ⟵ [];
Q.add(< $q_0$,0,0>);
while Q.notEmpty() do
    <q,q.distance,q.index> ⟵ Q.top();
    current_prefix = prefix[q.index..prefix.length];
    foreach label from q to state q' do
        match ⟵ transform(label,current_prefix, distance - q.distance);
        if m.size > 0 then
            result.add(q');
        match ⟵ transform(current_prefix,label, distance - q.distance);
        for m in match do
            if q.index + m.index + 1 ≥ prefix.length then
                result.add(q');
            else
                Q.add(<q',q.distance + m.distance, q.index + m.index+1>);
return result;
```

extracted from the head of the queue. First we try to transform any prefix of the label on that transition into what's left to parse, within the edit-distance limits. The function *transform* tries to achieve this. It returns a *match* object which is a list of all the pairs of minimum distance necessary to transform a prefix of the first parameter into the second and the index inside the first parameter that delimits this prefix, as long as the number of edits required for the transformation is no bigger than the maximum threshold provided as the third parameter. If the transformation can be done, we can safely consider that the prefix has been parsed and we can expand suggestions from the destination state of that edge.

Alternatively we try to transform the $current\_prefix$ to be parsed into the *label* leading from the current state, in order to be able to traverse that particular edge. Basically, we try to get *label* from any prefix of $current\_prefix$, using the minimal possible number of edits for that particular prefix. We make entries for all possibilities in Q as long as distance does not exceed the maximum threshold.

This can be seen as simulating the algorithm presented in Chapter 2 for error tolerant prefix matching on regular tries, with the difference that since some states have been merged in the radix tree, we simulate the addition of active nodes by making multiple entries for the same state with different index and distance values.

Example 3.8. For an example, consider again the radix tree in Figure 3.6, and consider parsing the (mistyped) string ”amstrdam”, with a maximum allowed edit distance of 1. The algorithm starts by placing the start state in the queue with its corresponding entries for the index to which the prefix was already parsed and the distance needed for that, both set to 0. When this is examined, we first try to transform the only label leaving the current state (”amste”) to the prefix (”amstrdam”). The match object in the pseudocode is used to store the different indices and the edit distance needed to achieve this for that index. In this case an edit distance 4 is the minimum that can be achieved for indices corresponding to ends of substrings ”amst” and ”amste”. Since this is not achieved within the maximum allowed edit distance, the match object will be empty and the state at the end of the edge labeled ”amste” is not stored as a final result. Next, we try to transform substrings of ”amstrdam” to ”amste” with the minimum possible edit distance. We can achieve this with edit distance 1 only for the prefixes ”amst” and ”amstr” and entries are made for both in the queue, with the requirement that we are able to parse the rest of the prefix within edit distance 0 starting from the state to which the edge labeled ”amste” leads. This is only possible by continuing from the substring ”amst”, and thus we have found a way to parse the string ”amstrdam” within edit distance 1, by traversing edges that, put together, spell ”amsterdam”.

Note that we do need to store all prefixes that can be expanded, as the shortest one is not always the best choice. Referring back to the previous example, if the supplied prefix had been ”amstwrdam”, ”amste” could again be obtained two ways within edit distance 1: from both ”amst” and ”amstw”. However, in this case, expanding the longer prefix is clearly the only choice that would allow us to parse the string ”amsterdam” within edit distance 1.

Moreover, storing entries for just the minimum overall distance in the *match* object is also insufficient. Consider parsing the string *abb* over two edges *xb* and *abb*. Trying to traverse the edge *xb* could be done with the empty string or with the prefix *a* within edit distance 2, and with *ab* within edit distance 1. But if we choose to traverse it using *ab*, we require edit distance at least 2 to traverse the remaining edge with *b*, and this would lead to a total edit distance of 3 needed to traverse both edges. On the other hand, choosing the empty prefix to traverse the first edge *xb*, despite requiring an edit distance of 2, leads to a lower edit distance overall, because then traversing the second edge using *abb* can be done within edit distance 0. ■

3.6 Implementing location bias

In this section we describe the algorithm that allows us to meet Optional Requirement 3: take into account the location of the user when making suggestions. We will see that the choices made in designing our algorithms so far allow us to support this functionality by implementing a small modification to the suggestion making algorithm that we introduced in Section 3.4.

Algorithm 7 shows how this can be achieved. This algorithm is an extension of Algorithm 5. The idea to implementing a location bias (relative to a latitude-longitude pair known as the *bias point*) is to apply a (downwards) scaling of the weights of the suggestions before they are added to the priority queue, depending on the distance between the bias point and the location of the suggestion. This is accomplished by multiplying the weight of the association with a value between 0 and 1. Since a priority queue is used to find suggestions by choosing the largest weight at any point in time, this means deferring retrieving suggestions that are far away from the bias point, and giving priority to expanding other, more promising, states. The formula that we use for the scaling factor is $\alpha = \frac{1}{1+distance}$, where the distance is any measure of the distance between the location of the bias point and the location of the to-be-suggested location.

Example 3.9. Consider again the radix tree constructed in Figure 3.4 and the suggestion making process for the supplied prefix 'be'. The queue would be initialized to the state at the end of the edge labeled 'berlin' and an associated weight of 30. As this state is expanded, both states at the ends of strings '$potsdamer$platz' and 'er$straße$munich' would be added, with associated weights 10 and 20 respectively. The latter of the two would be the first to be expanded. Consider that the query is coming from a location having latitude approximately 52.509 and a longitude of approximately 13.381. That means that the user would be located somewhere in Berlin. This means that the distance to Berliner Straße, Munich is quite large. Suppose this would create a scaling factor $\alpha = 0.2$, according to some distance measure. This would mean that suggestion 2 is added with weight 4, and hence would no longer be the first to be extracted from the queue during the next iteration, as in Example 3.7. The state holding suggestion 1 with associated weight 10 would be expanded next, and since the distance is very small, even after scaling we would add suggestion 1 to the queue with a weight of about 10, which means that on the next iteration, it would be the first to be retrieved from the queue, as the top suggestion, ahead of the suggestion for Berliner Straße, Munich. ■

In order to see whether the algorithm works correctly, we have to first make precise what its correct behavior should be. For each invocation of the algorithm, we are given a lat-long pair which is fixed for that particular instance. The algorithm then should retrieve a ranked list of the top *n* suggestions, having the highest rescaled weights. That is, by specifying a fixed lat-long pair to represent a bias point, we are essentially creating a new

Algorithm 7: GetSuggestions(prefix, n, lat, long)

```
input  : A prefix string prefix, a maximum number of suggestions n, the user latitude,
         lat, and the user longitude, long
output: The list of top n suggestions taking into account the suggestion weights and
         their proximity to the bias point
result ⟵ [];
Q ⟵ [];
state ⟵ getState(startState,prefix);
Q.add(state,state.total_count);
while Q.notEmpty() ∧ result.size < n do
    current ⟵ Q.top();
    if currentState.isState() then
        foreach suggestion k associated to current do
            $\alpha \longleftarrow \frac{1}{1+distance((lat,lon),(k.lat,k.long))}$;
            Q.add(k,$\alpha\cdot$ current.key_count[k]);
        foreach transition character c from current do
            Q.add(current.transition_state[c],current.transition_count[c]);
    else
        result.add(current);
return result;
```

problem instance for Algorithm 5, with modified weights associated to each suggestion. However, in Algorithm 7, we are only rescaling before adding suggestions to the priority queue, and weights of states are not modified, since we cannot know which (or even how many suggestions) are associated to the tree rooted in a certain state without exploring the whole tree. Nonetheless, the corollary to the following theorem shows that Algorithm 7 will still allow us to find the correct result.

Theorem 3.4. *When a suggestion is retrieved from the priority queue by Algorithm 7, its weight is the highest of the rescaled weights of suggestions in the queue, including those hidden in not yet expanded states.*

Proof. Let suggestion k be a suggestion that is being retrieved from the queue by the algorithm. By the priority queue property, it is clear that the rescaled weight of k, which we will denote $\alpha_k \cdot w_k$, is higher than the weight of any other suggestion currently in the queue (which must also have been rescaled). What we need to show is that the rescaled weight of k is also higher than the rescaled weight of any suggestion that can be retrieved by expanding any state currently in the queue. Consider an arbitrary state q with weight w_q, which is in the priority queue when k is retrieved. By the priority queue property, we have $\alpha_k \cdot w_k \geq w_q$ (1). Again, recall that by the way the radix tree is constructed, for any suggestion k' having weight $w_{k'}$ that we may obtain by expanding q we will observe that $w_q \geq w_{k'}$ (2). Let $\alpha_{k'}$ be the scaling factor associated to suggestion k', calculated based on its proximity to the bias point. From the definition of the scaling factor, we know that

$0 < \alpha_{k'} \leq 1$, and hence $w_{k'} \geq \alpha_{k'} \cdot w_{k'}$ (3). Putting relations (1), (2) and (3) together, we get $\alpha_k \cdot w_k \geq w_q \geq w_{k'} \geq \alpha_{k'} \cdot w_{k'}$. Thus, the rescaled weight of suggestion k is guaranteed to be no lower than the rescaled weight of suggestion k', so we can retrieve k from the queue ahead of k'. Since state q and suggestion k' associated to it were chosen arbitrarily, we can conclude that suggestion k can be retrieved ahead of all suggestions currently in the queue, as well as the suggestions that have not yet been encountered, despite the fact that we have not yet applied rescaling to find their actual weight given the bias point provided as a parameter. □

Theorem 3.4, coupled with the fact that the priority queue is initialized to contain the state that hides the subtree of all suggestions that can be made for a given prefix, yields the following immediate corollary.

Corollary 3.1. *Algorithm 7 correctly determines the list of the top n suggestions that can be made for a given prefix, based on the rescaled weights according to the suggestions' proximity to the bias point.*

We end by presenting a variation to the biased suggestion making algorithm. We first motivate it through an example. The biased suggestion making algorithm seems to be a good solution for mobile applications, where the user is located at a certain latitude and longitude and is likely to be interested in search results in their immediate vicinity. But consider an application such as the online RoutePlanner (`http://routes.tomtom.com/`), which may also want to offer biased suggestions, depending on the location that the user is currently looking at. A typical RoutePlanner screen is shown in Figure 3.7.

Here, the user is looking at a nearly complete map of Europe, and is free to further zoom in or out. A good choice for a bias point in this case seems to be the center of the image, located somewhere in the South of Germany in this particular example. However, since the user is looking at a map of Europe, it does not seem right to make ”Langenau, Germany” a much more likely suggestion than ”London, United Kingdom” for prefix ”L”, for example, simply because it is located much closer to the center of the currently visible portion of the map. In other words, we should not ignore the fact that the user is currently looking at a very large portion of the map, and is equally likely to be interested in any part of it (but perhaps less likely to be interested in other regions of the map that are not visible). For such applications, the biased suggestion making procedure can be modified, as shown in Algorithm 8.

The idea is simply to restrict applying the scaling to just the suggestions outside the visible region of the map. A radius is supplied for this purpose, representing the approximate distance to each edge of the visible region of the map. Hence, suggestions maintain their weights inside the supplied radius. For our previous examples, suggestions in the visible part of Europe are made, according to their original weights. Other suggestions outside the visible region of the map are only made depending on their relevance for the entered prefix and their proximity to the visible region of the map.

Building on this idea, it is easy to adapt the algorithm to e.g. use a bounding box rather than a radius if this is a better choice for the application that makes use of this functionality.

Figure 3.7: Partial view of the European map on www.routes.tomtom.com

3.7 Summary

Having presented the algorithms to support all the functionality that we set out to include in our auto-completion system, in the next chapter we will experimentally test these algorithms and the assumptions and hypotheses made in developing them. We will also focus on the issues that are important for the applicability of these algorithms in practice and try to derive useful guidelines for their use based on the experimental results.

Algorithm 8: GetSuggestions(prefix,lat,long,r)

```
input  : A prefix string $prefix$, a maximum number of suggestions $n$, the user latitude,
         $lat$, and the user longitude, $long$, and a radius $r$
output: A set of active states to be expanded
result ⟵ [];
Q ⟵ [];
state ⟵ getState(startState,prefix);
Q.add(state,state.total_count);
while $Q.notEmpty() \wedge result.size < n$ do
    current ⟵ Q.top();
    if $currentState.isState()$ then
        foreach $suggestion\ k\ associated\ to\ current$ do
            if $distance((lat,lon),(k.lat,k.long)) > r$ then
                $\alpha \longleftarrow \frac{1}{1+distance((lat,lon),(k.lat,k.long))-r}$;
                Q.add(k,$\alpha\cdot$ current.key_count[k]);
            else
                Q.add(k,current.key_count[k]);
        foreach $transition\ character\ c\ from\ current$ do
            Q.add(current.transition_state[c],current.transition_count[c]);
    else
        result.add(current);
return $result$;
```

Chapter 4

Evaluation of the proposed algorithms

In this chapter we will experimentally evaluate the algorithms described in the previous chapter, with the goal of testing the hypotheses made while making different choices in developing these algorithms. Where applicable, we will also evaluate different choices that can be made when using these algorithms and how these choices impact performance.

To test the proposed algorithms we used a JAVA implementation. This choice was made primarily to facilitate the inclusion of the new functionality within existing projects within TomTom, which also run JAVA. The experiments were run on a machine with an Intel® Core ™ i7-2620M processor (2.7 GHz) and 8 GB of RAM. Two hyperthreading enabled processor cores were available, but all our implementations will be sequential and will run on a single processor core, in a single thread, unless otherwise stated.

Next we describe the structure of this chapter. Section 4.1 is the longest section and tests the radix tree implementation starting from verifying the hypotheses that motivated the choice of a radix tree over a prefix tree. Because this section aims to test all aspects related to the way we defined the Probabilistic Radix Tree, in addition to verifying how the two main requirements introduced in Chapter 1 are met, we will also verify here how Optional Requirement 1 is met, as this is strictly tied to the radix tree structure. Motivated by some curious findings about the influence of the distribution of suggestion weights on performance, we will also run multiple tests from which we can learn more about how to configure this distribution in order to get the best performance from the auto-completion system.

Section 4.2 is dedicated to testing the performance of the error tolerant prefix matching, which aims to meet Optional Requirement 2, and drawing conclusions about its proper use in the auto-completion system. In Section 4.3 we test the performance of the system when making location biased suggestions in order to meet Optional Requirement 3. In Section 4.4 we will motivate and run tests regarding the suitability of implementing the described algorithms in a distributed environment, again deriving useful tips for getting good performance from such an implementation.

4.1 Radix-tree implementation

This section aims to answer a series of practical questions regarding the choices made in developing the algorithms presented, and more precisely, those questions that have to do with the actual radix tree structure. The questions that we will look at are as follows:

- Does choosing a radix tree implementation offer the expected benefits in terms of memory usage and run-time performance, compared to a prefix tree?
- Does the use of the radix tree for key matching in an associative data structure affect performance, when we need to support queries where the terms are out of order, with respect to the standard hierarchical format?
- What is the influence of the distribution of weights on the performance of the auto-completion system?

In conducting the first experiments, we use a list of street-level and city-level addresses corresponding to the largest 140 cities and towns in Germany, and all street level addresses in them, for a total of approximately 205,000 addresses.

Figures 4.1 and 4.2 show the different scaling behavior of the number of states needed by the prefix tree and radix tree implementations as a function of strings and addresses supported, respectively. The distinction is made because each address has several strings associated to it, in order to support substring matching. Nevertheless, the two graphs are very similar. As expected, given the nature of our supported queries, the number of states required by the radix tree implementation is significantly smaller.

However, this reduction in state count does come at the cost of having to store strings instead of characters as edge labels. To maintain these labels, each state will require more memory in a radix tree compared to a prefix tree. So despite the big difference in state counts, we should verify that the overall memory requirements are actually reduced. Figure 4.3 shows the outcome of this test. Because it is not trivial to evaluate the memory requirements of a JAVA program, these test results are a bit coarser, but conclusive, nonetheless.

The tests run so far make the point that using a radix tree instead of a prefix tree leads to a great reduction in the number of states and in the amount of memory used. Compelling as the results may be, since we are building a real-time auto-completion system, we still need to make sure that we are not paying any considerable penalty for this compact representation. First we address the perhaps less critical issue of performance during building the radix tree. We first raised this issue in the previous chapter, when describing the procedure for adding a new term to the Probabilistic Radix Tree. The procedure is far more complex compared to the equivalent procedure for a prefix tree, but we did argue that the asymptotic complexity of the algorithm as a function of the length of the term added does not change, but the constant time operations involved could have more of an impact, especially since the length of all terms can be assumed to be bounded by some reasonable constant itself. Nevertheless, we don't expect the run-time in the case of the radix tree to suffer too much.

Note also that given the state reduction, there are fewer states to be updated along the path of adding each new term to the radix tree. Indeed, Figure 4.4 shows that the run-time for this procedure follows similar trends for the two structures.

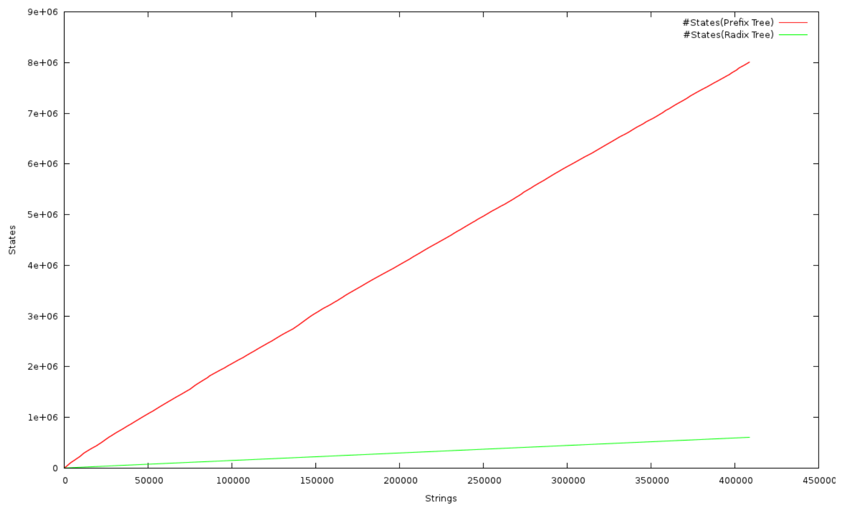


Figure 4.1: Comparison of the scalability in terms of state counts vs. number of supported strings for Prefix Trees and Radix Trees

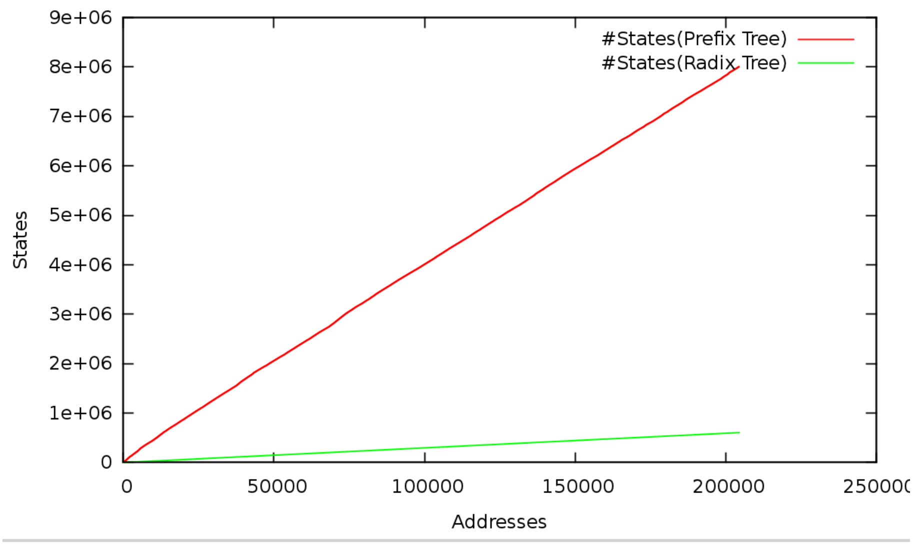


Figure 4.2: Comparison of the scalability in terms of state counts vs. number of supported addresses for Prefix Trees and Radix Trees

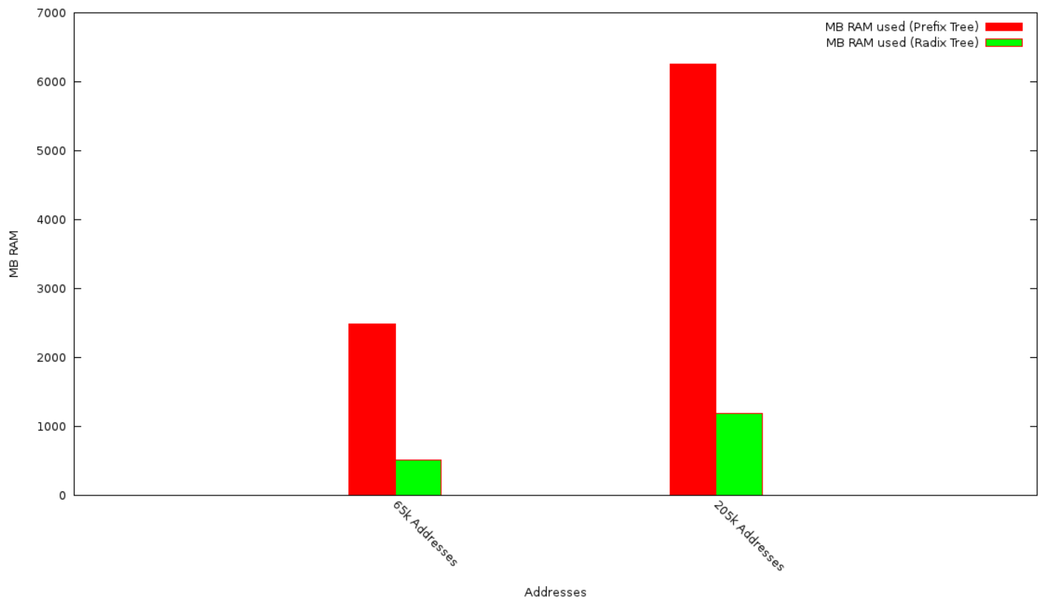


Figure 4.3: Comparison of the memory requirements vs. number of addresses for Prefix Trees and Radix Trees

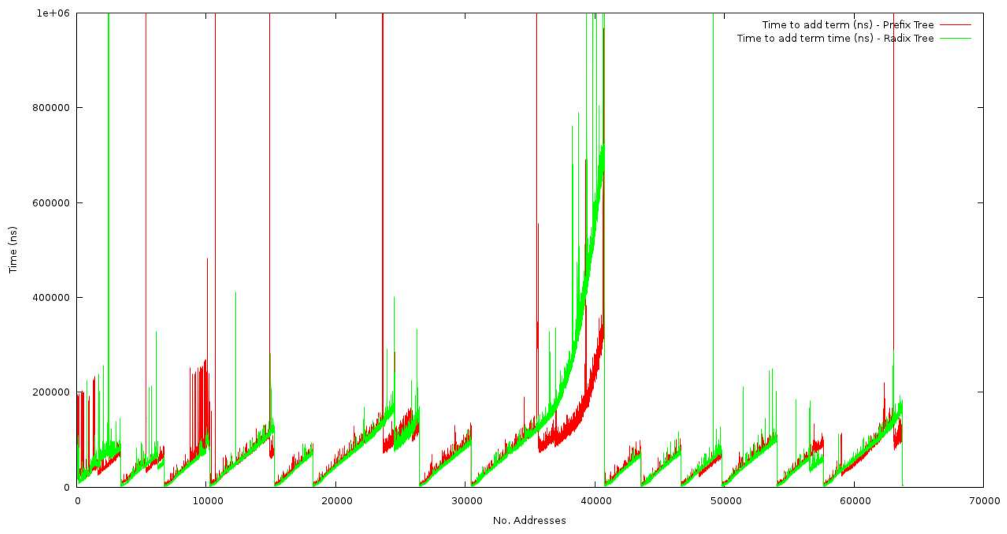


Figure 4.4: Scalability of run-times for adding terms to Prefix Trees and Radix Trees vs. the number of addresses already supported by the tree

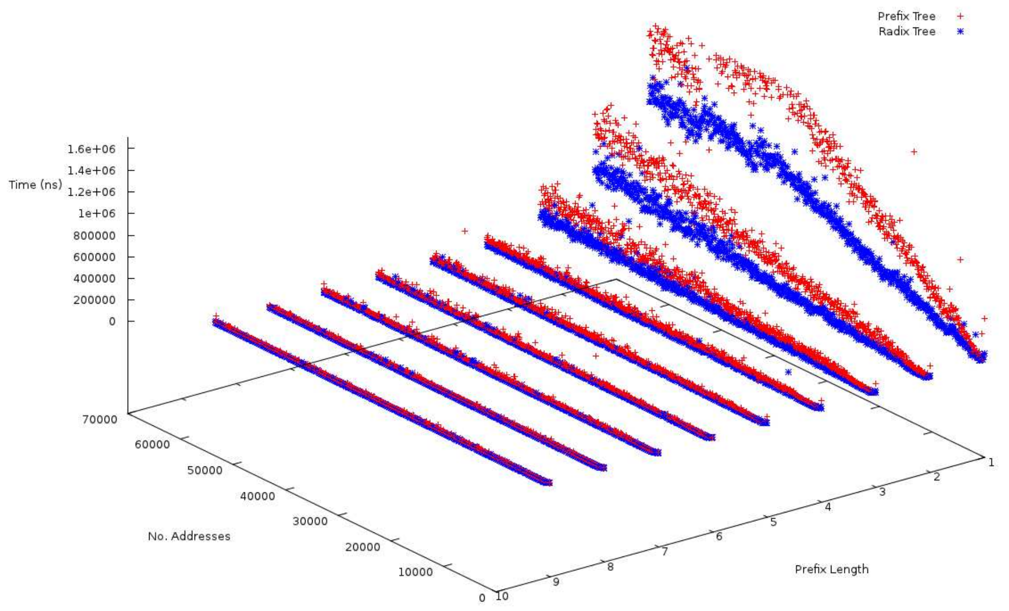


Figure 4.5: Scalability of the run-times for the suggestion making process for Prefix Trees and Radix Trees for varying prefix lengths and number of addresses

Remark 4.1. Note that the measured run-times are in the nanoseconds range. We believe that the sawtooth trend observed for the measured run-times is owed to external factors, such as buffering and caching, since the measurements were made at intermediate points in the process of reading data from disk and adding terms to the tree. We do not see any reason for this pattern to arise as a result of the algorithm logic. What we are interested in, though, is that the run-times for the two data structures are very similar, so we can conclude that we did not create performance issues as far as the operation of adding supported terms is concerned, by opting for a radix tree implementation. ■

Next, we turn out attention to the far more important performance issue: the run-time required for making suggestions. We compare the prefix tree against the radix tree on this performance measure. When describing the suggestion making algorithm in the previous chapter we mentioned that the algorithm requires no adapting when moving from prefix trees to radix trees. Coupled with the significant state count reduction noticed in radix trees compared to prefix trees, we should expect the radix tree to offer significant improvements over the prefix tree. Figure 4.5 shows the outcome of verifying this. The figure shows the run-time in nanoseconds for making suggestions for prefixes of different lengths, on trees supporting different numbers of addresses. All prefixes are guaranteed to generate some number of suggestions, so it is never the case that the algorithm has no state to expand in order to produce a list of suggestions.

As expected, the radix tree is much faster, and the difference is significant in the case

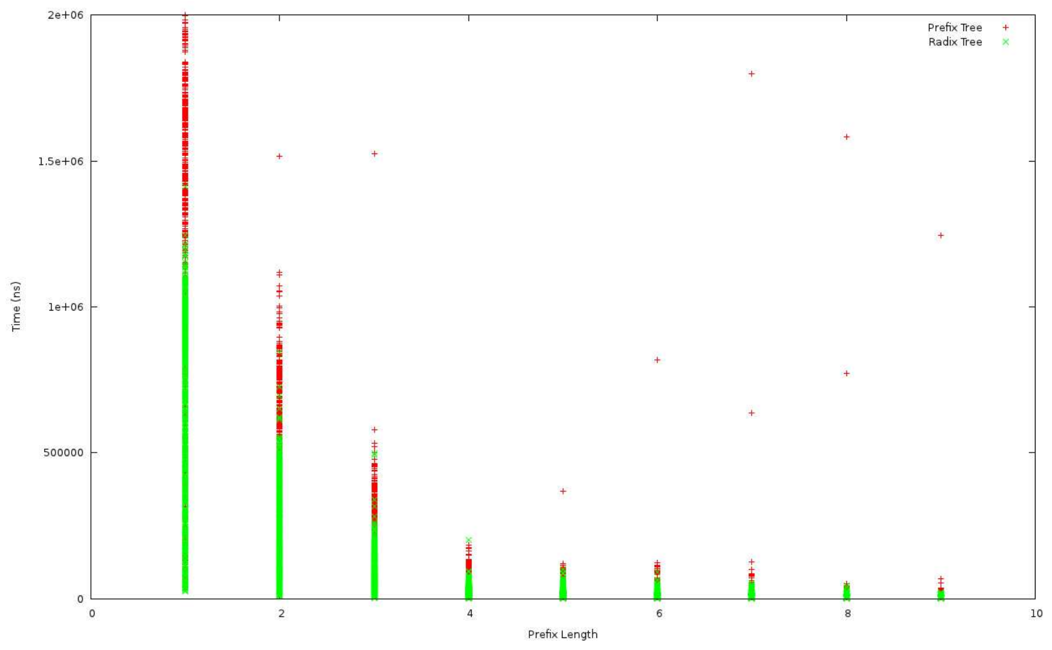


Figure 4.6: The impact of the prefix length on the suggestion making run-time for Prefix Trees and Radix Trees

of small prefix lengths, where expanding states means exploring a large number of possibilities before completing a list of suggestions. We also take this opportunity to point out that the run-times seen so far fit comfortably within the definition of real-time performance as described in Chapter 2, as in very few tests the 1 ms barrier is passed, and that usually just for the prefix tree, on short prefixes.

Figures 4.6 and 4.7 project figure 4.5 on the prefix length and the address count planes respectively, in order to clarify the run-time behavior relative to each of these parameters.

We find the tests conducted so far sufficient in order to conclude that radix trees offer an overall better approach, so in the following we will only concentrate on them, which will also allow us to move to larger data sets. Specifically, in the following experiments we will use a list of the largest 256 cities in the United States and the 1,115,291 street level addresses in them.

We will next look at the penalty that we pay for being able to match terms of an address in an out-of-order fashion. For this purpose we look at the time required to make suggestion for prefixes one character in length for both the case when substring matching is supported, as well as when it isn't. The choice of this minimum prefix length corresponds to a worst-case scenario, because as we have seen in the previous tests (and not surprisingly), making suggestions for these short prefixes takes the longest. The results are plotted in Figure 4.8.

Again, we attribute the spikes that can be observed to external factors, as the run-time ranges we are dealing with are very small, and we focus on the general trend, which seems to be similar for the two types of matching.

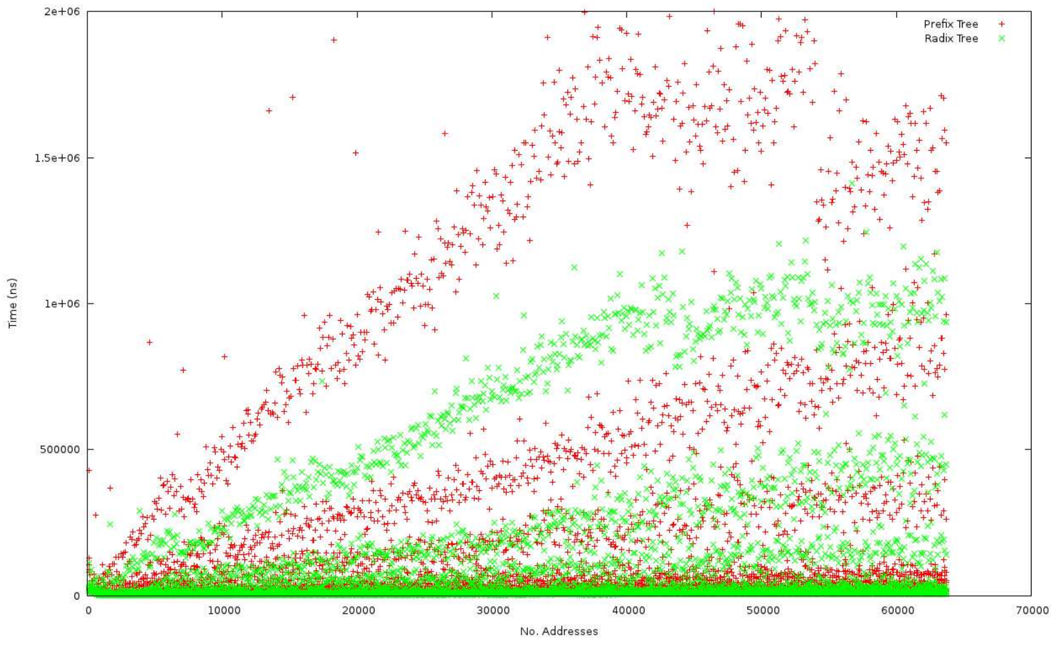


Figure 4.7: The impact of the number of supported addresses on the suggestion making run-time for Prefix Trees and Radix Trees

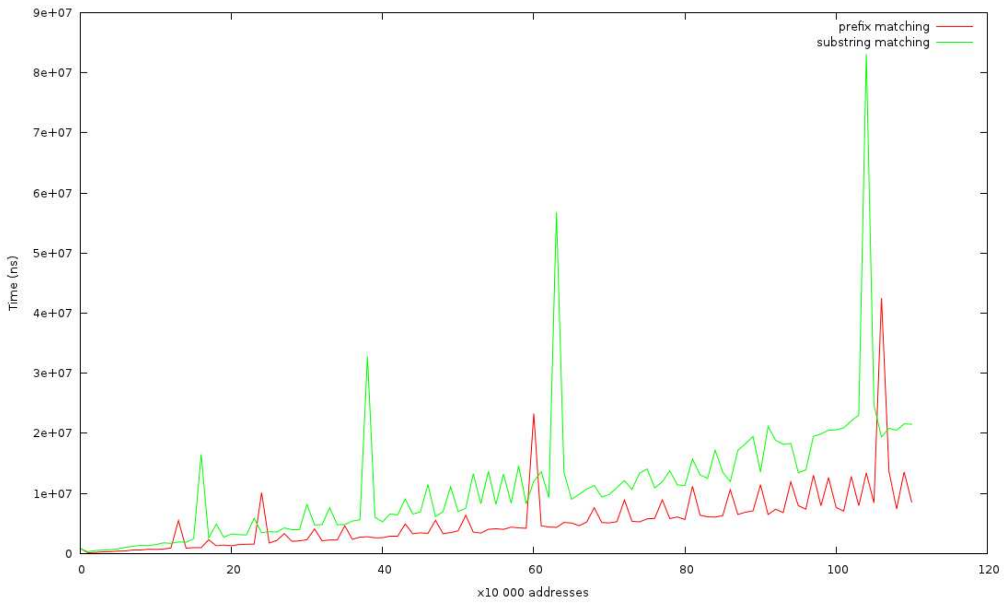


Figure 4.8: Scalability of suggestion making run-times when substring matching is enabled or disabled

There are in this case three strings that are added to the radix tree for each individual address. Thus, for an address of the form ”*street, city, state, country*”, we also add the strings corresponding to the hierarchies ”*city, street, state, country*” and ”*city, state, street, country*”. As Figure 4.8 shows, the price to pay for this flexibility is within a constant factor and the suggestion making run-time exhibits scalability very close to the one seen for the version without substring matching.

The experiments run so far have used a very simple distribution, in which we have three types of cities (capital, large city, medium city), and all cities and streets within them had a weight that depends only on this type. More specifically, cities have weights 5,000, 4,000 and 3,000 depending on type, and the streets in them have weights 500, 400 and 300 respectively. Intuitively, this corresponds to a near worst case scenario, because of the limited possible values of weights attributed to each association. That is, assuming we have an association *k* in the priority queue, its weight will be lower than any state that can be expanded, at least to one association of similar type. It is beyond the scope of this thesis project to determine ways of generating distributions that produce more relevant suggestions, but we want to test the hypothesis that doing so will also result in improved performance for the suggestion making algorithm.

To do this, we simplify our distribution further: all cities will have weights 1,000, and all streets will have weights 100. We will test this against a uniform distribution of weights in the much wider range 1 - 10,000, across all cities and streets. We expect that given a certain prefix, more diversity will lead to a higher chance of expanding suggestions that hide portions of the tree that correspond to lower weight suggestions, thus speeding up the suggestion making process. The results of this experiment are shown in Figure 4.9 and seem very surprising at first. The very simple distribution is performing much better than the distribution with much greater diversity. The results (which were consistent over multiple runs) clearly contradict our hypothesis in its simple form stated above.

We will address this apparently very strange behavior soon, but first, we would like to address the hypothesis that increased diversity has benefits, by creating a test environment that more directly addresses this aspect. The reason for the insistence on at least establishing that greater diversity does not incur a performance penalty in itself is that diversity is a requirement, and it is strongly recommended that it is used for reasons other than performance, such as offering a detailed ranking between the different suggestions that can be made.

Figure 4.10 shows a comparison between the run-time required to make suggestions when the range of weights is uniformly distributed across the interval 1 to 100 compared to the interval 1 to 10,000.

Once again, prefixes of length 1 are used, to simulate a worst case scenario. It should also be noted that the run-times we are dealing with now have significantly increased, and we even observe run-times that exceed 100 ms. This time, the behavior is as expected, and indeed the tree built with addresses the weights of which exhibit greater diversity can be used to produce better run-times. This is in line with the intuition that having a greater diversity creates a higher potential for having states that are not worth expanding before making some higher weighted suggestion. However, this difference is far less impressive compared to the benefit that we noticed when using the very simple distribution which we

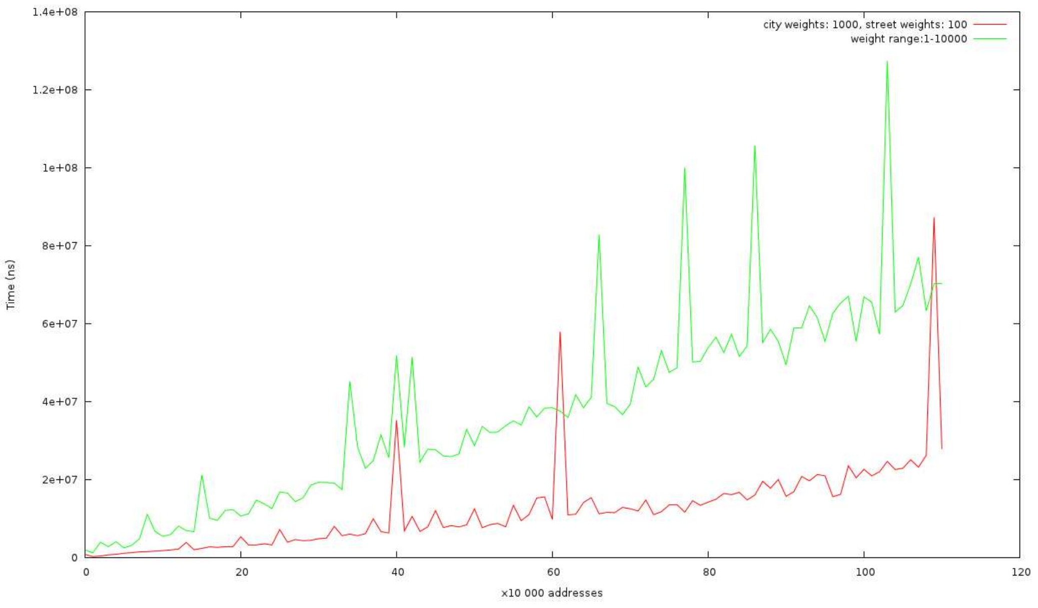


Figure 4.9: Scalability of suggestion making run-times for different distributions of weights

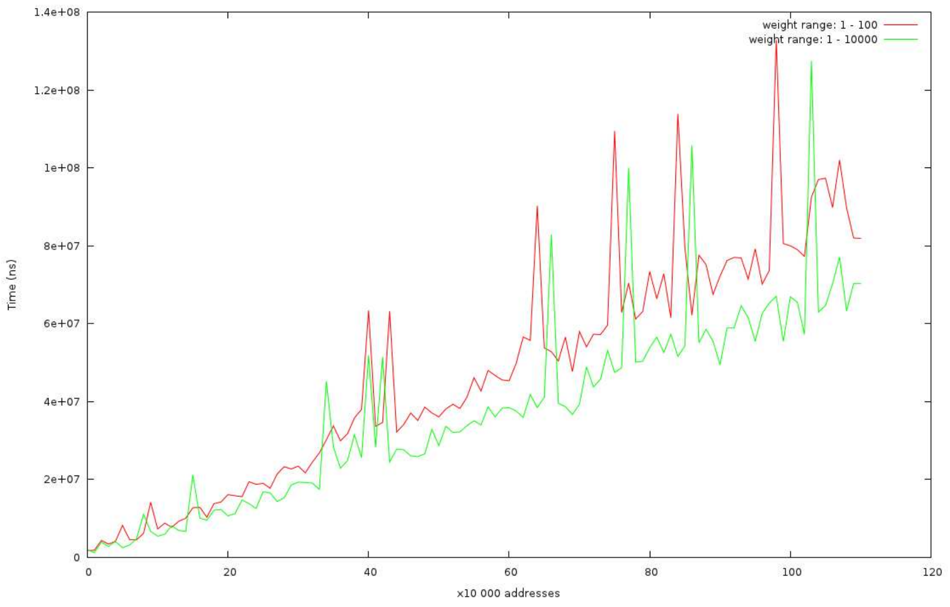


Figure 4.10: Scalability of suggestion making run-times for different distributions of weights

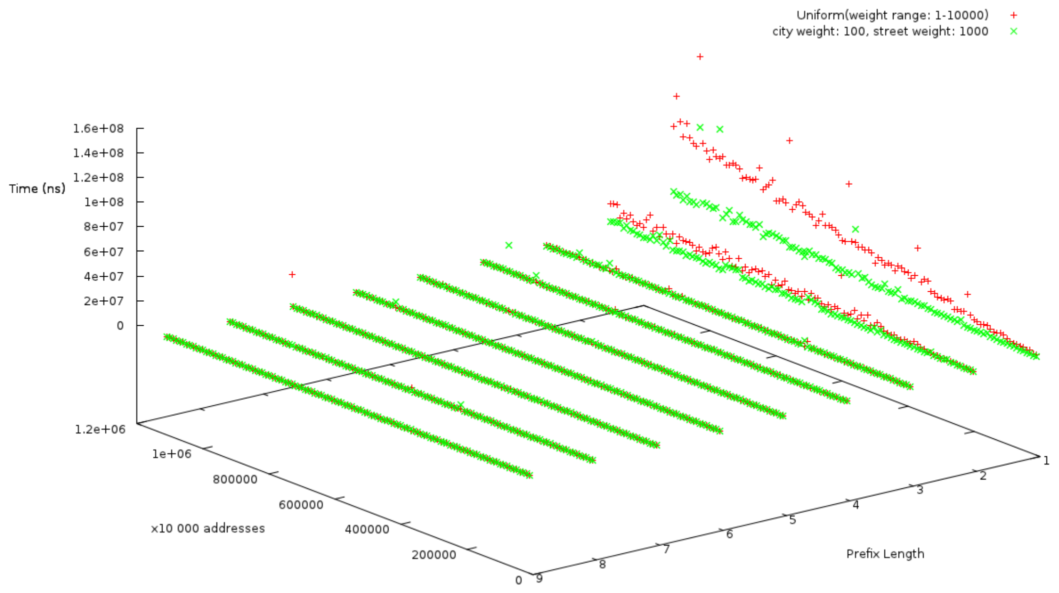


Figure 4.11: Scalability of the run-times for the suggestion making process for varying prefix lengths and number of addresses, using different weight distributions

mentioned earlier (with all city weights 1,000 and all street weights of 100).

Returning to the example in Figure 4.9, we would like to get to the bottom of why such a simple distribution performs so well. Clearly the number of states expanded by the simple distribution is much smaller than the corresponding number in the case of the uniform distribution. According to the logic of the suggestion making algorithm, this can only happen as a result of the fact that many suggestions outweigh states that could be expanded which hide lower weighted suggestions. Recalling that we tested against the one letter prefix worst case scenario, we hypothesize that as there are plenty of city candidates to choose from for most one-letter prefixes, it is these suggestions that are responsible for hiding enough street level suggestions to reduce the search space sufficiently to result in the behavior seen in Figure 4.9. This will be interesting to verify. Note that given our choice of weights, only 10 street suggestions can be hidden by a city suggestion, because if a state leads to more than 10 street suggestions, that state will have to be expanded before any suggestion for a city can be added to the results set, as the weights of cities are exactly 10 times greater than those of streets. For instance, states that can be reached by parsing prefixes like '5th' or 'West' will surely be expanded as well, given the large number of streets with such prefixes.

In order to retest, we introduce an extra parameter: the prefix length. Figure 4.11 shows the results: indeed, this significant difference is only noticeable for small prefixes of length 2 and especially 1 (this was verified by zooming in on smaller ranges of the run-time axis for different prefix lengths). These experiments show the importance of having a class of

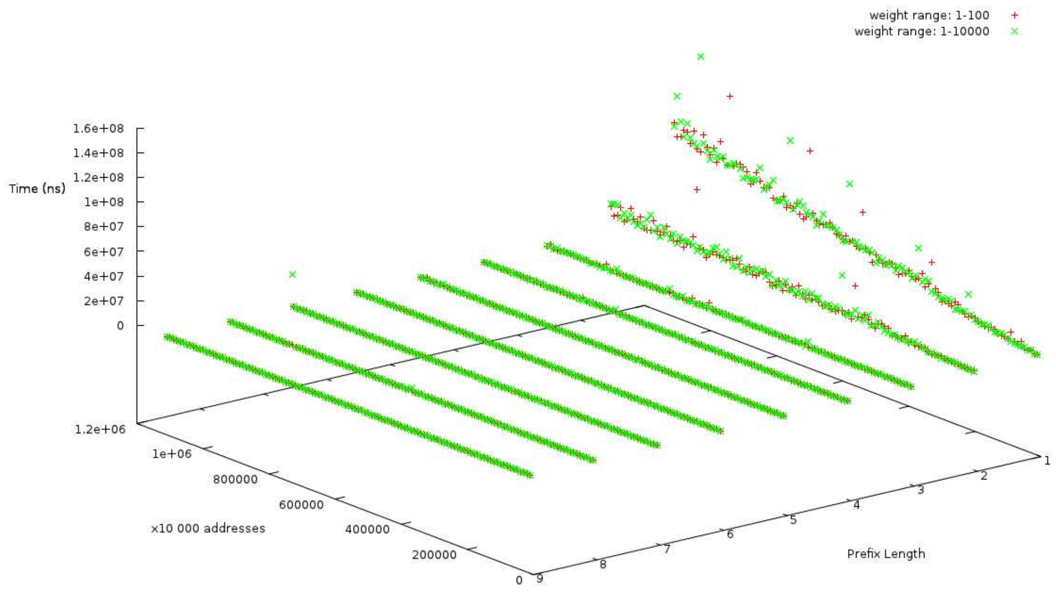


Figure 4.12: Scalability of the run-times for the suggestion making process for varying prefix lengths and number of addresses, using different weight distributions

a relatively small number of high-weight suggestions in allowing the system to cope with short prefixes.

This leads to a very interesting conclusion. All the experiments run so far show that no special tricks need to be added to the implementation to allow the algorithms to scale nicely with address counts, even when dealing with very short prefixes, as long as the data sample used when building the radix tree has the right kind of distribution of weights.

Thus, the distribution of weights over suggestions should preferably be structured in different levels, containing at least one level of few high-weight suggestions representing, say, country names or city names, in order to deal with very short prefixes. Within the different levels, a high diversity is recommended, but primarily in order to implement preferences among different same-level suggestions. A high diversity within different levels is less important for the run-time performance of suggestion making, since it makes little difference, as Figure 4.12 shows.

However, as long as the distinction between the different levels is clear enough, it is probably highly desirable to have some diversity within each level, in order to make suggestions that are properly ranked according to their relevance for the supplied prefix. It will be left to future work to find ways of generating such distributions, but our experiments underline the necessity to make sure that such distributions account for short prefixes, as described in this section.

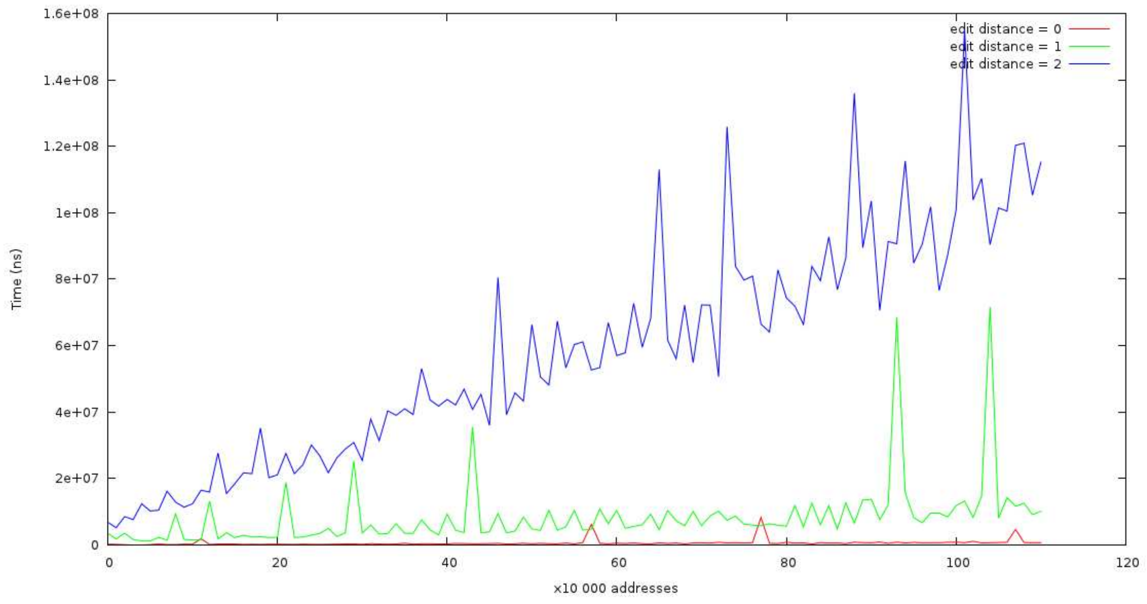


Figure 4.13: Scalability of the run-times for the suggestion making process for varying levels of error tolerance of prefix matching

4.2 Error tolerance support

In this section we will look at the run-time penalty for error-tolerant prefix matching. Figure 4.13 shows the difference in run-times for retrieving the top suggestions using exact prefix matching versus using a maximum edit distance threshold of 1 or 2 for parsing prefixes of length 4, which have been modified from exact matching prefixes, within the corresponding distance thresholds. We opted for this prefix length, as mistyping a prefix of length 1 or 2 cannot be interpreted as such, since the result would most likely be a prefix for some other possible suggestion. Moreover, we are only interested here in how different matching strategies compare to each other.

As the figure suggests, allowing for prefix matching that is tolerant to errors incurs significant overhead. The difference is so big, that it makes sense to try matching with different levels of tolerance in succession, i.e. try exact matching, and in case no states are retrieved, try matching with a maximum allowed edit distance of 1, and in case of failure again try matching, this time with a maximum allowed edit distance of 2.

This is a useful strategy, since for the very short prefixes (which also require the largest amount of time for making suggestions based on them), it is very unlikely that exact matching will produce no results, and as the prefix length increases, the run-times should drop for all kinds of matching. It is also perhaps fair to assume that the user is not mistyping a query, before suggesting a correction, as long as there are suggestions that can be made for the prefix as it is given by the user.

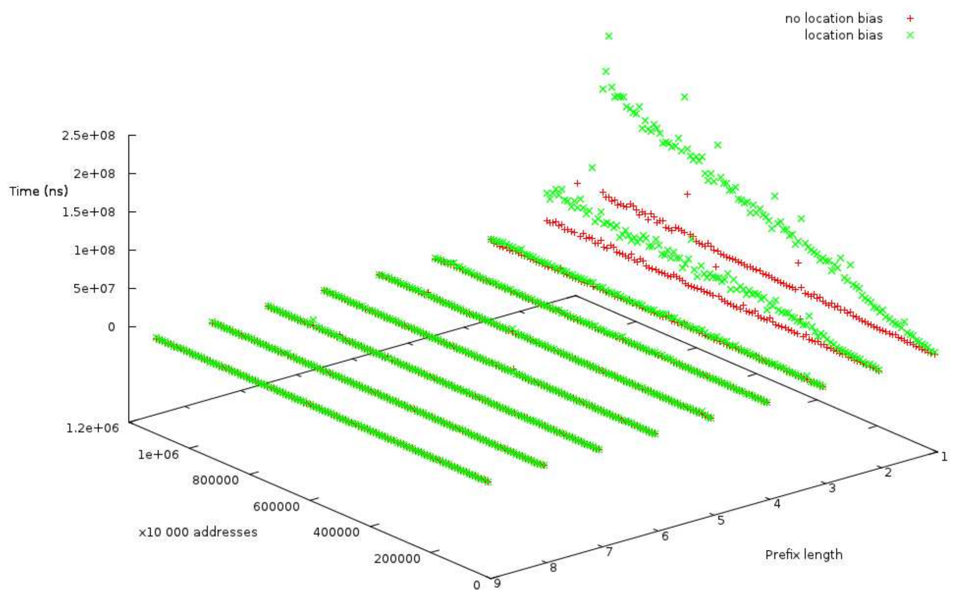


Figure 4.14: Scalability of the run-times for the suggestion making process for varying prefix lengths and number of addresses, with or without location bias support

4.3 Location bias impact

In this section we test the impact of implementing a location bias for making suggestions. For this purpose we initially used a distribution of weights as follows: cities have weights 3,000, 4,000 and 5,000 (depending on the city type, where only the capital city has weight 5,000) and the corresponding streets in them have weights 300, 400 and 500 respectively. The results of this experiment are summarized in Figure 4.14, for different prefix lengths.

We notice here that adding a location bias creates a fairly high overhead for short prefixes, with run-times reaching nearly 250ms for prefixes of length 1 with many possible suggestions to choose from, due to the rescaling that is done. However, this behavior can also be fixed with a better choice of a weighting scheme, as shown in Figure 4.15. In this case we found it useful to make a separation between streets in large cities and small cities. Hence, cities had weights 3,000,000, 4,000,000 and 5,000,000, whereas streets in large cities had weights 40,000 and 50,000, and streets in smaller cities had much smaller weights: 3,000.

These values were chosen somewhat arbitrarily, simply making use of the intuition that due to the rescaling, the advantage of having separate levels of suggestion relevance was lost. The choice of the best weighting scheme should be established experimentally, taking into consideration the suggestions to be made and how they can be divided into levels of relevance. This depends on the specific application that makes use of the auto-completion

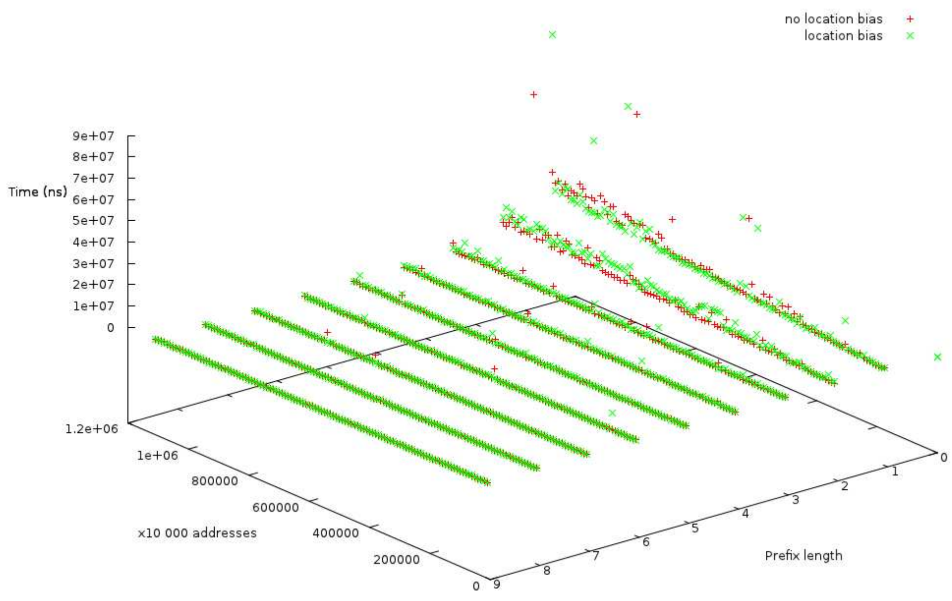


Figure 4.15: Scalability of the run-times for the suggestion making process for varying prefix lengths and number of addresses, with or without location bias support

system, and which has the responsibility of providing the data regarding the suggestions to be made and their weights.

We furthermore note that the range of distances plays a role here, i.e. in choosing a weighting scheme, it is important to get some hints from observing how high the distance can get, and equivalently how low the α factor (see Algorithm 7) will scale the suggestions to be made. This will depend on the specific distance measure being used and the unit in which it is calculated. What this experiment shows, however, is that it is easy to get good performance from the auto-completion system by using a weighting scheme that takes little effort and little imagination to come up with.

4.4 Distributing work

In this section we will look at ways of distributing the task of suggestion making among a number of computers. The need to explore this option arises from making the observation that the implementation in its current form requires a fairly large amount of memory. While this should be addressed as a problem in its own right, in an attempt to make use of as little memory as necessary, depending on the production environment used, it may eventually still be necessary to resort to using multiple computers for suggestion making.

The first test pits the straightforward implementation against an implementation distributing the load evenly across 2 and 4 trees respectively, with each tree attributed its own

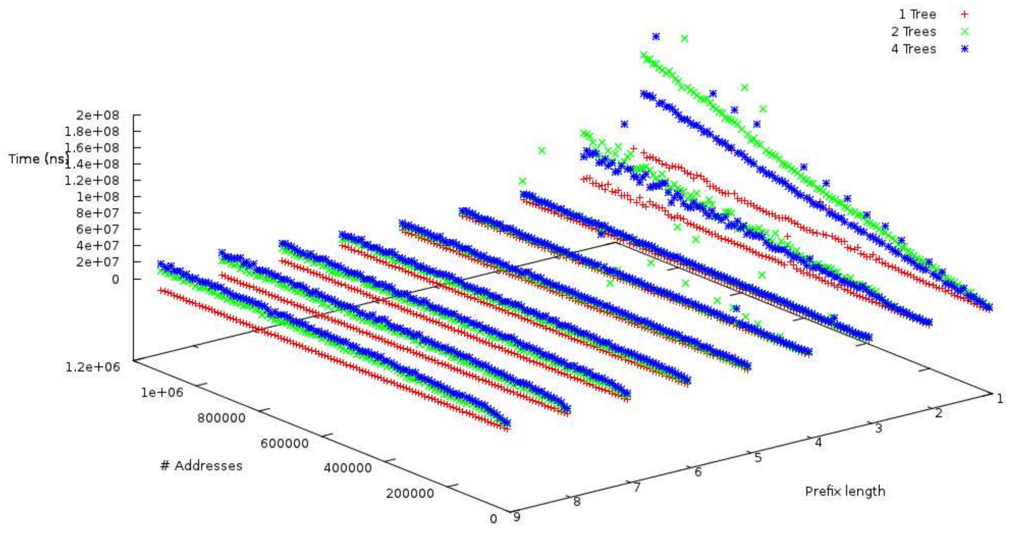


Figure 4.16: Scalability of the run-times for the suggestion making process for varying prefix lengths and number of addresses with different numbers of Trees working in parallel

thread for making suggestions. Hence the trees are queried in parallel (in as much as the system allows) and the results of all of them are merged in order to get a final ranked list of suggestions to return to the user. The overhead of setting up different threads to query trees is only added to the 2 and 4 tree implementation. The results (Figure 4.16) show that this has an impact, but one that scales nicely with the size of trees, but only for long prefixes. In the case of short prefixes, however, the scalability is much worse.

We expect that this bad scalability for short prefixes is not due to the overhead of distributing the work across multiple cores and merging the results, because such overhead should also be apparent for longer prefixes. But the scalability there is much better. This is most likely again caused by the distribution of supported suggestions.

For this experiment all trees were formed by randomly distributing addresses to trees, and thus all trees should exhibit similar weight distributions, but with a smaller number of suggestions for each possible weight in the case of each tree. Note that although the suggestions were distributed among several trees, we are still trying to get the total number of suggestions from each tree in order to merge them into a final result. All this means that each tree now has a smaller number of highly ranked strings from which to choose the same number of top suggestions to be made. A way to address this is to widen the gap between top level suggestions and lower priority suggestions. Figure 4.17 shows the effects of doing this only for when 2 trees are used, and Figure 4.18 shows the effects of doing this on 4 trees as well.

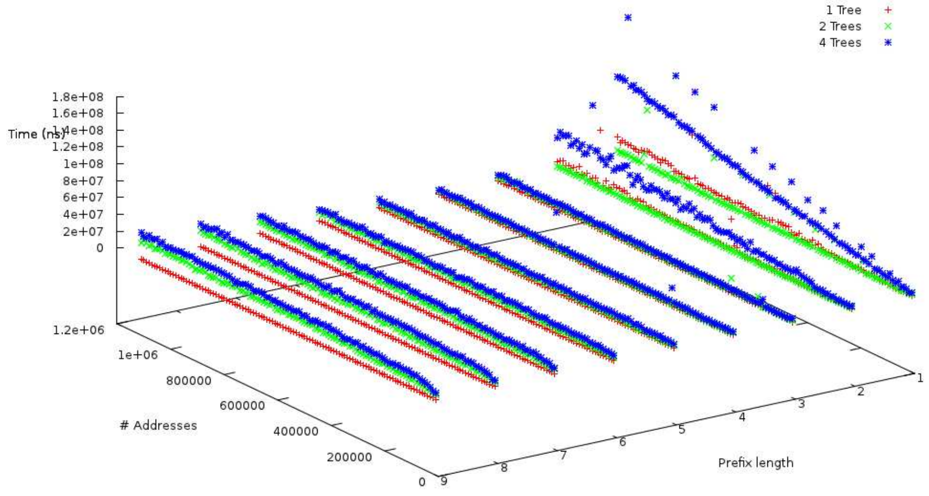


Figure 4.17: Scalability of the run-times for the suggestion making process for varying prefix lengths and number of addresses with different numbers of Trees working in parallel

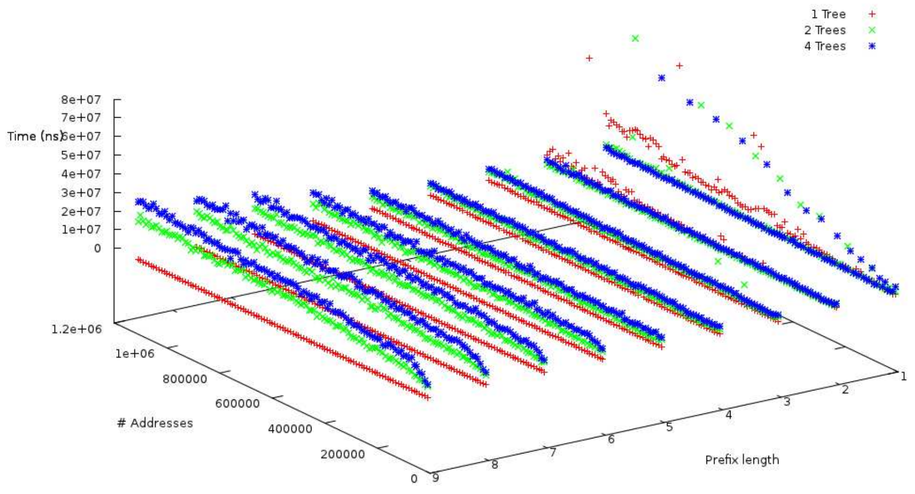


Figure 4.18: Scalability of the run-times for the suggestion making process for varying prefix lengths and number of addresses with different numbers of Trees working in parallel

4.5 Summary

In this chapter we have tested the hypotheses made in developing the Probabilistic Radix Tree model used by our auto-completion algorithms. We have seen that the radix tree offers significant improvements over a prefix tree both in terms of memory requirements, as well as in terms of suggestion making run-time performance.

We have also seen that we can successfully meet all the (main and optional) requirements that we introduced in Chapter 1, and we have learned that altering the distribution of weights associated to different suggestions is a very powerful tool in obtaining the best performance from the system, but, also very importantly, we have shown that this tool is not at all hard to use.

Chapter 5

Conclusions and Future Work

In this thesis we began with a presentation of TomTom's business need, which was the implementation of auto-completion functionality for the Andorra geocoding system. We started by separating the main requirements from the optional requirements. The main requirements were typical of general auto-completion systems: real-time performance and the support for setting priorities or weights to suggestions. Systems implementing such functionality exist in numerous applications, and techniques for implementing them have been investigated in literature. The same applies to Optional Requirement 2 identified in the first chapter. The optional requirements 1 and 3, on the other hand, are specific to geocoding systems, and although systems offering such functionality exist, there are no publicly available descriptions of algorithms for implementing such functionality, to the best of our knowledge.

In Chapter 2 we have surveyed existing literature for approaches to dealing with the problem of real-time auto-completion in the general context of information retrieval systems. Building on the conclusions from related literature, we opted for a prefix tree based approach, which we then gradually modified in order to meet all the requirements identified for our auto-completion system.

We have described ways of adding the required functionality while still meeting the very strict run-time requirements. The most important thing to mention here is the impact that the probability distribution over the supported suggestions has on the run-time of the suggestion making process, as identified in the previous chapter. These notes are very important for getting the right performance when using the auto-completion system in specific applications. We have tried to offer some guidelines through our experiments, but ultimately it will be up to the application that will make use of the system to configure its data for optimal performance.

Perhaps the most important conclusion is that different levels of importance need to be defined, such that suggestions within different levels are quite clearly separated. We have seen that at least one 'high-priority' level of relatively few suggestions is necessary to be able to handle prefix strings of only 1 or 2 characters. This requirement can be met quite naturally in the context of geocoding, as a class of high priority suggestions can be defined as the set of supported country name suggestions, or state names, etc. It is also possible to define this high priority class of suggestions based on other criteria completely unrelated to the type of suggestions. For instance, the frequency with which a particular suggestion is found relevant by users could be used to define a class of high priority suggestions, regardless of their type (street, city, etc.).

Having met all the requirements that have been set in the first chapter, there are two main directions for future work: improving the relevance of suggestions being made by properly assigning weights to them, and dealing with the implementation details that need to be addressed in order to get the system into production. We will look at these two directions more carefully in the following section.

5.1 Future work

It is beyond the scope of this thesis to investigate how to best distribute suggestion probabilities or weights to obtain the best results both in terms of suggestion making run-time performance and relevance of suggestions. Thus we will leave it for future work to find ways of best separating the suggestions into a set of levels of importance and then defining distributions within each level in order to meet this double requirement. This can also be interpreted as an optimization problem to be solved, and numerous techniques can be tried for this purpose. Distributions can be defined taking into consideration such data as the length of streets, or the importance of streets, coupled with the importance of the cities or countries they are in. Such information can be derived from the source data, and may offer useful hints. Ideally, though, use logs for the geocoder can be used to define such distributions based on what users search. Also, once up and running, the auto-completion system could employ learning mechanisms based on its own usage, in order to optimize its performance.

There are also still some challenges to get from the implementation used for this thesis project to something that can be used in a production system. The main concern is the relatively high memory requirements. Future work should focus on possible alterations that can be made to the implementation in order to reduce memory requirements. Ideally, better choices can be made for the tree implementation without changing any of the interfaces or template methods (see Appendix A).

Ultimately, depending on the production environment, it may be easier to distribute work across multiple computing nodes. As we've established in the previous chapter, with some consideration for how the distribution of suggestion probabilities is affected, this is a viable solution.

Another idea for future work is to try an approach to making prefix string corrections even if there are suggestions that exactly match the supplied user prefix. This can be an option if we know the user made a very common typo which gives them an obscure suggestion, whereas without the typo a very common suggestion could be made. Then we may decide to at least add the highly weighted suggestion to the result set after the exact matching one. To implement this, a different error tolerant prefix matching strategy is required, perhaps similar to the one in [10], but this would require first collecting data about what spelling mistakes are common.

Appendix A

Implementation Details

In this appendix we describe the implementation of the software system that was used for the experimental evaluation. Ideally, this system should come as close as possible to one that could be used in production. The aim of this appendix is to serve as a documentation for the code written. The code consists of three packages (in the sense that packages are defined in the Java programming language) which we will discuss in subsequent sections. For each package, we will survey the classes, their roles within the system, details of the implementation where necessary, as well as possible considerations for future use.

A.1 The *suggestions* package

The *suggestions* package defines classes that represent suggestions to which the Probabilistic Radix Tree can point. Here we define a simple implementation of the suggestion manager, which is the key component in allowing flexible prefix matching and error tolerant and location biased suggestion making. We also define the so-called *SuggestionsEngine* which serves as the entry point for the auto-completion system, providing a simple interface that allows a user of the system to make use of all its functionality by interacting with it via only two method calls. The UML diagram corresponding to this package is shown in Figure A.1.

The classes in this package are as follows:

- *Suggestion*. This is an interface that defines the minimal behavior of a suggestion that the system supports. Specifically, a Suggestion should offer a way of retrieving the *suggestionString*, which is the string displayed to the end-user as a completion suggestion and a method that returns all strings that can be used to match that suggestion. This last method is called *getMatchingStrings* and it takes as a parameter a StringNormalizer (discussed below) and a weight, and it returns a set of strings that can be used to match this suggestion, along with their corresponding weights, calculated based on the weight attributed to the suggestion and the possible distortion applied to the *suggestionString* in order to get the matching string.

- *AddressSuggestion*. This is an abstract class that implements Suggestion (speaking only strictly from a Java language point of view, as it does not actually implement any of its methods), and it imposes that concrete classes extending it have associated with them a location (a latitude and longitude pair) that can be retrieved by implementing the abstract methods defined here for this purpose. The class does implement a method *distance* for calculating the distance to a bias point provided as a parameter, using its own latitude and longitude. For now this method has a simple implementation which serves as an approximation: it treats coordinates as points in a plane and computes the Euclidean distance between them. For more accuracy, this method can be implemented to return the actual distance in e.g. kilometers. Note that this exact implementation will also affect the choice of the weight distribution for supported queries.

- *StringNormalizer* is an interface which exposes a method *normalizeString* which should be applied both on matching strings that can be parsed using the edges of the radix tree, as well as on prefixes provided to retrieve suggestions, in order to allow for some flexibility when parsing user-supplied prefixes.

- *AddressStringNormalizer* implements *StringNormalizer* such that strings passed to the *normalizeString* method are lower cased and all delimiter sequences are replaced by a single placeholder character: $. Hence the strings ”Frankfurt (Oder)” and ”frankfurt-oder ” are the same when normalized using this method.

- *CitySuggestion* and *StreetSuggestion* are classes that implement *AddressSuggestion* and are constructed by supplying all the elements necessary for describing such suggestions in each case, such as street name, city name, lat-long, etc.

- *SuggestionsManager* is an interface describing the functionality of an associative data structure which maps keys to suggestions and offers a way of retrieving keys based on suggestions and vice-versa. The addition of new suggestions is meant to be transparent to the user, as getting a suggestion key should result in the creation of one, in case a key is not already associated to the supplied suggestion.

- *SuggestionsManagerImpl* is a simple implementation of *SuggestionsManager* using integer keys and managing their association to suggestions by using an in-memory hash map.

- *SuggestionsEngine* is a concrete class, offering a simple interface (not in the Java language sense) to the auto-completion system. It manages a *Tree* instance (see section A.2) and through calling its constructor, it is provided a *DataReader* (see section A.3) that populates the tree with suggestions and their supported strings. It also exposes methods for getting a ranked list of top-*k* suggestions. This is accomplished by simply wrapping around similar methods exposed by the tree, after only applying normalization to the prefix provided as a parameter.

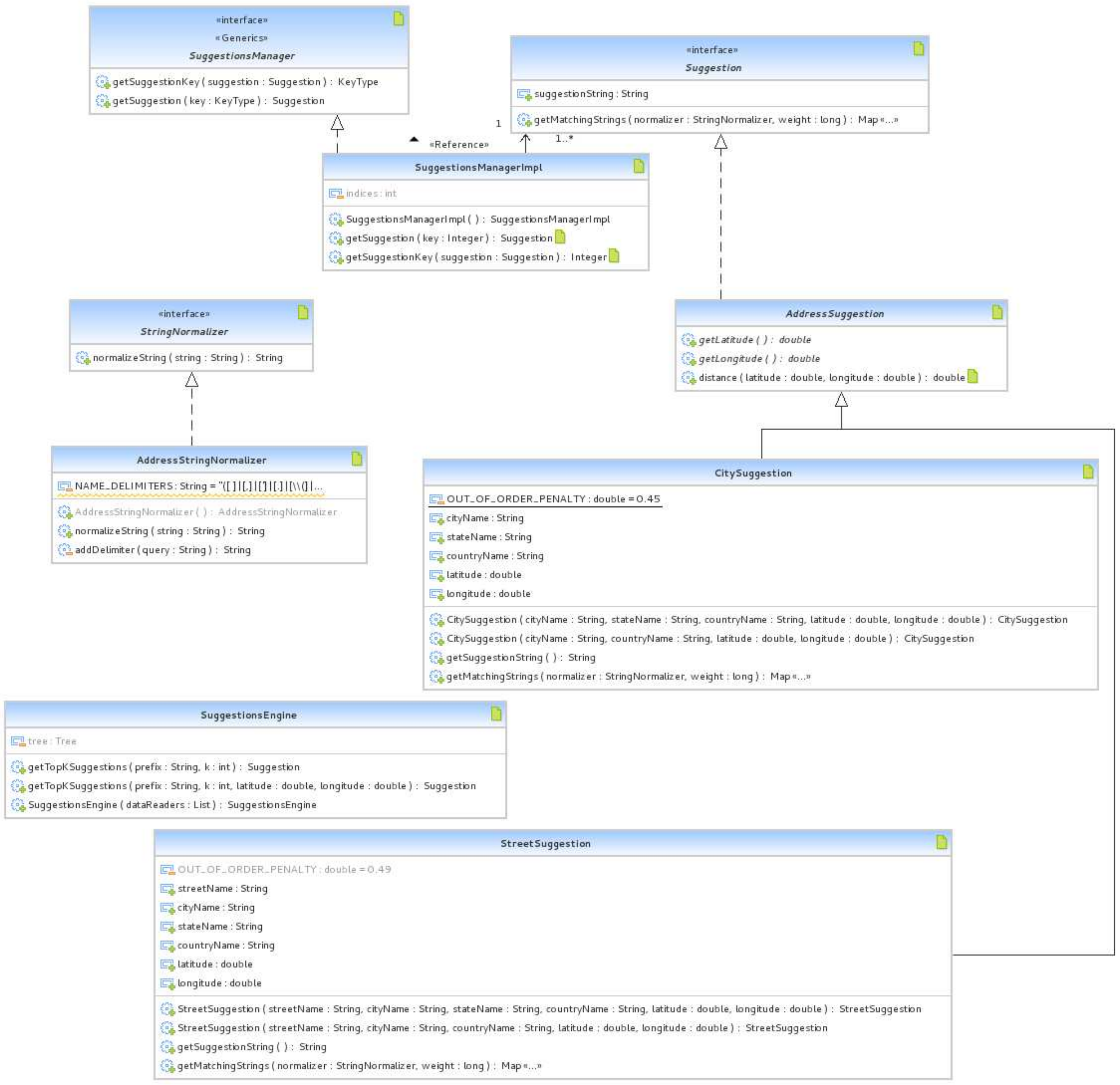

«interface»
«Generics»
SuggestionsManager
getSuggestionKey (suggestion : Suggestion) : KeyType
getSuggestion (key : KeyType) : Suggestion
«interface»
Suggestion
suggestionString : String
getMatchingStrings (normalizer : StringNormalizer, weight : long) : Map«...»
1
1..*
«Reference»
SuggestionsManagerImpl
indices : int
SuggestionsManagerImpl ( ) : SuggestionsManagerImpl
getSuggestion (key : Integer) : Suggestion
getSuggestionKey (suggestion : Suggestion) : Integer
«interface»
StringNormalizer
normalizeString (string : String) : String
AddressSuggestion
getLatitude ( ) : double
getLongitude ( ) : double
distance (latitude : double, longitude : double) : double
AddressStringNormalizer
NAME_DELIMITERS : String = "([ ]|[,]|[']|[.]|[\\(]|...
AddressStringNormalizer ( ) : AddressStringNormalizer
normalizeString (string : String) : String
addDelimiter (query : String) : String
CitySuggestion
OUT_OF_ORDER_PENALTY : double = 0.45
cityName : String
stateName : String
countryName : String
latitude : double
longitude : double
CitySuggestion (cityName : String, stateName : String, countryName : String, latitude : double, longitude : double) : CitySuggestion
CitySuggestion (cityName : String, countryName : String, latitude : double, longitude : double) : CitySuggestion
getSuggestionString ( ) : String
getMatchingStrings (normalizer : StringNormalizer, weight : long) : Map«...»
SuggestionsEngine
tree : Tree
getTopKSuggestions (prefix : String, k : int) : Suggestion
getTopKSuggestions (prefix : String, k : int, latitude : double, longitude : double) : Suggestion
SuggestionsEngine (dataReaders : List) : SuggestionsEngine
StreetSuggestion
OUT_OF_ORDER_PENALTY : double = 0.49
streetName : String
cityName : String
stateName : String
countryName : String
latitude : double
longitude : double
StreetSuggestion (streetName : String, cityName : String, stateName : String, countryName : String, latitude : double, longitude : double) : StreetSuggestion
StreetSuggestion (streetName : String, cityName : String, countryName : String, latitude : double, longitude : double) : StreetSuggestion
getSuggestionString ( ) : String
getMatchingStrings (normalizer : StringNormalizer, weight : long) : Map«...»

Figure A.1: The UML diagram corresponding to the *suggestions* package

A.2 The *trees* package

The *trees* package is the core package containing the definition of the Probabilistic Radix Tree, along with the other components necessary for its implementation. Figure A.2 shows the UML diagram generated from these components. Next we discuss the main classes in this package individually:

- *Node* is an interface to a node in the abstract *Tree* structure. The interface requires that implementers offer a way to add terms to the node, retrieve suggestions associated to this node, get the total weight corresponding to the node and get the descendant nodes in the tree structure.

- *Tree* is an abstract class requiring implementations for (abstract) methods for adding suggestions, parsing prefix strings (and returning the corresponding *Node*), both with and without error tolerance support. The class does implement a template method for getting a ranked list of the top-*k* suggestions corresponding to a given prefix. The method corresponds to Algorithm 5 and its variations described in chapter 3. It is a template method because it relies on subclasses to implement the abstract methods that it uses, for getting states based on prefix strings. This same method is used for both biased and unbiased suggestion making, depending on the type of *CandidateAdder* parameter it is provided, as discussed next.

- *CandidateAdder* is an interface, exposing methods the implementers of which have to define, in order encapsulate the way suggestions are added to a priority queue. This is a step in the suggestion making algorithm of the *Tree* and encapsulating it into objects of base type *CandidateAdder* allows us to use the same method for unbiased or different kinds of biased suggestion-making, simply by changing the concrete type of *CandidateAdder* that is passed to the method (this is an implementation of the 'Strategy' software design pattern). Implementations are currently provided for simple addition of Suggestions according to their weights (*PlainCandidateAdder*), addition of suggestions with rescaled weights according to the distance to a provided bias point (*BiasedCandidateAdder*) and addition of suggestions with rescaled weights according to the distance to a given bias point, but only outside a provided radius (*BiasedRadiusCandidateAdder*).

- *SuggestionCandidate* is a class encapsulating an entry in the priority queue used by the suggestion making algorithm of *Tree*. It is meant to hold a reference to either a state to be expanded in the suggestion making process or a suggestion that can be made to the user. *SuggestionCandidate*s are instantiated by *CandidateAdder*s, which assign weights to them according to the type of the suggestion making process (biased/unbiased/etc.).

- *RadixNode* and *RadixTree* offer complete implementations of the *Node* and *Tree* abstractions respectively, according to the algorithms discussed in this thesis for Probabilistic Radix Trees. The intent of this separation is to define the algorithms in an abstract way as far as possible, while allowing for different implementation decisions.

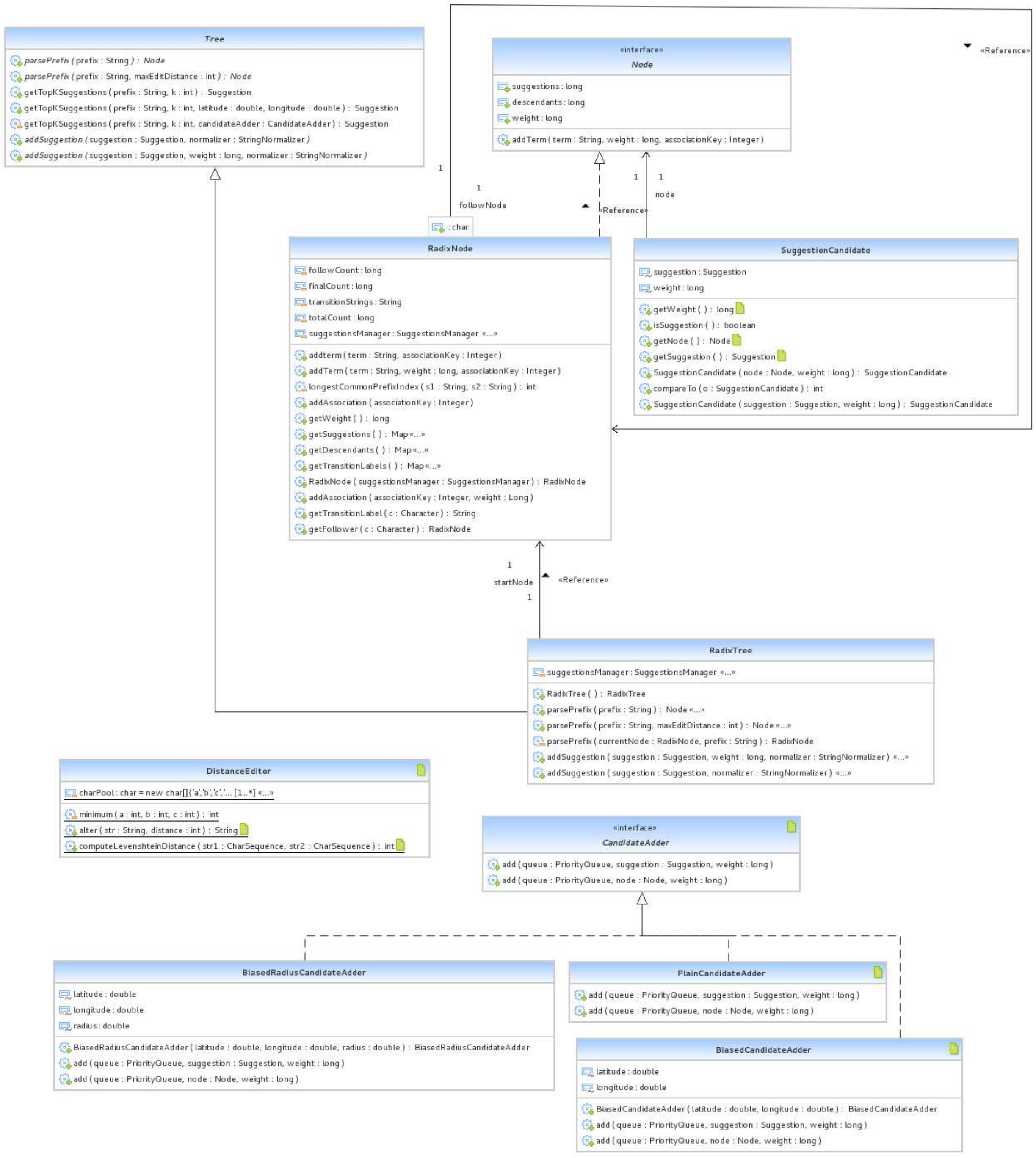

Tree
parsePrefix (prefix : String) : Node
parsePrefix (prefix : String, maxEditDistance : int) : Node
getTopKSuggestions (prefix : String, k : int) : Suggestion
getTopKSuggestions (prefix : String, k : int, latitude : double, longitude : double) : Suggestion
getTopKSuggestions (prefix : String, k : int, candidateAdder : CandidateAdder) : Suggestion
addSuggestion (suggestion : Suggestion, normalizer : StringNormalizer)
addSuggestion (suggestion : Suggestion, weight : long, normalizer : StringNormalizer)
«interface»
Node
suggestions : long
descendants : long
weight : long
addTerm (term : String, weight : long, associationKey : Integer)
«Reference»
1
1
followNode
«Reference»
1
1
node
: char
RadixNode
followCount : long
finalCount : long
transitionStrings : String
totalCount : long
suggestionsManager : SuggestionsManager «...»
addterm (term : String, associationKey : Integer)
addTerm (term : String, weight : long, associationKey : Integer)
longestCommonPrefixIndex (s1 : String, s2 : String) : int
addAssociation (associationKey : Integer)
getWeight ( ) : long
getSuggestions ( ) : Map «...»
getDescendants ( ) : Map «...»
getTransitionLabels ( ) : Map «...»
RadixNode (suggestionsManager : SuggestionsManager) : RadixNode
addAssociation (associationKey : Integer, weight : Long)
getTransitionLabel (c : Character) : String
getFollower (c : Character) : RadixNode
SuggestionCandidate
suggestion : Suggestion
weight : long
getWeight ( ) : long
isSuggestion ( ) : boolean
getNode ( ) : Node
getSuggestion ( ) : Suggestion
SuggestionCandidate (node : Node, weight : long) : SuggestionCandidate
compareTo (o : SuggestionCandidate) : int
SuggestionCandidate (suggestion : Suggestion, weight : long) : SuggestionCandidate
1
startNode
1
«Reference»
RadixTree
suggestionsManager : SuggestionsManager «...»
RadixTree ( ) : RadixTree
parsePrefix (prefix : String) : Node «...»
parsePrefix (prefix : String, maxEditDistance : int) : Node «...»
parsePrefix (currentNode : RadixNode, prefix : String) : RadixNode
addSuggestion (suggestion : Suggestion, weight : long, normalizer : StringNormalizer) «...»
addSuggestion (suggestion : Suggestion, normalizer : StringNormalizer) «...»
DistanceEditor
charPool : char = new char[]{'a','b','c',... [1..*] «...»
minimum (a : int, b : int, c : int) : int
alter (str : String, distance : int) : String
computeLevenshteinDistance (str1 : CharSequence, str2 : CharSequence) : int
«interface»
CandidateAdder
add (queue : PriorityQueue, suggestion : Suggestion, weight : long)
add (queue : PriorityQueue, node : Node, weight : long)
BiasedRadiusCandidateAdder
latitude : double
longitude : double
radius : double
BiasedRadiusCandidateAdder (latitude : double, longitude : double, radius : double) : BiasedRadiusCandidateAdder
add (queue : PriorityQueue, suggestion : Suggestion, weight : long)
add (queue : PriorityQueue, node : Node, weight : long)
PlainCandidateAdder
add (queue : PriorityQueue, suggestion : Suggestion, weight : long)
add (queue : PriorityQueue, node : Node, weight : long)
BiasedCandidateAdder
latitude : double
longitude : double
BiasedCandidateAdder (latitude : double, longitude : double) : BiasedCandidateAdder
add (queue : PriorityQueue, suggestion : Suggestion, weight : long)
add (queue : PriorityQueue, node : Node, weight : long)

Figure A.2: The UML diagram corresponding to the *trees* package

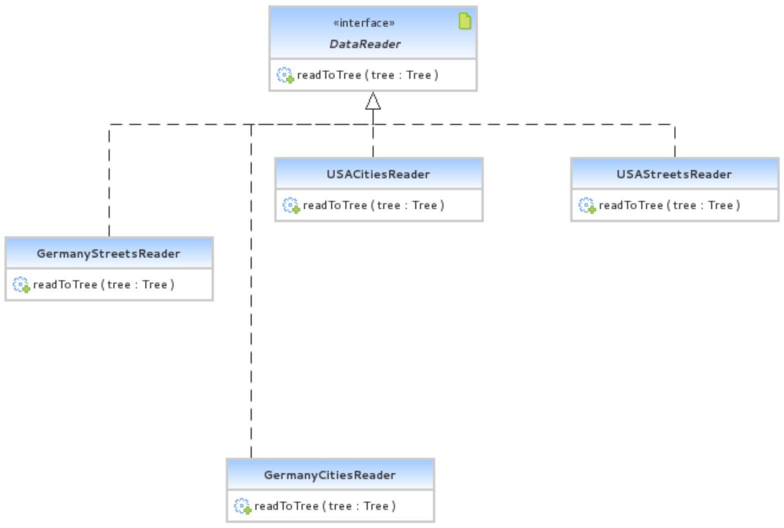

«interface»
DataReader
readToTree (tree : Tree)
USACitiesReader
readToTree (tree : Tree)
USAStreetsReader
readToTree (tree : Tree)
GermanyStreetsReader
readToTree (tree : Tree)
GermanyCitiesReader
readToTree (tree : Tree)

Figure A.3: The UML diagram corresponding to the *data* package

- *DistanceEditor* is a utility class which implements edit-distance calculation between two strings, as it is required by the algorithms discussed in this thesis. That is, it is passed two string parameters and it returns the minimum distance necessary to transform prefixes of the first string into the second one, along with the indices that delimit these prefixes.

A.3 The *data* package

The *data* package consists of classes that populate a *Tree* with supported strings from a given data source. This will need to change (or at least be expanded) in order to incorporate support for application-specific data sources and processing. There is an interface that defines a simple protocol for populating a *Tree* from a given data source, called *DataReader*. Concrete *DataReader*s have to encapsulate the knowledge of reading and interpreting a known data source and making the necessary calls to the supplied tree to support the suggestions they require.

Figure A.3 shows the implementations of *DataReader* interface that have been used in the experimental evaluation presented in this thesis.

Bibliography

[1] B. Filipic T.R. Lynam B. Zupan A. Bratko, G. V. Cormack. Spam filtering using statistical data compression models. *Journal of Machine Learning*, 7:2673–2698, 2006.

[2] E. Alpaydin. *Introduction to Machine Learning, Second Edition*. The MIT Press, 2010.

[3] R. Asok. Symbolic dynamic analysis of complex systems for anomaly detection. *Signal Processing*, 84(7):1115–1130, 2004.

[4] H. Schutze C.D. Manning, P. Raghavan. *An introduction to Information Retrieval*. Cambridge University Press, 2009.

[5] F.J. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7:171–176, 1964.

[6] C. de la Higuera. Characteristic sets for polynomial grammatical inference. 1995.

[7] H. Hembrooke T. Joachims B. Pan G. Gay, L. Granka. Accurately interpreting click-through data as implicit feedback. *ACM Conference On Research and Development In Information Retrieval*, 28, 2005.

[8] H. Hembrooke T. Joachims B. Pan F. Radlinski G. Gay, L. Granka. Evaluating the accuracy of implicit feedback and query reformulations in web search. *ACM Transactions on Information Systems (TOIS)*, 25, 2007.

[9] I. Weber H. Bast. Type less, find more: Fast autocompletion search with a succinct index. *ACM Conference On Research and Development In Information Retrieval*, 29, 2006.

[10] P. Hsu H. Duan. Online spelling correction for query completion. *WWW*, 2011.

[11] C. von der Malsburg L. Wiskott J. Fellous, N. Kruger. Face recognition by elastic bunch graph matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19:775–779, 1997.

[12] R.B. Miller. Response time in man-computer conversational transactions. *AFIPS*, 1968.

[13] M. Mohri. Finite-state transducers in language and speech recognition. *Association for Computational Linguistics*, 23(2):269–312, 1997.

[14] A. Ray P. Adenis, K. Mukherjee. State splitting and state merging in probabilistic finite state automata. *American Control Conference*, pages 5145–5150, 2011.

[15] J. Liu P. Dolan, E. R. Pedersen. Personalized news recommendation based on click behavior. *Proceedings of the 15th international conference on Intelligent user interfaces*, 2010.

[16] F. Thollard P. Dupont, C. de la Higuera. Probabilistic dfa inference using kullback-leibler divergence and minimality. *Proceedings of the 17th International Conference on Machine Learning*, 2000.

[17] Y. Esposito P. Dupont, F. Denis. Links between probabilistic automata and hidden markov models: probability distributions, learning models and induction algorithms. *Pattern Recognition*, 38(9):1349–1371, 2005.

[18] C. de la Higueroa F. Thollard E. Vidal R. C. Carrasco, F. Casacuberta. Probabilistic finite state machines - parts i and ii. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(7):1013–1025, 2005.

[19] J. Oncina R. C. Carrasco. Learning stochastic regular grammars by means of state merging method. *Springer-Verlag*, pages 139–152, 1994.

[20] J. Oncina R.C. Carrasco. Learning deterministic regular grammars from stochastic samples in polynomial time. *RAIRO (Theoretical Informatics and Applications)*, 33:1–20, 1999.

[21] S. Inenage G. Mauri G. Pavesi A. Shinohara M. Takeda S. Arikawa, H. Hoshino. Online construction of compact directed acyclic word graphs. *Discrete Applied Mathematics - 12th annual symposium on Combinatorial Pattern Matching*, pages 169–186, 2001.

[22] R. Kaushik S. Chaudhuri. Extending autocompletion to tolerate errors. *ACM SIGMOD*, 2009.

[23] S. Verwer. *Efficient Identification of Timed Automata, theory and practice*. 2011.