Semantic Web Empowered City Trip Planning

Master Thesis of Chris Dijkshoorn & Bas Groenewoud

Year
2011
Supervisors
Emanuele Della Valle
Spyros Kotoulas
Second Reader
Frank van Harmelen

VU UNIVERSITY AMSTERDAM
POLITECNICO DI MILANO

Acknowledgements

Firstly, we would like to thank our supervisors Emanuele della Valle and Spyros Kotoulas. Emanuele della Valle for not dictating what we should do and letting us make our own mistakes, while subtly steering us in the right direction. Every meeting, we proposed a number of new features. Helping us deal with the ever growing size of the application, by discussing alternative approaches and solutions, has helped us focus on the things that matter.

We thank Spyros Kotoulas for his adequate and fast support, concerning many things. While working on a mini master project, we stated the desire to study abroad. Before we knew it, a position in Italy was arranged. Helping us manage working with the LarKC platform and taking care of many organizational things are among the things we are very grateful for.

Secondly we would like to thank the people at Cefriel, especially Daniele Dell'Aglio and Irene Celino. The provided support and knowledge concerning the Semantic Web and specifically LarKC, was very welcome. When we proposed to present our end result at Cefriel, we were surprised it was taken so seriously. Nevertheless we enjoyed the day and the presentation.

Thirdly we would individually like to thank some people for their help in realizing this master thesis. *Chris:* I would like to thank my family, for their unconditional support, despite that I gave them a hard time explaining to people, what it exactly was I did in Italy. I also appreciate the exercised patience of all my friends visiting from the Netherlands while listening to city trip planner related stories. I thank Bas for taking the time to consider some of my ideas and bringing forth the energy for discussing whether we should actually be implementing them.

Bas: I would like to thank my girlfriend Tessa for her love, support, her multiple visits to Italy and not getting too angry for not showing up on Skype whenever I was working on the city trip planner and forgot about the time. I would also like to thank my parents and my brother for their unconditional support and also for their visit to Como. I would like to thank Chris for his endless flow of new ideas, his never ending enthusiasm and for always trying to make the problem even more complex. Finally, I would like to thank the Italian people for making my stay in Como much more comfortable by providing me with excellent coffee, tasty pizza's and the always delicious ice creams.

Abstract

Making a planning for a city trip is not an easy thing to do. Often, the time a tourist spends in a city is limited so choices of what to visit have to be made. Making a careful selection of things to visit and deciding on the order of visiting them, takes time. Support in this process would sometimes be welcome. In this master thesis, we introduce an application which automates this task, using the LarKC platform and Semantic Web techniques. The used dataset is deducted from the LinkedGeoData.org project and an ontology is created to structure it.

We introduce two strategies for selecting points of interest: the Distance Times Rating strategy and the Radius strategy. Each strategy is able to generate multiple sets with selected points. Next, a Traveling Salesman Problem with Time Windows algorithm is used to determine the optimal sequence of visiting the points, resulting in multiple city trip plans.

The city trip plans are ranked based on the quality of the included points, the time spent at those points and the travel time. The two selection strategies are evaluated using the Discounted Cumulated Gain method. This is an evaluation measure which considers the relevance of the found city trip plan in combination with its position in the list of results. The city plans generated by the two selection strategies are compared to an optimal situation, created by running a baseline that generates all the possible combination of points of interest, considering a given set of user preferences.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Tourists visiting a city for a short period of time will not be able to visit everything the city has to offer. Therefore, a selection has to be made. Guidebooks are widely used to find appropriate activities in an unknown city. The books include information collected by editors and new versions are published on regular basis to keep the provided information up to date. The guidebook is an important part of the process of visiting a new place: "because it mediates the relationship between tourist and destination, as well as the relationship between host and guest" [9]. But would it not be better to use information that is always up to date and tailored to a person's interests [13]?

After deciding on what to do in a city, a tourist still has to figure out in what order to visit the chosen places. Considering the locations and opening times of interesting sights and activities, this can be a complex puzzle similar to the Traveling Salesman Problem (TSP) [3] and its variants, the Traveling Salesman Problem with Time Windows (TSPTW) [19] and the Time Dependent Traveling Salesman Problem (TDTSP) [23]. These problems are extensively researched in the field of Operational Research (OR) and algorithms finding optimal solutions in a limited period of time do exist.

The combination of the processes of choosing what to visit and in what order to visit the chosen points is called the Tourist Trip Design Problem (TTDP) [49]. By solving this design problem automatically in a short time span while considering the preferences of the user, a tourist could set out to experience the city in moments. However, there is no easy way of solving the problem, mainly due to the huge amount of different possible combinations [45].

Multiple attempts to create electronic guidebooks were made [16] [30], but one of the problems which arose, concerned the acquisition of appropriate data and maintaining this data. The most common solution is hiring staff who enter the data manually. Depending on the size of the city, this is quite a demanding task. When the data also comprises information about events, continuous revisions have to be made. Luckily, today a lot of relevant data is published on the Internet, enabling the automation of this data acquisition process.

Services such as Google Maps[1], Foursquare[2] and Eventful[3] provide a variety of useful information. Some of the data in these datasets is "user generated" and, provided that there is a lively community, it will be updated on a frequent basis. None of the sources mentioned above will include all the information that the others include, so in order to enrich the obtained data, combining information from multiple sources is useful.

Semantic Web techniques can help us effectively combine data from all over the Web [44]. The appropriate way of saving this data is by using the Resource Description Framework (RDF) [12]. The data can be structured and interlinked using the Web Ontology Language (OWL) [40]. Taxonomies of tourist attractions are already available and can, with some adaptions and refinements, function as a basis for a useful tourist attraction ontology [35].

There are multiple sources of geographical information on the Internet, although sources utilizing Semantic Web techniques are more scarce. GeoNames[4] offers a well structured dataset containing a lot of information about geographical entities such as countries and place names. More fine grained information can be obtained using the OpenStreetMaps initiative [28], which is a popular system enabling collaborative map making. Many Points Of Interest (POIs) are identified and due to the lively community, the amount and quality of the data keeps increasing [43]. Data from the OpenStreetMaps project is transformed into RDF and made available by LinkedGeoData.org, which provides a SPARQL endpoint but also releases complete datasets [7]. In addition to providing information about POIs, extracts of the maps can also be used in combination with routing algorithms to find paths from one point to another [43].

Processing the obtained data and reasoning over it is no easy task. The use of a Semantic Web platform can simplify the process. LarKC is such a platform, it is defined by the developers as: "a platform for massive distributed incomplete reasoning that will remove the scalability barriers of currently existing reasoning systems for the Semantic Web" [20]. The LarKC platform is developed to make effective reasoning with huge amounts of data, from different data sources, possible. LarKC allows developers to build applications using different kinds of plug-ins that run on top of the platform.

A use case developed for the platform relevant to city trip planning, is the Urban LarKC application [47] [48]. It focuses on a specific case of pervasive computing: urban computing. Whereas most forms of pervasive computing handle small contained spaces with sensors (e.g. "smart houses" and "intelligent rooms supporting the elderly"), urban computing is concerned with large urban environments like cities [31]. It is hard to rig a city with sensors, due to privacy issues and it being a costly activity. This problem is, however, solved by a multitude of data sources made available for different reasons. Take for example traffic information, timetables for public transport and data concerning events and sights in a city. Combining and reasoning over this heterogeneous data is

[1] http://maps.google.com/
[2] https://foursquare.com/
[3] http://eventful.com/
[4] http://www.geonames.org/

a perfect test case for the LarKC platform. At the moment the Urban LarKC application provides users with three types of information: information about events, information about monuments and path finding information.

Combining the Urban LarKC project features with ways to solve the TTDP could result in very useful city trip plans from a tourists point of view. The used data will be up to date and structuring it using an ontology will enable intelligent reasoning for solving the TTDP. This adds another layer of reasoning to the current implemented features of the Urban LarKC application: the selection of appropriate POIs and the process of finding an optimal route visiting these points. We believe that using LarKC in combination with plug-ins containing strategies to pick the appropriate POIs and a plug-in for finding the shortest route, will result in useful city trip plans in a short period of time. This leads to the formulation of the following hypothesis:

> **Hypothesis** "The LarKC platform is able to generate 10 city trip plans of good quality in less than a minute, using strategies for selecting points of interest from the web of data combined with a TSPTW solver."

The research questions below will clarify the goals of the research we conduct. The first question refers to the point selection process. Since the chosen points will greatly influence the quality of the resulting city plan, an important goal of the research is to develop strategies which intelligently pick the POIs.

> **Research Question 1** How can we effectively select a set of points of interest which will result in a good city trip plan?

Research question number 2 is a data related question. When we do not want to enter the data about interesting points ourselves, we need datasets to extract the appropriate information from. The earlier mentioned GeoNames and LinkedGeoData datasets could be a source of POI information. Structuring this data is a must in order to be able to differentiate between different kinds of POIs. A possibility is that the gathered data is not structured in the way we would like it to be, so we need to find a way to transform it.

> **Research Question 2** Which datasets do we use for the extraction of points of interest and how to structure this data?

We have to find ways to evaluate the city trip plans we obtain. A set of user preferences can result in a multitude of city trip plans. Which plans are great and which ones are poor? How to compare the obtained results, considering time aspects? A tourist will probably not want to wait a couple of days for a result. The last research question is all about this evaluation process.

> **Research Question 3** How to evaluate the quality of a city trip plan?

The creative process leading towards this master thesis, as well as the process of writing the master thesis itself, are done in close collaboration, where the workload is equally divided between the authors. Every important decision in the process was made in consultation with our supervisors and supported by the both of us. In order to provide the exam committee of the VU University Amsterdam with the possibility of judging us separately, we distinguish ourselves by both developing a selection strategy. The DTR selection strategy is developed by Chris Dijkshoorn and described in Section 5.2. The Radius strategy is developed by Bas Groenewoud and described in Section 5.3. The master thesis is written in the context of the LarKC project under direct supervision of Emanuele Della Valle at the Como department of Politecnico di Milano, made possible by an internship scholarship of Erasmus.

The remaining of this master thesis is structured as follows. Chapter 2 gives an overview of relevant technologies and the city trip planning applications which are currently available. The problem setting can be found in Chapter 3. Chapter 4 contains the rationale for the architecture of the LarKC application. Strategies that intelligently create sets of points of interest are extensively discussed in Chapter 5. Chapter 6 is a chapter dedicated to the data used within the application. Our experiences while developing the application are documented in Chapter 7. The evaluation of the point selection strategies and overall application is given in Chapter 8. A discussion of these results is given in Chapter 9 in addition to an overview of possible extensions. The final chapter is the conclusion of this master thesis.

Chapter 2

State Of The Art

In this chapter we relate our electronic city trip planner to other areas. Since we are using Semantic Web technologies and data, we elaborate on the Semantic Web and specifically the Resource Description Framework and the Web Ontology Language in Section 2.1. In Section 2.2, we provide an extensive insight in the LarKC platform, since our application runs on this platform. In Section 2.3 we continue with an general introduction into the traveling salesman problem. We also give an introduction into more restrained versions of this problem, namely the traveling salesman problem with time windows and the time dependent traveling salesman problem with time windows, because solving TSP related problems is a significant part of the thesis. In Section 2.4, we introduce two sources of data which are the most prominent data providers used in the application: OpenStreetMap and LinkedGeoData. The next two Sections are focused on travel guides. Section 2.5 focuses on the current state of the art in electronic guide books. We conclude this chapter with Section 2.6, containing a description of the competitors in the field: the automated city trip planners.

2.1 Semantic Web

This section is used to elaborate on the Semantic Web. We start with an introduction of the basics of the Semantic Web in Section 2.1.1. The Linked Data principles are given in Section 2.1.2. The main building block used for describing the data is the Resource Description Framework and is described in Section 2.1.3. We conclude with Section 2.1.4, containing a description of languages which are used to define the data and the relations between concepts, RDF Schema and the Web Ontology Language.

2.1.1 Semantic Web Basics

The Semantic Web, as envisioned by Tim Berners-Lee, extends the World Wide Webs infrastructure with techniques making represented information not only

readable for humans, but also interpretable and operable for machines [8]. The key principles of the Semantic Web are: making datasets available on the Web, data integration through the interlinking of datasets and the addition of semantics to the data through the use of ontologies.

Standards have been developed to enable the publication and structuring of data on the web. The Resource Description Framework (RDF) is used to make statements about resources. Resources are “things" we want to talk about, like a person, a book or a place. These resources are identified by Universal Resource Identifiers (URIs) [2]. RDF Schema extends RDF and provides a minimal ontology representation language, whereas the Web Ontology Language provides more expressiveness [44]. The SPARQL query language is used to retrieve information from RDF data sources.

Ontologies can be seen as common conceptualizations, they define the data and relations between concepts. A multitude of ontologies are developed, describing diverse concepts ranging from social network information to modeling information in life sciences. By utilizing the knowledge obtained from ontologies, machines can reason over data, thereby deducting even richer information. A summary of the information originating from a multitude of sources and reasoning procedures can be presented to a user, surpassing by far the information a user could have deducted on its own.

2.1.2 Linked Data

The World Wide Web uses hyperlinks to connect one page to another. Applying this simple but useful principle to data by linking one resource to another results in a web of data. Over the years, an increasing amount of datasets is interlinked using URIs. The Linking Open Data Cloud as depicted in Figure 2.1 expands every year as a result of this. The web of data is evolving into one global data space, steadily expanding whenever a new dataset is added and interlinked with the already included datasets.

Linked Data is a term used for a set of best practices for publishing and connecting structured data on the Web [11]. Four guidelines, as introduced by Tim Berners-Lee, help integrate the different datasets in a global data space:

1. Use URIs as names for things
2. Use HTTP URIs so that people can look up those names
3. When someone looks up a URI, provide useful information, using the standards (RDF)
4. Include links to other URIs, so that they can discover more things

Figure 2.1 depicts a cloud of datasets adhering to these principles. The arrows between the datasets represent links between data. Some datasets serve as linking hubs, containing many concepts which are also represented in more specific datasets. DBPedia is the main hub of the Linked Data Cloud. DBPedia is

an aggregation of the boxes shown on Wikipedia pages, containing facts about concepts.

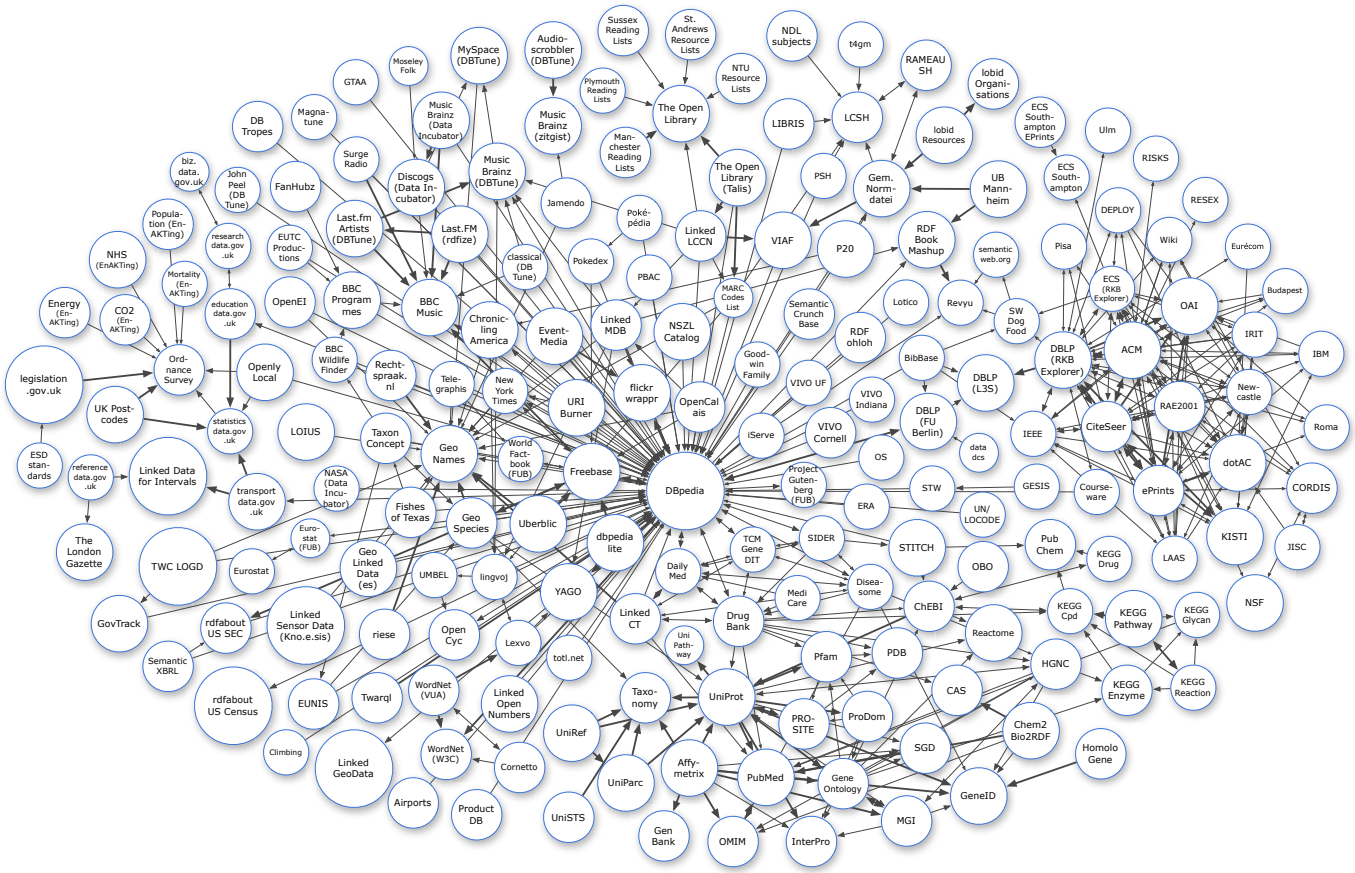


Figure 2.1: Linking Open Data cloud as of September 2010, showing datasets which have been published adhering to the Linked Data principles. By Richard Cyganiak and Anja Jentzsch (http://lod-cloud.net/).

2.1.3 Resource Description Framework

The main building block of the Semantic Web is the Resource Description Framework (RDF). RDF is used to make statements about resources. A single statement is called a triple and consists of a subject, a predicate and an object. An example of a triple, originating from the LinkedGeoData dataset, describing the statement that the construction of the Duomo of Milan was finished in

the year 1965 can be found in Figure 2.2. As one can see, the XML Schema data type integer is used, whereas date would have been the better choice. This clearly shows the freedom provided by the Semantic Web where each person can interpret, as long as a range is not specified, such properties as they like.

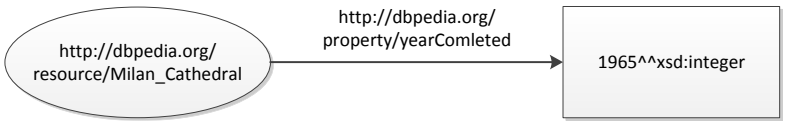


Figure 2.2: Triple representing the statement that the construction of the Duomo of Milan was finished in the year 1965.

The subject of a statement is a resource described by a URI or a blank node. A blank node is used when we want to make statements about something of which we do not now the exact identity. The predicate defines the relation between subject and object. On the Web, URLs are used to link sites, without actually specifying what sort of link it is. Semantic Web predicates are resources described by URIs and it is therefore possible to indicate the nature of the relation between subject and object. The object can be either a resource, blank node or a literal. Literals are atomic values, represented by a string. The sort of data type represented by the string can be defined using schemes, were XML Schema is the most widely used scheme.

There are multiple ways of representing RDF in textual form. RDF is, in the first place, designed to be readable for machines. This resulted in the RDF/XML serialization. RDF/XML uses XML to represent the triples, but is hard to read for humans. N-Triples represents triples in a more natural way, with the URI representations in subject, predicate, object order and ending with a full stop. The Terse RDF Triple Language (Turtle) is a superset of N-Triples, which uses abbreviations to make the RDF more readable for humans. Notation3 (N3), a superset of Turtle language, is further optimized in order to be a human readable language for data on the web.

2.1.4 RDF Schema & The Web Ontology Language

RDF Schema (RDFS) extends RDF with a minimal ontlogy representation language. RDFS has the expressivity to describe classes, subClasses and properties of classes [1]. A class hierarchy can be created using the attribute *subClassOf*. There is a similar attribute for properties: *subPropertyOf*. For properties the domain and range can also be set.

The need for more expressiveness in ontologies resulted in OWL, the Web Ontology Language [40]. OWL utilizes the RDF and RDFS structure and adds more options to describe properties and classes, such as relations between classes, cardinality, equality, richer typing of properties and characteristics of properties.

2.2 Large Knowledge Collider

This section is used to describe the Large Knowledge Collider platform (LarKC). In Section 2.2.1 we give a general introduction of the ideas behind LarKC and its architecture. LarKC plug-ins, which are the building blocks of LarKC, are discussed in Section 2.2.2. Workflows putting these building blocks together are described in Section 2.2.3.

2.2.1 LarKC Architecture

The Large Knowledge Collider (LarKC) is a platform aiming to enable massive distributed reasoning, using a combination of techniques from different research fields such as information retrieval, data mining and cognitive psychology. The platform enables developers to use small linked building blocks to achieve the desired functionality [6]. The algorithm shown in Figure 2.3 shows the original idea of the main loop used by the LarKC platform [20].

```
1: loop
2:     obtain a selection of data (RETRIEVAL)
3:     transform to an appropriate representation (ABSTRACTION)
4:     draw a sample (SELECTION)
5:     reason on the sample (REASONING)
6:     if more time is available and/or
7:     the result is not good enough (DECIDING) then
8:         increase the sample size (RETRIEVAL)
9:     else
10:        exit
11:    end if
12: end loop
```

Figure 2.3: The algorithm illustrating the initial idea of LarKC.

Over time it became apparent that this algorithm was too rigid for some of the use cases [5]. To provide developers with more flexible approaches, the decider plug-in got more control over the overall process, instead of only deciding whether the process should stop. Also the possibility of the creating complex workflows was added, which we will discuss in more depth in Section 2.2.3. A diagram originating from [5], describing the current architecture of LarKC can be found in Figure 2.4. The architecture has three domains: a platform domain, an infrastructure domain and a user domain.

The platform domain is a core part of the architecture. The LarKC *runtime environment* enables the initialization and invocation of the workflows and their plug-ins, through the use of an executor. The executor loads a workflow description, picks the appropriate plug-ins from the plug-in registry and executes the workflow. Executors can have endpoints, which enable communication with users through queries. The *management interface* allows users to submit their

workflows to the platform. The platform also manages the LarKC *data layer* in which the data is stored. The data layer is also used to enable commutation from one plug-in to the other using *SetOfStatemens* objects.

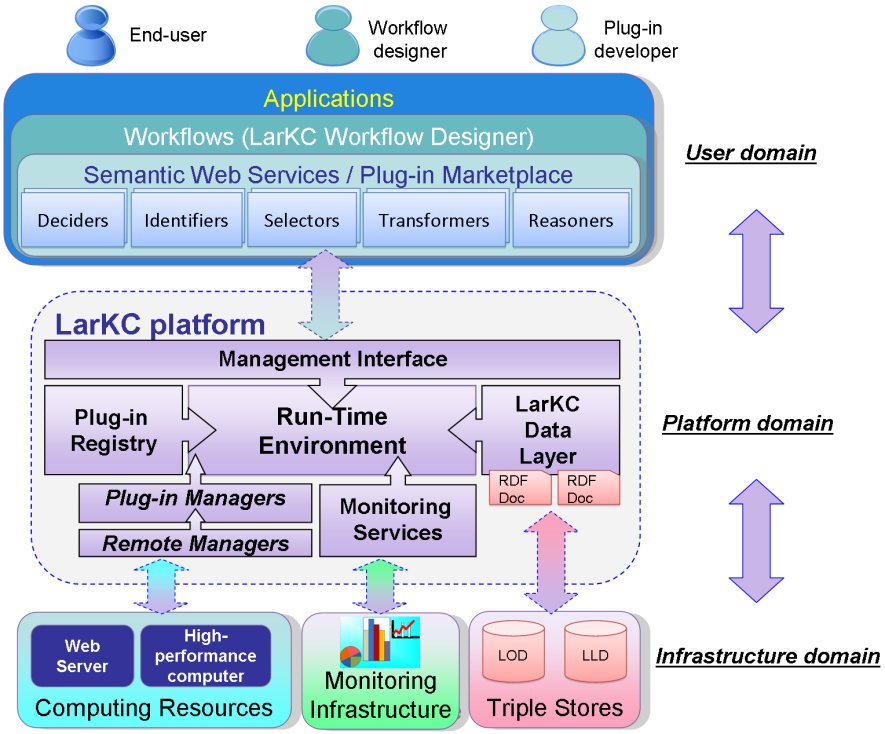


Figure 2.4: The high-level view of the LarKC architecture.

Two elements of the platform enable the correct execution of the plug-ins. The *plug-in registry* is used to manage the available plug-ins, so they can be retrieved whenever they are needed for the execution of a workflow. The *plug-in manager* manages the execution of the plug-in itself and enables the possibility of remote execution.

The infrastructure domain comprises elements supporting workflows running on the LarKC platform. LarKC's design is aimed to enable distributed computing, so when more *computing resources* are needed, it can be chosen to outsource some tasks to external systems, such as a web server or even a super computer. *Triple stores* can be used to externally store information. *The monitoring infrastructure* is added to be able to test the functioning of the LarKC elements.

Within the user domain, three distinct categories of users can be identified. *The end-users*, which are people who will use the applications running on LarKC. *The workflow designers*, which are developers constructing applications

by putting already existing plug-ins together using a workflow. *The plug-in developers* will design and create plug-ins as described in Section 2.2.2. In the following subsection we will elaborate more on two aspects of the user domain: workflows and plug-ins.

2.2.2 LarKC Plug-ins

The building blocks of a LarKC application are called plug-ins and can be reused or created from scratch. An on-line marketplace[1] containing the currently available plug-ins, can be used by developers to find plug-ins available for reuse. Tools, such as a wizard for the Eclipse development environment, are available to aid the process of plug-in creation.

The plug-ins utilized by the LarKC platform are mapped to the concepts introduced in the algorithm depicted in Figure 2.3. Although, each plug-in is essentially the same, there are some conceptual differences. Below, we will shortly discuss the different intended functions the plug-ins can have.

Retrieval
A plug-in that does retrieval of data takes a query as input and identifies SetOfStatement objects relevant to answering this query. In an urban computing scenario this could for example entail finding points of interest using DBpedia documents and instead of taking the complete database into account, only pointers to relevant documents are passed along to the next plug-in.

Abstraction
An abstraction plug-in can be used for two different tasks. The first task is the transformation of queries. When a query committed by a user is not appropriate for a given data source or reasoner plug-in, it can be altered so it matches the restrictions. Another option concerning queries is to construct multiple alternative queries from the original query.

The second type of abstraction plug-in is used to transform one data structure into another. A good example is converting structured info obtained through an API (usually provided in XML) into XML/RDF.

Selection
A selection plug-in is used for selecting appropriate parts of provided SetOfStatements objects, using a predefined strategy. A simple example is the GrowingDataSetSelecter, which initially selects a small part of the data and keeps expanding it over time, whenever there is more data available.

Reasoning
A reasoning plug-in is used to execute a SPARQL query against the data. The output of a reasoning plug-in depends on the reasoning task, in case of a select query, variable bindings containing the variables specified in the query are returned. If the SPARQL query is a construct or describe query, an RDF graph

[1] LarKC Plug-In marketplace `http://www.larkc.eu/plug-in-marketplace/`

is returned. In case of an ask query, a Boolean Information Set is returned indicating whether the pattern could be found in the SetOfStatements.

Deciding
As discussed before, a decider plug-in is generally used to influence the workflow process. A decider plug-in can be used to determine whether there are enough results obtained and whether they are of sufficient quality. If this is not the case, the plug-in can interfere.

2.2.3 LarKC Workflows

Workflows are used within LarKC to create a sequence of plug-ins. An example of such a sequence is the event pipeline used within the Alpha Urban LarKC application [47] [48]. This pipeline is graphically depicted in Figure 2.5. A workflow is described using RDF/XML or Notation3, using a specified vocabulary. The Notation3 representation of the event workflow can be found in Figure 2.6. Using the workflow description, the endpoint can be specified and it is possible to pass parameters from one plug-in to another.

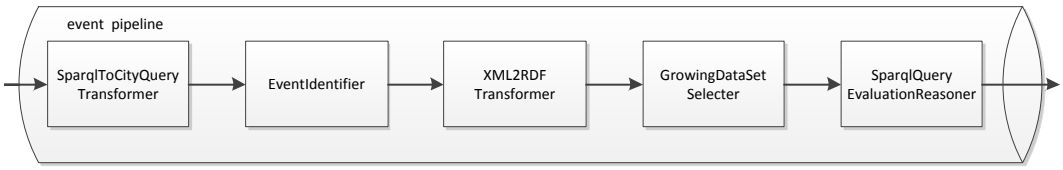


Figure 2.5: Graphical representation of the event pipeline workflow.

The event pipeline is used to retrieve events for a certain city, which is specified in a SPARQL query. The events are retrieved from the Eventful web-service[2] and because Eventful only accepts queries which are conform their specifications, the SPARQL query needs to be transformed. This is done by the SparqlToCityQueryTransformer. Next, the EventIdentifier is used to find the appropriate event data and stores a pointer for each found event. Since Eventful outputs its data in the XML format and our reasoner requires RDF data, the XML2RDFTransformer is needed to transform the results into the RDF format. Because the transformer only receives pointers, the XML data for the separate events have to be collected. The GrowingDataSetSelecter is used to select data for the SparqlQueryEvaluationReasoner.

Many of the plug-ins used in the event pipeline are generic, they can also be used in other workflows. This is one of the strong points of the LarKC

[2] `http://eventful.com/`

platform. When many people contribute generic plug-ins, some of the tasks will be as simple as writing a workflow, which will be able to accomplish its functionality using already existing plug-ins. The ease of changing the plug-ins within a workflow will also aid in testing different setups, using different plug-ins for the same task and testing which ones are better suited for the job.

```
1: @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
2: @prefix larkc: <http://larkc.eu/schema#> .
3: @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
4:
5: # Define plug-ins
6: _:SparqlToCityQueryTransformer a
7: <urn:eu.larkc.plugin.urbancomputing.SparqlToCityQueryTransformer> .
8: _:EventIdentifier a
9: <urn:eu.larkc.plugin.urbancomputing.EventIdentifier> .
10: _:XML2RDFTransformer a
11: <urn:eu.larkc.plugin.urbancomputing.XML2RDFTransformer> .
12: _:GrowingDataSetSelecter a
13: <urn:eu.larkc.plugin.GrowingDataSetSelecter> .
14: _:SparqlQueryEvaluationReasoner a
15: <urn:eu.larkc.plugin.SparqlQueryEvaluationReasoner> .
16:
17: # Connect the plug-ins
18: _:SparqlToCityQueryTransformer larkc:connectsTo _:EventIdentifier .
19: _:EventIdentifier larkc:connectsTo _:XML2RDFTransformer .
20: _:XML2RDFTransformer larkc:connectsTo _:GrowingDataSetSelecter .
21: _:GrowingDataSetSelecter larkc:connectsTo
22: _:SparqlQueryEvaluationReasoner .
23:
24: # Define a path to set the input and output of the workflow
25: _:path a larkc:Path .
26: _:path larkc:hasInput _:SparqlToCityQueryTransformer .
27: _:path larkc:hasOutput _:SparqlQueryEvaluationReasoner .
28:
29: # Connect an endpoint to the path
30: <urn:myQueryendpoint> a <urn:eu.larkc.endpoint.sparql> .
31: <urn:myQueryendpoint> larkc:links _:path .
```

Figure 2.6: The workflow description of the event pipeline of the Urban Alpha LarKC application, written in *Notation3*.

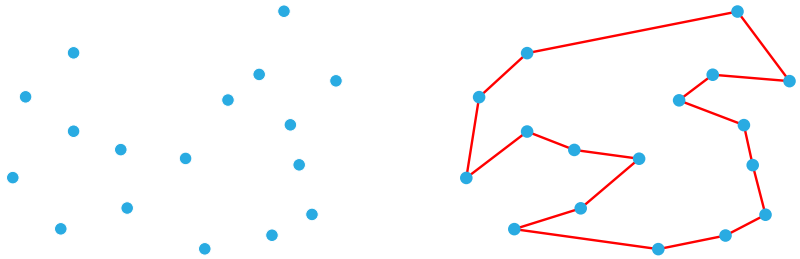
2.3 Traveling Salesman Problem

In this section, we discuss the well-known Traveling Salesman Problem. We introduce the problem and its history in Section 2.3.1. In Section 2.3.2 we

discuss a more constrained version that takes time constraints into account: the Traveling Salesman Problem with Time Windows. In Section 2.3.3 we conclude by introducing another constrained version, where the costs of traveling changes over time. This is the so-called Time Dependent Traveling Salesman Problem.

2.3.1 Generic Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is one of the most studied combinatorial problems. The TSP problem is a NP-hard problem and can be formulated as following: "Given a set of cities along with the cost of travel between each pair of them, the traveling salesman problem, is to find the cheapest way of visiting all the cities and returning to the starting point. The 'way of visiting all the cities' is simply the order in which the cities are visited; the ordering is called a tour or circuit through the cities" [3]. An example can be seen in Figure 2.7.



(a) TSP input (b) TSP output

Figure 2.7: A generic TSP example, where the figure on the left represents the input and the figure on the right the optimal solution.

The first notion of the TSP originates from the year 1832 in the book "Der Handlungsreisend – wie er sein soll und was er zu thun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiss zu sein – Von einem alten Commis-Voyageur", which described the explicit need for good tours for business men that were traveling around, in order to save time and resources. The term "traveling salesman problem" has emerged somewhere during the 1930s or 1940s, most likely at Princeton, and has been a widely used term since the 1950s [3].

Over time many solutions to the TSP have been proposed, making it one of the most studied problems in the fields of operational research and theoretical computer science. Also, researchers from related fields have shown interest in the problem. Examples are solutions that are based on evolutionary computing [22] and ant colony optimization [18].

We can conclude that the traveling salesman problem is a famous example of a well explored problem that has fascinated researchers for several decades, or even centuries.

2.3.2 Traveling Salesman Problem With Time Windows

The Traveling Salesman Problem with Time Windows (TSPTW) can be formulated as following: "the traveling salesman problem with time windows is the problem of finding a minimum-cost path visiting a set of cities exactly once, where each city must be visited within a specific time window" [21]. This problem statement is similar to the one of the regular TSP, with time windows as an addition.

A good TSPTW solver could be very useful when solving planning problems that have time related constraints. For example a parcel delivery company could use it to schedule the routes, where the nodes are customers which have to receive their parcel within a certain time window. The introduction of time windows makes this problem much harder to solve, since there are far less solutions than with the regular TSP problem. As with the regular TSP, exact algorithms as well as heuristic approaches can be found in the literature. We start by discussing some exact solutions that guarantee an optimal solution.

In [33], Langevin et al. introduce a two-commodity flow formulation that can be extended to the makespan problem. Dumas et al. use a dynamic programming approach in order to take advantage of the time window constraints to significantly reduce the state space and the number of state transitions [19]. Focacci et al. propose a hybrid approach for solving the TSPTW that merges constraint programming propagation algorithms and operational research techniques for coping with the optimization perspective [21]. Similarly, Pesant et al. propose a branch-and-bound solution using a constraint programming model [41].

Due to the limitations of the exact approaches in finding an optimal solution within a reasonable amount of time, the need for effective heuristics arises. A number of heuristic approaches have been proposed, some more successful than others. We discuss some of the most notable solutions in short and discuss one solution more extensively, which is the fastest solution up until now.

In 1997 Carlton and Barnes used a tabu search approach with penalty functions that also considered infeasible solutions [15]. In 1998 Gendreau et al. proposed a solution that gradually builds the route and once a feasible route is found, improves it with local optimizations [24]. In the year 2000, Calvo introduced an ad hoc objective function to obtain a solution close enough to a feasible solution of the original problem [14]. Given this solution, all the subtours are inserted into the main path using a greedy insertion procedure. More recently in 2007, Ohlmann and Thomas obtained very good results by using a variant of simulated annealing incorporating a variable penalty method [39].

The fastest solution currently known is proposed in [17] by da Silva and Urrutia. This paper proposes a two phase heuristic, where the first phase is a constructive phase that constructs a feasible solution using a variable neighborhood search (VNS) solution. The second phase is an optimization phase that optimizes the solution using a general VNS (GVNS) solution.

Since the proposed solution is a heuristic search, there is no guarantee that the optimal solution is found. However, from the results in the paper we can

conclude that the solutions that are found are generally very good and, in a number of cases, even optimal. More important is the computation time. While the results are slightly better than the, up until then, fastest compressed annealing method [39], the computation time is reduced by 77.68%. Up until now there has been no solution that outperforms the GVNS solution in the area of computation time.

2.3.3 Time Dependent Traveling Salesman Problem

The Time Dependent Traveling Salesman Problem (TDTSP) is generally stated as follows: "The time dependent traveling salesman problem can be seen as a generalized version of the classical traveling salesman problem where arc costs are dependent of their position in the tour" [25]. One can look at this as a regular traveling salesman problem in which the costs of traveling between nodes may vary over time.

Where the researchers on the subject of the TSPTW problem generally agree with each other on the formulation of the problem, this lies entirely different with the TDTSP. The time dependent traveling salesman problem was first formulated by Fox in 1973 [23], who illustrates it with examples from the brewing industry. The research of Picard and Queyranne [42] can be seen as the first major breakthrough in this field, they present the first effective method that is able to solve TDTSP problems of up to 20 nodes. Based on a formulation of a quadratic assignment model in [42], Gouveia and Voss modified the formulation and presented several, in their eyes, better and more concise formulations of the TDTSP. In [10] Bigras et al. investigate and evaluate various earlier proposed formulations of the TDTSP using a branch and bound algorithm.

Overall we can state that research performed in the field of the TSPTW is generally focused on the algorithmic solution, whereas performance improvements in the field of the TDTSP are reached through alterations in the formulation of the problem.

2.4 OpenStreetMap.org & LinkedGeoData.org

In this section we will discuss two closely related projects, OpenStreetMap.org and LinkedGeoData.org. OpenStreetMap is a project aiming to map the world with the use of user contributions. LinkedGeoData is a project using the data gathered by OpenStreetMap and transforms it into RDF. This makes it a valuable geographic data source for the Semantic Web.

2.4.1 OpenStreetMap.org

The aim of the OpenStreetMap project (OSM) is "to create a set of map data that is free to use, editable and licensed under new copyright schemes" [28]. The project has many aspects in common with Wikipedia. The content is user generated, user maintained and code for exploiting and editing the gathered

data is written by members of the community. A drawback of the openness of the OSM project is the lack of quality control.

The main website of OSM, as depicted in Figure 2.8, offers users a "slippy map", much like Google Maps, which can be used to browse the available geographical data. The export tab of the site provides users with the option to export information to a multitude of file formats. Other tabs provide information about added GPS traces, the changelog of the depicted area, user experiences in the form of user diaries and an editing tab which allows registered users to edit information.

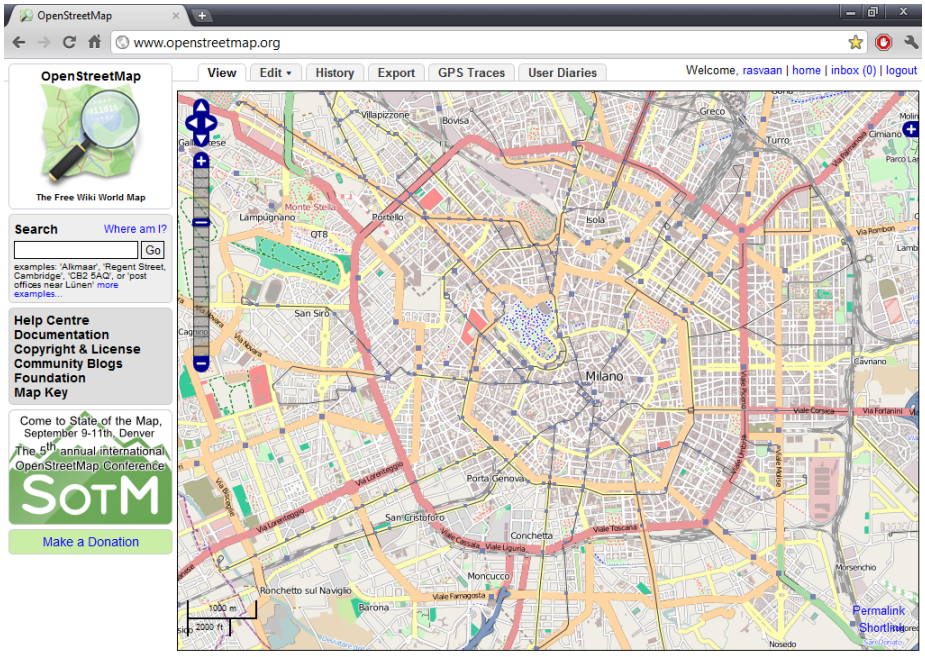


Figure 2.8: The website of the OpenStreetMap project.

One of the most important features of the OSM project is the ease of editing. Users can use the Flash-based Potlatch editor, or the more advanced Java OpenStreetMap Editor, to add, change or remove data. These editors enable users to upload detailed GPS information gathered using mobile GPS devices, with which the road networks are traced. Other sources used for enriching the data originate from publicly available data sources, such as the TIGER data in the U.S., or areal imagery, which is for example provided by Yahoo. Unlike Wikipedia, OSM only allows registered users to add information to the maps.

The underlying database is implemented in PostgreSQL. The geographical entities are stored as points, which are called nodes within the OSM community. Linear features of maps are called ways and are defined by a list of ordered nodes. Relations are used to group multiple nodes or ways together. Key-value

pairs can be added to each nodes, which can be used to annotate points of interest, for example **type=bar**. These pairs are called tags and are part of an extensive taxonomy, which is under continuous development by the OSM community. Table 2.1 shows information about the data statistics of the OSM project.

Category	Amount
Number of users	440.630
Number of uploaded GPS points	2.446.895.811
Number of nodes	1.157.997.265
Number of ways	102.564.798
Number of relations	1.061.454

Table 2.1: OpenStreetMap statistics as of July 2011

The map provided on the OSM website is not the only way to access the data. An Editing API is provided which can be used to fetch and save data from/to the OSM database. Every week a complete dump of the data is put on-line. The information is structured using a XML format and made available through a Bzip2 compressed file, with a current size of 17.1 GB.

2.4.2 LinkedGeoData.org

For the Semantic Web to function as a place for data integration, it needs large vocabularies to link data and information to, such as the DBpedia project. The LinkedGeoData (LGD) project aims to be the geographical counterpart of DBPedia, providing a dataset consisting of geographical data in the RDF format. The goal of the project is "to contribute rich, open, and integrated geographical data to the Semantic Web using OpenStreetMap as its base" [46]. The overall architecture of the system used by LGD can be found in Figure 2.9.

The data gathered by the OSM project serves as a basis for the LGD project. This data is transformed into RDF using the *LGD Dump Module*. This module utilises Java classes and XML snippets in order to convert the tags used within the OSM dataset into RDF. URIs for nodes and ways are represented by `lgd:node<id>` and `lgd:way<id>`. In order to be able to easily acquire the most appropriate type of a data instance, the property `lgdo:directType` is added.

The interlinking of knowledge bases is a key concept of publishing Linked Data. The central interlinking hub is DBPedia and the first aim of the LGD project is to link LGD instances to DBpedia instances using the `owl:sameAs` relationship [7]. The matching is accomplished using three criteria: type information, spatial distance and name similarity. Precision is preferred over recall, because of the far reaching consequences of a wrongly assigned `owl:sameAs` relation. In a new release of LGD, links were made to the GeoNames and Food and Agriculture Organization of the United Nations datasets.

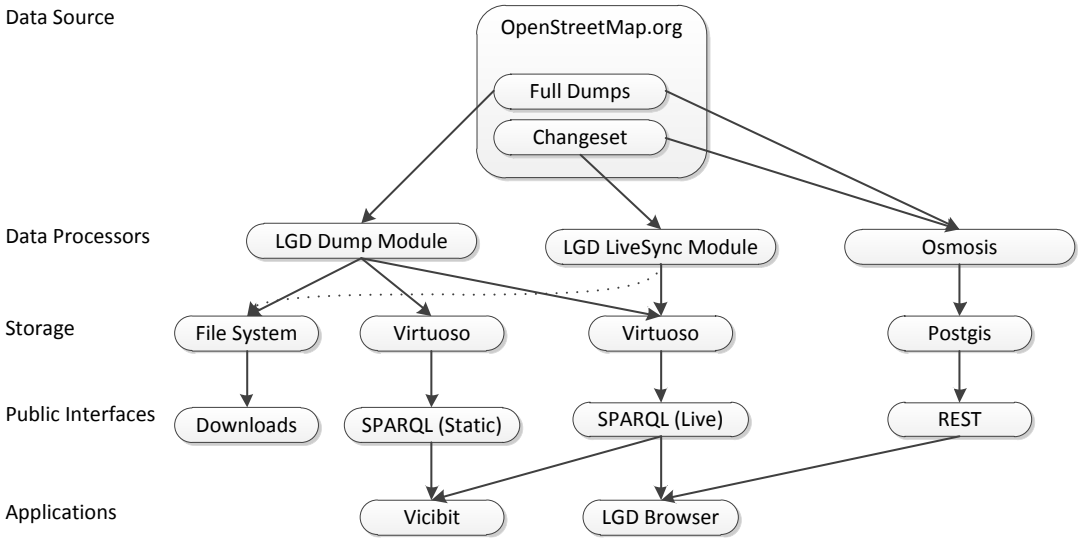


Figure 2.9: Overview of LinkedGeoData's architecture.

The converted data is uploaded to a file server, making it accessible for downloading. The data is also loaded onto a Virtuoso triple store, which makes it available for SPARQL querying, for applications such as Vicibit and the LGD Browser. A live and a static SPARQL endpoint are provided. The static one uses the full dumps of the OSM dataset, whereas the live one also incorporates the latest changesets. A REST service is also provided by the LGD project, which uses both the full dump and the changesets.

2.5 Electronic Guides

Electronic guides are applications running on computers or mobile devices, that can be consulted for travel information. While attempts are made to replace traditional guide books with electronic applications, the traditional guide books are still very popular. However, exact numbers cannot be given, because the sales figures of companies such as Rough Guide and Lonely Planet are not publicly available.

Most travelers trust and rely on the content of traditional guidebooks. A reason for preferring guidebooks over electronic guides might be the perceived quality of information. Most Guidebooks are edited with care, but will the information provided by their electronic counterpart be of the same quality? Another reason might be the need for an Internet connection, used by the applications to retrieve information. Due to high costs of data roaming, this can be very expensive for tourists who are traveling abroad.

While we do not have the exact ratio between the use of traditional guidebooks and electronic guides at our disposal, we can state with certainty that a

lot of headway can be made in the area of electronic guides. In order to gain a better perspective on the current state of the art concerning electronic guides, we take a look at two of the most important applications in this area. In Section 2.5.1 we discuss the web-service TripAdvisor and in Section 2.5.2 we discuss the mobile applications of Rough Guide and Lonely Planet.

2.5.1 TripAdvisor

On the about-us section ot the TripAdvisor website[3] the following is stated:

> "TripAdvisor is the world's largest travel site, enabling travelers to plan and have the perfect trip. TripAdvisor offers trusted advice from real travelers and a wide variety of travel choices and planning features (including Flights search, TripAdvisor Mobile and TripAdvisor Trip Friends) with seamless links to booking tools".

TripAdvisor presents itself as a website that tourists can use to plan their journeys. The provided information is mainly based on user generated content. This has also been confirmed by Miguéns et al. [37] in 2008, where a case study is presented in which the TripAdvisor data of Lisbon is analyzed. By examining the data that is available for this city, they come to the conclusion that the website is mainly used for the rating of hotels.

Besides the large number of hotels that can be found, Miguéns et al. also noticed that there were lots of restaurants and bars present and reviewed on the site. Information about other categories such as museums, monuments etc. was more scarce and of less quality. The article was published in 2008 and a lot has been changed since then. Therefore, we take a look ourselves and perform a small case study considering the city of Milan.

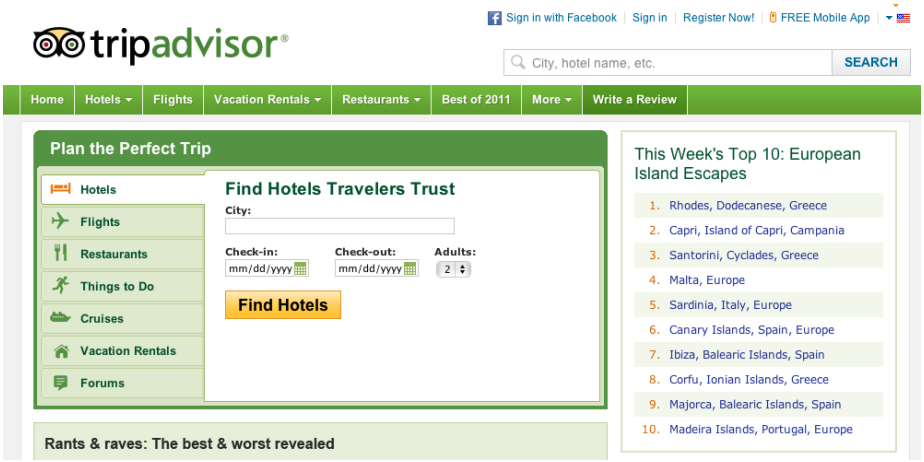


Figure 2.10: Part of the website of TripAdvisor.

[3] http://www.tripadvisor.com/pages/about_us.html

On the TripAdvsior homepage, as can be seen in Figure 2.10, a menu can be found that can be used to start searching, after one has selected a category. There are 3 categories that are interesting for us: hotels, restaurants and things to do. We start by taking a look at the hotel search, resulting in 434 found hotels, where most of the hotels have over 100 reviews. This is an impressive result, for a medium sized city such as Milan.

The number of 1507 returned restaurants is also impressive, but what really stands out is the amount of detail of the information provided by the results. One can make a selection based on cuisine, neighborhood, scene and price range.

When looking at the amount of attractions returned for the city of Milan, we can conclude that this feature has improved since 2008. As far as we can see, all the major touristic attractions, such as the Duomo and the Galleria Vittorio Emanuele, are present. The total amount of attractions found adds up to 222.

From our small case study it can be concluded that the TripAdvisor site hosts a lot of high quality travel information, which is primarily maintained by a dedicated group of users.

2.5.2 Lonely Planet & Rough Guide City Apps

The Lonely Planet[4] and Rough Guide[5] companies have released similar apps that are able to guide tourists through cities. Both apps are mobile applications that can be used in combination with mobile devices running iOS or Android. The information in guide books such as the Lonely Planet books and the Rough Guide books is often of high quality, due to the fact that it is written by professional reporters. Incorporating this information into mobile apps has several advantages.

First of all, tourists do not have to carry a heavy guide book with them all the time, but instead only have to bring a smartphone or a tablet device, such as the iPad. Another advantage is that the travel information is stored on the device, so that there is no need for an Internet connection. For each city a separate app has to be released. Rough Guide currently released apps for 6 cities, while Lonely Planet has released apps for over 100 cities.

The main feature of both apps is a map showing all the major sights in the neighborhood of the tourist's current position (determined using GPS). The sights are divided into several categories ranging from attractions to restaurants. A feature that is only available in the Rough Guide apps is the ability to suggest things to do for indecisive users. Traditional guide books lack the possibility of implementing this feature. Another feature is the possibility for creating lists of POIs, which one wants to visit.

Overall we can say that both the city apps are interesting for tourists. They have an added value with respect to traditional guidebooks by doing things such as showing ones current location using GPS and suggesting places where the user could go next.

[4] http://www.lonelyplanet.com/mobile/
[5] http://www.roughguides.com/apps/cities/

2.6 Automated City Trip Planners

In the last decade, a range of applications have been developed supporting tourists in their decision making. An overview of systems that compose tours of POIs to visit, is presented in [45]. Souffriau et al. conclude their overview article with the following sentence: "Providing adequate tour scheduling support for tourist decision support applications is a daunting task for the application developer". Developers take different approaches to solve the problem. In this section we will discuss the most influental ones.

The Dynamic Tour Guide by ten Hage et al. is the first mobile agent able to compute tours of POIs on the fly [27]. It uses Tour Building Blocks (TBB) to compose appropriate sequences of POIs. The TBBs are structured using an ontology and are matched with the user preferences. Search iterations using a directed depth first algorithm are performed for five seconds, after which the best tour is presented to the tourist. One of the stated benefits of the automated tour guide is the possibility of achieving a higher dispersion of tourists over the attractions.

P-Tour, a personal navigation system for tourists is introduced in [36]. A destination list is used to offer tourists the possibility of selecting interesting POIs or they can choose to add destinations themselves, through the input of latitude and longitude values. Users need to assign importance degrees and time windows to POIs. This information is used to pick an appropriate number of POIs and generate a route using a variant of the TSP. The algorithm is able to generate solutions within 5 seconds, for a dataset of 30 POIs.

An extension of the P-Tour system is developed by Kinoshita et al. enabling multiple day tours, through partitioning of the dataset [32]. Wu et al. extend the P-Tour system in such a way that it takes the weather forecast into account [51]. The ratings of the POIs are dependent on whether it is fine, cloudy or rainy weather. Nagata et al. add functionality to handle the sometimes conflicting interests of group members on a sightseeing trip, using a Genetic Algorithm-based algorithm [38].

A multi-agent ontological recommendation application for the city Tainan in Taiwan, is developed by Lee et al. [34]. This system includes a context decision agent and a travel route recommendation agent. The first agent uses fuzzy inference mechanisms to match the requirements of the tourist to the developed ontology, resulting in eight selected POIs. The second agent retrieves contextual information, generates a route using a TSP algorithm and plots this on a Google map instance.

As opposed to the earlier mentioned applications, which make a distinction between the selection of POIs and generating a route between them, Vansteenwegen et al. advocate an integrated approach, using a Team Orienteering Problem with Time Windows algorithm (TOPTW). "In the TOPTW, a set of locations is given, each with a score, a service time and a time window. The goal is to maximise the sum of the collected scores by a fixed number of routes. The routes allow to visit locations at the right time and they are limited in length." [50] Trips consisting of multiple days are made possible by generating multiple tours,

visiting different POIs (the *Team* part of TOPTW).

A web application utilizing this TOPTW algorithm is implemented for five cities in Flanders, Belgium[6]. The dataset consists of POIs selected by the local tourist offices and contain at most 216 POIs. POI descriptions are provided in English and Dutch. Usage statistic were collected and in two months 17.510 unique visitors visited the site, generating 20.395 trips. The collected user feedback was positive.

The city trip planner we develop is more similar to the city trip planners that have separate components for selecting POIs and route generation than to the all in one solution by Vansteenwegen et al. Despite the similarities in architecture, there are areas in which our application differs from the described automated city trip planners. In the following paragraphs, we compare the current state of the art with our application.

As most applications, the selection of POIs is separated from the routing algorithm. The POI selection is, first and foremost, based on the user preferences, but when the overall time window allows it, our application will add additional POIs to the plan that are not specified by the user. For the routing component, a TSPTW algorithm taking the time constraints of POIs into account is used.

The main point on which we distinguish ourselves, is the data we use. As opposed to the above solutions, that all use closed datasets, we have much more data at our disposal by using Semantic Web resources. Also, because of the earlier mentioned Linked Open Data cloud, we can provide tourists with rich semantics by using the links that exist between the different Semantic Web data sources, in order to retrieve more information.

The number of POIs resulting from these Semantic Web sources is considerably larger than the datasets used by the other applications. This makes the process of finding appropriate sets of POIs in combination with determining good routes much harder. This results in the need for good selection strategies, whereas other applications can just generate all the possible solutions and pick the best one.

We use the Semantic Web platform LarKC to achieve the overall functionality, which has the advantage that the components of the application can be easily interchanged. When we decide to use a different point selection strategy or TSP solver, this can easily be achieved by creating the corresponding plug-in and incorporating it in the LarKC workflow. In the future, the platform will also simplify the integration of additional data sources.

Similar to the Dynamic Tour Guide by Ten Hage et al., we also develop an ontology, but instead of using the concept tour building blocks, we use POIs. We assign the different kinds of POIs to classes and subclasses in order to create a useful hierarchy.

[6] `http://www.citytripplanner.com/`

Chapter 3

Problem Setting

In this chapter we introduce the problem by providing the reader with an illustrative scenario of the use of the application in Section 3.1. User requirements following from this scenario are given in Section 3.2. A formal description of the problem space can be found in Section 3.3.

3.1 Scenario

In this section a scenario is given, which will illustrate the future use of the city trip planner application. In the consecutive chapters we will use this scenario to clarify the theory and strategies. The general process of using the system (and thus of any scenario) can be found in Figure 3.1.

A tourist is visiting Milan for the first time in his life. After a demanding flight with EasyJet (it is cheap, but you can not count on them being in time at your destination), he has barely enough energy to get to his hotel. Using a scrap of paper with the appropriate public transport lines, he manages to get near his destination. Using the Google maps app on his phone he decides on which way to go. Finally arrived at the hotel, he has not got the energy to go into town and goes straight to bed.

A little overslept the tourist gets out of bed at 11 o'clock in the morning. Unable to get breakfast in the hotel, he grabs something to eat in the cafe opposite the hotel. Two days left in the inspiring city, so much to do, so much to see. But were to go first? Instead of spending the rest of the morning planning a way through the city, he opens an app on his phone while nourishing the great taste of Italian coffee in the bar next to the hotel. The app will generate a route through the city, visiting some of the most beautiful sites.

He has got multiple options of how to proceed. The simplest is to just let the application create a generic route, using the centre of the city as a starting point. Another option is to choose the start point himself, traversing multiple categories like museums, hotels and monuments until he finds the starting point he likes. He chooses the third option though, just selecting the point on a

displayed map. He selects the central square of the city, Piazza del Duomo.

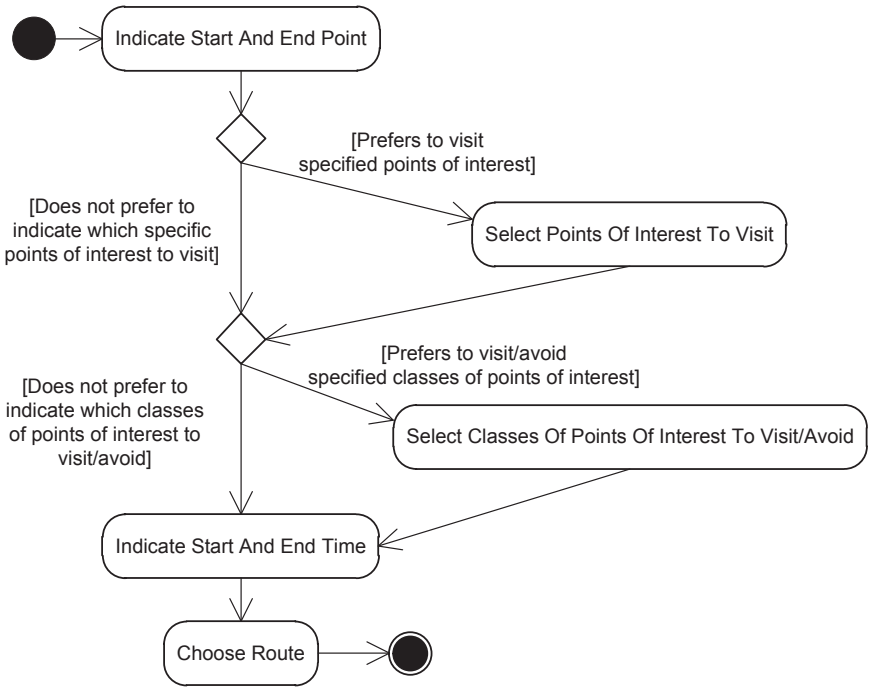


Figure 3.1: UML Activity Diagram describing the use of the system.

From a list of the main points of interest he selects the Vittorio Emanuele monument, who was the first king of united Italy. He also adds the category Church to his preferences. When you visit the country that is home to the Roman Catholic Church, you should at least visit some of the beautiful churches right? He also adds the preference to have dinner at some point in time, but no fast food.

Indicating that he wants the planning to start at 2 p.m. and eventually return around 12 p.m. at the same point he starts from, the app indicates it is calculating routes. Within a minute it displays three routes. All of them include multiple alternative POIs in addition to the ones he selected. Dinner is also planned!

He chooses the first route and a graphical representation of a route is presented by the app, displaying all the sights he is about to visit. The sights all have a rating and it appears that only the ones with a high rating are selected. The application changes to map view, which displays a map with a route projected on it. He finally manages to swallow the last bit of croissant and sets out to visit the first POI.

3.2 User Requirements

Below we provide a list with the requirements that the application will have to fulfill in order to be able to accomplish the results described in the scenario of Section 3.1.

- The user **can** select one or more specific POIs to include in the trip.
- The user **can** select one or more categories of POIs to include in the trip.
- The user **can** select one or more categories of POIs to exclude from the trip.
- The system **must** propose a trip that satisfies the users request.
- The system **may** add additional POIs to fill up the remaining time.

3.3 Problem Formalization

This section will describe the conceptual framework. It starts with the introduction of a plan for a city trip and the main concepts to realise such a plan. Equations used to evaluate the found plans are given next.

Let G be the graph containing POIs and edges between these POIs, where the edges represent travel distance. $\mathcal{UI}$, $\mathcal{UC}$ and $\mathcal{NUC}$ are sets containing user preferences where $t^{start_{plan}}$ and $t^{end_{plan}}$ indicate the start and end time of the plan. P is an ordered list of POIs defining a plan for a city trip. The process of finding a suitable plan can be formulated as a function F (Equation 3.1).

$$F(G,\mathcal{UI},\mathcal{UC},\mathcal{NUC},t^{start_{plan}},t^{end_{plan}}) \rightarrow P \tag{3.1}$$

Graph G is defined as $G = (\mathcal{S},\mathcal{E})$, where $\mathcal{S} = \{POI_1 \dots POI_n\}$ is the set of all POIs contained in the problem space, and $\mathcal{E} = \{(i,j) : i,j \in \mathcal{S} \cup \{POI_{start}, POI_{end}\}, i \neq j\}$ represents the edges between the POIs. POI_{start} is the starting point of the planning and POI_{end} the end point. Both points are retrieved from the user input. Let $t_{i,j}$ be the travel time from i to j in seconds. The graph G of Milan could, for example, have POIs *Duomo*, *LaScala* $\in \mathcal{S}$. Traveling from La Scala to the Duomo takes 6 minutes, so $t_{duomo,laScala} = 360$.

Each $POI_i \in \mathcal{S}$ is described using a tuple $< r_i, [a_i, b_i], d_i, C_j >$. A POI has an associated rating $r \in [0,1]$ that describes the relevance of the POI. A time window specific for a POI_i is defined by $[a_i, b_i]$, where a_i is the start time and b_i is the end time. The time window sets the period of time in which the POI is relevant to a user. Each POI_i has a nominal average duration d_i, which is the time in minutes a tourist will probably spend at the sight. Each POI_i is also a direct instance of a class C_j. The classes are disjoint, which means that a POI can only be a direct instance of exactly one class.

An example of a POI is the Duomo of Milan. It is one of the major visitor attractions of Milan, so it has a rating of 0.87. A tourist can enter the Duomo

from 7:00 to 18:45, which corresponds to the a_{duomo} and b_{duomo} value. A tourist will probably spend around one hour at the Duomo, which results in a d_{duomo} of 60 minutes. The Duomo is an instance of the church class.

There are three sets of user preferences which our trip planner needs to fulfill, $\mathcal{UI}, \mathcal{UC}$ and $\mathcal{NUC}$. $\mathcal{UI} = \{POI_1, \ldots, POI_k\}$ represents the instance preferences. This set is used to give the user the option to indicate specific POIs that he/she will definitely visit. Examples are the Duomo in Milan or the Colosseum in Rome. $\mathcal{UC} = \{(C_1, X_1), \ldots, (C_m, X_m)\}$ represents the class preferences. C_i represents the class and X_i the amount of times a tourist likes to visit POIs categorized with C_i. This way a user can for example indicate the need to visit three pubs, $\mathcal{UC} = \{(Pub, 3)\}$, without actually selecting the specific instances. $\mathcal{NUC} = \{C_1, \ldots, C_p\}$ represents the non preferred classes. With the help of this set a user can specify which classes he or she definitely would not like to visit.

The POIs in $\mathcal{UI}$ do not affect the set $\mathcal{UC}$, this implies that when a user indicates to appreciate visiting a class C_i and also adds a POI_j to $\mathcal{UI}$, which is an instance of this class, this will not effect X_i.

Furthermore, all the user preferences should be fulfilled, so only solutions adhering to the inclusion of a number of instances of C_i equal to $X_i + |\{POI_j \in \mathcal{UI}\} \cap \{POI_j \; instance \; of \; C_m\}|$ are valid solutions.

We define a plan as $P = (< POI_1, t^{start_1}, t^{end_1} >, < POI_2, t^{start_2}, t^{end_2} > , < POI_l, t^{start_l}, t^{end_l} >)$ of length l that satisfies the following constraints:

For plan P:	For each POI $< r_i, [a_i, b_i], d_i, C_j >$
$t^{start_i} \geq a_i$	$t^{start_1} > t^{start_{plan}}$
$t^{end_i} \leq b_i$	$t^{end_l} < t^{end_{plan}}$
$t^{end_i} - t^{start_i} <= d_i$	

The start and end time of a plan, $t^{start_{plan}}$ and $t^{end_{plan}}$, are specified by the user, who, for example, can start the planning at 2 in the afternoon and end it at 12 in the evening.

An implication of the estimated duration d_i of points is that there will be a lower and upper bound to the number of points that can be visited in the time interval between $t^{start_{plan}}$ and $t^{end_{plan}}$. This can be seen in Equation 3.2.

$$\frac{t^{end_{plan}} - t^{start_{plan}}}{Maximum(d_i)} \leq l \leq \frac{t^{end_{plan}} - t^{start_{plan}}}{Minimum(d_i)} \tag{3.2}$$

Multiple plans which satisfy all constraints are normally expected. Only the best solutions should be presented to the user. In order to determine which plans are good and which plans are not, we rank the plans using two metrics. One that judges the quality of the POIs that are incorporated in the selection and one that judges the travel time of the route. In equation 3.6 we define the quality of the selected POIs as a summation of the score of each separate POI, divided by the total time interval between $t^{start_{plan}}$ and $t^{end_{plan}}$. The POI score described by Equation 3.3, is influenced by the way it matches the user preferences, the general relevancy of the POI (the rating r) and its service time

d. The different types of POI matching are illustrated in Figure 3.2. When dealing with instance preferences, there is the possibility of a perfect match or a non-match and when dealing with class preferences, we have the possibility of a perfect match, subsume match and non-match/plugin match (Equation 3.4). The score of POI_i can then be calculated as following:

$$POIScore(POI_i) = r_i \cdot d_i \cdot matchScore \tag{3.3}$$

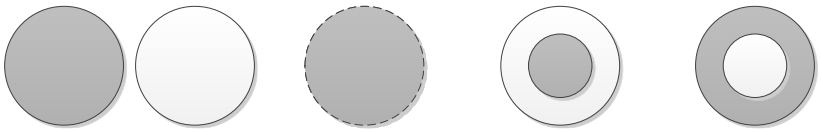
$$matchScore = \begin{cases} 1 & \text{if perfect match} \\ 0.66 & \text{if subsume match} \\ 0.33 & \text{if non-match or plugin match} \end{cases} \tag{3.4}$$

$$ServiceTime(P) = \sum_{i=1}^{n} d_i \tag{3.5}$$

$$Score(P) = \frac{\sum_{i=1}^{n} POIScore(POI_i)}{t^{end_{plan}} - t^{start_{plan}}} \tag{3.6}$$

In the case of a non-match or a plugin match, the POIs are still multiplied by 0.33. We choose to do this, because this will increase the score when there are more points included in plan P. Algorithms selecting points for P should not stop when the user preferences are fulfilled and there is still time left to visit additional POIs. At those moments, other POIs can be added, to increase the overall rating and make the user experience points he/she had not thought of before.

In the POIScore we multiply the rating r and $matchScore$ by the service time d. In the score function the sum of POI scores is divided by the size of the overall time window defined by the user. This is to ensure that plannings with multiple mediocre points with a small r and a small d, will not outweigh a great POI with a high r and a high d. So ten shops with a rating 0.5 and duration of 10 will not outweigh a museum with rating 0.9 and duration of 120.



(a) Non-Match (b) Perfect Match (c) Subsume Match (d) Plugin Match

Figure 3.2: Different types of matching between classes.

The total travel time computed by Equation 3.7, is judged by Equation 3.8 in comparison to the maximum and minimum travel time. The maximum travel time is defined by the user through $t^{end_{plan}} - t^{start_{plan}}$. The minimum travel time will always be 0, this is the case when a POI is at the same place as the next POI. Therefore, we can rewrite equation 3.8 into 3.9. The travel time score

is scaled between $[0, 1]$ where 1 means that the user will not travel at all and 0 means that the user will only be travelling, without seeing any POI.

$$TravelTime(P) = \sum_{i=0}^{l} t_{i,i+1} \tag{3.7}$$

$$Time(P) = 1 - \frac{TravelTime(P) - minimumTravelTime}{(t^{end_{plan}} - t^{start_{plan}}) - minimumTravelTime} \tag{3.8}$$

$$Time(P) = 1 - \frac{TravelTime(P)}{c_1 \cdot (t^{end_{plan}} - t^{start_{plan}})} \tag{3.9}$$

The rating of any planning P can be calculated by using the grading function in Equation 3.10, where $K_{eagerness}$ and $K_{laziness}$ are constants between $[0, 1]$. When $K_{eagerness} \gg K_{laziness}$ the grading function will tend to care less about the travel time, resulting in plans more appropriate for active people, who want to see the POIs best fitting their preferences and with the highest rating. In case of $K_{eagerness} \ll K_{laziness}$, a low travel time is very important, which will result in plans more appropriate for lazy people, at least people who do not appreciate to travel a bit in order to see great POIs.

$$Rate(P) = K_{eagerness}\ Score(P) + K_{laziness}\ Time(P) \tag{3.10}$$

Examples of city trip plans corresponding to the user requirements of the scenario of Section 3.1 can be found in Section 8.2.3. These plans adhere to all the constraints specified in this chapter.

Chapter 4

Architecture

The overall system architecture, as used within the LarKC environment, is described in Section 4.1. In Section 4.2 we discuss the conceptual model of our main data type, the point of interest. Section 4.3 describes the way of modelling the distances between the points of interest. A description of the ontology used to structure the data can be found in Section 4.4. The plug-ins used to accomplish the overall functionality are discussed one by one in Section 4.5. The frontend, which is not part of LarKC, is discussed in Section 4.6

4.1 System Architecture In LarKC

The high-level architecture of our LarKC application is graphically depicted in Figure 4.1. The Planning Decider controls and loads the workflows, the Point Selecter generates $\mathcal{T}$-sets, the TSPTW Reasoners makes plans P, the RankDecider grades and ranks the found plans and the Cartographer generates output files of the best plans.

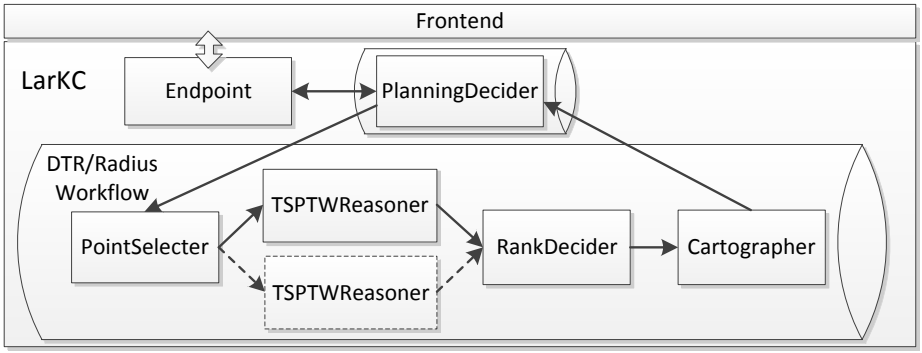


Figure 4.1: The high-level system architecture in LarKC.

The Planning Decider is the first plug-in loaded by the platform and controls the overall process of generating city trip plans. On start up, appropriate POI data is loaded into the data layer in addition to the user preferences. The plug-in is connected to the SPARQL endpoint and receives the queries, which it uses to query the data layer of LarKC after a successful city trip generation iteration. The number of desired plans is communicated to the Point Selecter plug-in.

The Point Selecter determines multiple sets of points of interest according to a specified strategy, considering the graph G, $t^{start_{plan}}$, $t^{end_{plan}}$ and the user preference sets $\mathcal{UI}$, $\mathcal{UC}$ and $\mathcal{NUC}$. The amount of generated sets depends on the number of plans desired by the user. The computed sets $\mathcal{T}$ are used in the following process, the TSPTW Reasoner.

The TSPTW Reasoner retrieves the weights of the edges between the points from graph G and uses this combination of points and weights to compute the optimal routes. As discussed in Section 2.3.2, this is computationally speaking a very complex process, making this plug-in the bottleneck of the workflow. In order to speed up the process, we decide to enable the parallel execution of this plug-in, using a LarKC workflow parameter.

The output of the TSPTW Reasoner is a list P, containing a specific sequence of POIs. These plans are graded by the RankDecider using the function shown in equation 3.10. The plans are ranked and send to the next plug-in, the Cartographer.

The Cartographer takes the plans P as input, computes the detailed routes between the POIs and patches them together. This results in a representation of a route, a list of path information objects. This list could be used to graphically show the route on a map, or for navigating between the different POIs. The overview of the different plans P is presented to the user by the Planning Decider.

4.2 Modelling Points Of Interest

The LarKC plug-ins require data in order to function correctly. The primary kind of data used by the application are geographical points which might be of interest to a tourist. These points of interest (POIs) can be all sorts of entities. Examples are churches, squares, shops, restaurants and museums. A depiction of the general data structure can be found in figure 4.2. The Duomo of Milan is depicted in 4.3, according to the proposed data structure. All the LarKC components use information represented by these points in one way or another.

The POIs should include latitude and longitude values in order to determine their position respective to one another. Also, a time window $[a, b]$ is important, where a is the opening time and b the closing time. The indication of the time d in minutes can be seen as a duration description: the time a tourist will likely spent at the given POI.

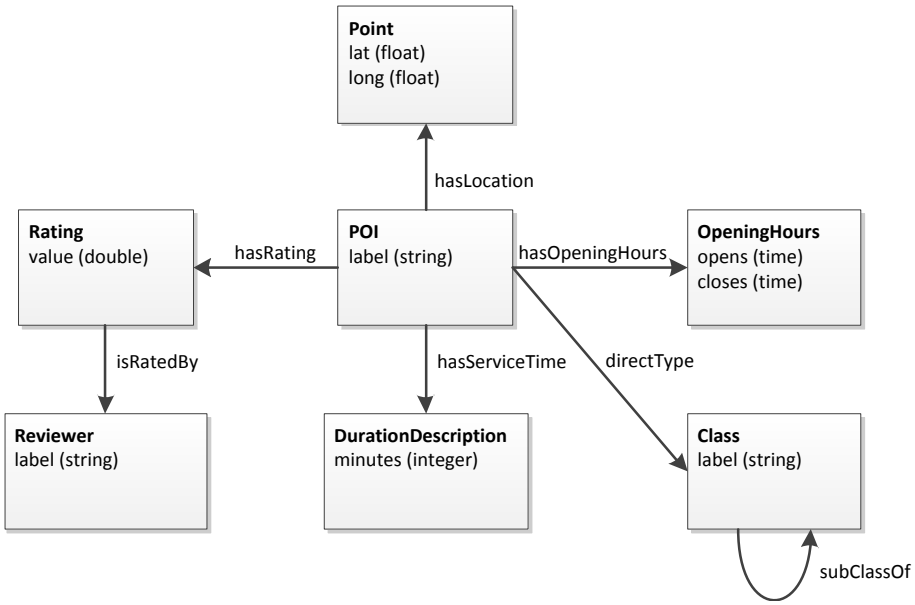


Figure 4.2: Graph depicting the information linked to a point of interest.

Heuristics can be used to determine probable time spent at different sorts of POIs: a tourist will probably spent less time visiting a square than a museum. A rating value r will indicate how interesting a certain POI is, which can be used to distinguish interesting POIs from less interesting POIs.

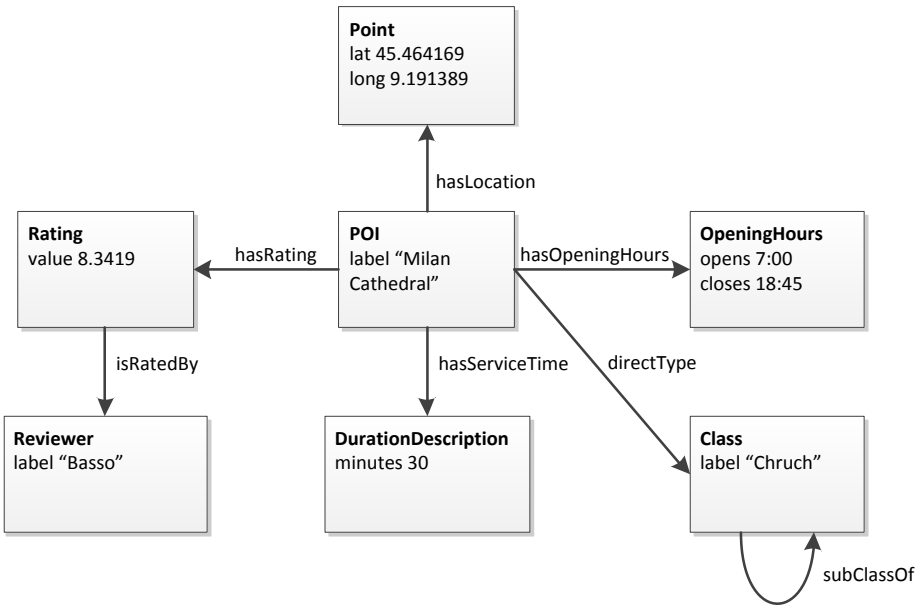


Figure 4.3: Graph of the data linked to the Duomo POI.

4.3 Modelling The Distance Between Points Of Interest

The TSPTW solver needs a set of points, the location of the points, the time windows of- and probable time spent at the respective points. These things can all be retrieved from the data linked to the POIs. Data that has to be deducted are the weights of the edges in $\mathcal{E}$. The most trivial choice for the weights is the distance between two points, but one could also think of more complex weights incorporating the comfort of the journey between two points, causing the most pollution or whether it is a beautiful path or not.

The deduction of distance could be done by an external service, for example Google Maps. Another option is doing it locally, using for example the OpenStreetMap dataset in combination with path finding algorithms. At this point of development of the application we decide to determine the distance locally and not to use a routing algorithm. For simplicity, we determine the Euclidean distance on the fly. This will still enable the strategies and TSPTW to utilize realistic distances between the points.

4.4 Categorizing Points Of Interest

In order to effectively store data about points of interest and doing this in a way that makes the information easily retrievable, we have to store the data in a structured format. A good way of structuring information is by using an ontology to define an hierarchy between concepts within our data. Using the ontology we can structure the data by assigning individuals to a node of the corresponding class. This structure can then be utilized to find better fitting activities to include in plan P, a tourist could for example specify to only go to points belonging to the nightlife classes while avoiding the cultural attractions.

Considering the ontology, we have two options: creating our own ontology that includes all of the concepts, relationships and properties we need or, reusing an existing ontology that already describes everything we need. Since creating a conceptual valid ontology is a complex and time consuming task, we would prefer to reuse an existing ontology. Reusing ontologies is also a key concept of the Semantic Web, it ensures that within different systems the same definitions of concepts are used.

One of the candidate ontologies for reuse is the ontology used for structuring the LinkedGeoData dataset. This ontology is automatically deducted from the OpenStreetMap data using a configuration file [7]. The ontology contains concepts that are defined more than once, which makes the ontology not appropriate for our application. For example, the ontology contains the concept Tourism with a subclass TourismMuseum and a class Historic with the subclass Museum. Another example is the concept Hotel, the Tourism class has a subclass TourismHotel as well as a subclass Hotel.

The LinkedGeoData ontology is relatively shallow, where the selection process of POIs could benefit from an extensive subclass structure. For example, the

class Tourism has the subclasses Hotel and Hostel, where a division of Tourism with subclass Places_To_Sleep with, on its turn, subclasses Hostel and Hotel would have been better in our case.

A more detailed taxonomy, with multiple levels of classes, would enhance the functioning of the City Trip planner. This is why we decide to create our own ontology, using the editor Protégé[1]. The resulting ontology is a taxonomy of classes, a part of it is depicted in Figure 4.4. All the classes are displayed in Appendix A.

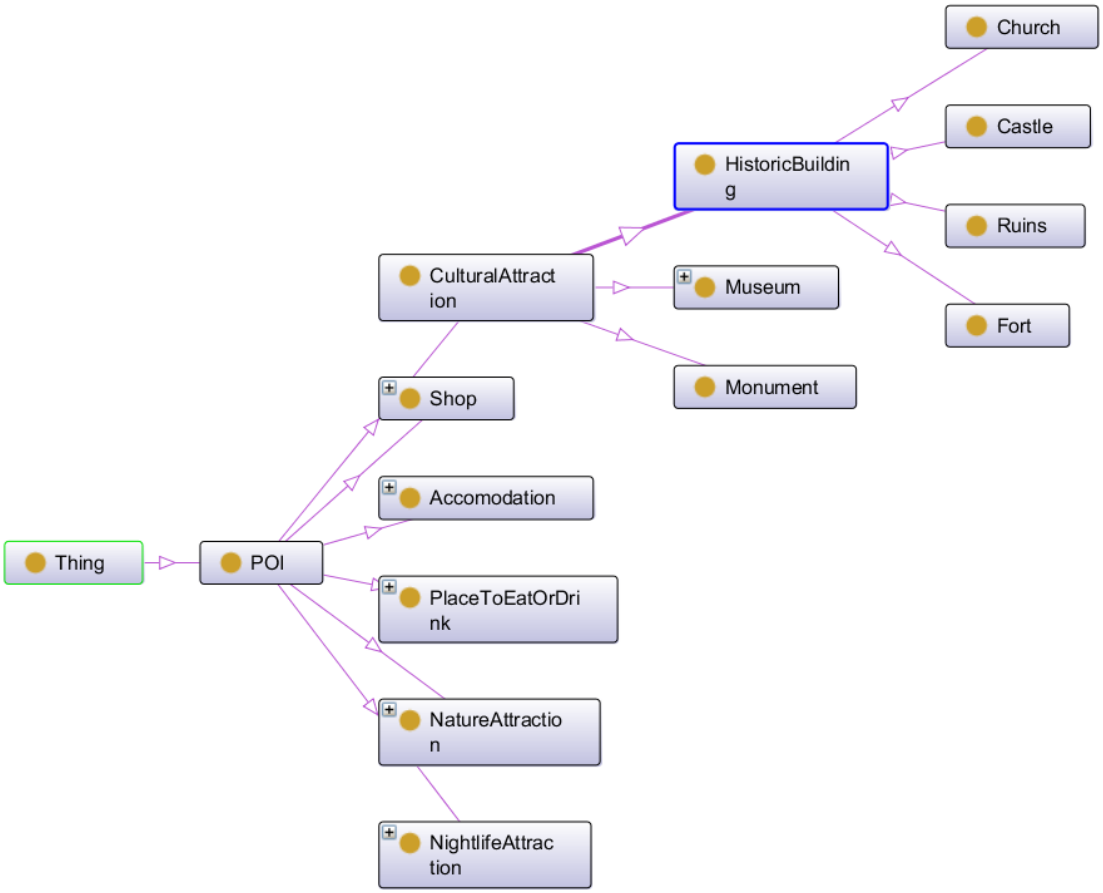


Figure 4.4: A part of the ontology, with the branch of the HistoricBuilding completely expanded.

In Figure 4.4 all the subclasses of HistoricBuilding are shown. When, for example, a tourist indicates the desire to visit a historic building, the application now not only considers visiting the points of interest with the class historic

[1] Available at: `http://protege.stanford.edu`

building, but also its subclasses Church, Castle, Ruins and Fort.

4.5 LarKC Plug-ins

Here we describe each individual LarKC plug-in on a conceptual level. We start with the Planning Decider in Section 4.5.1. We continue with the Point Selecter in Section 4.5.2. In Section 4.5.3, we describe how the selected sets are turned into city trip plans by the TSPTW Reasoner. The plans are judged by the Rank Decider as described in Section 4.5.4 and we conclude with the Cartographer in Section 4.5.5.

4.5.1 Planning Decider

As can be seen in the system overview of Figure 4, the Planning Decider loads the workflow which is used to accomplish the city trip planning generation, using either the DTR strategy or the Radius strategy. It conducts the creation of the desired number of plans and communicates with the frontend using the query endpoint. The Planning Decider is the first plug-in initialized by the application. When the Planning Decider is initialized, it waits for an incoming query from the endpoint until taking further action.

There are two strategies available for the point selection process and depending on the users preference, one of the strategies is loaded through its corresponding workflow. At the moment a query is submitted, the workflow is loaded and the city trip generation process starts with the Point Selecter. When the last plug-in, the Cartographer returns its results, the Planning Decider uses the SPARQL query obtained from the endpoint on the received data and outputs the query results to the endpoint.

4.5.2 Point Selecter

The main type of input needed by the TSPTW Reasoner is a set $\mathcal{T}$, a set of points for which it will estimate a short route visiting all the nodes in the given set. Whether this route is relevant and interesting for a tourist mainly depends on the chosen points. $\mathcal{T}$ should contain POIs that are useful and worthwhile to visit for a tourist [4]. A PointSelecter plug-in is used to create these $\mathcal{T}$-sets.

A set $\mathcal{T}$ is indirectly judged by the grading function $Rate(P)$ (equation 3.10), after the TSPTW reasoner computed the shortest route visiting all the points in the set. The set of points selected by the PointSelecter should thus adhere to a number of constraints in order to maximize the score of the grading function $Rate(P)$. Two point selection strategies trying to accomplish this, are described in Chapter 5. For each of these selection strategies a separate LarKC plug-in and workflow is created, enabling one to choose which strategy to use.

4.5.3 TSPTW Reasoner

The TSPTW reasoner takes care of calculating a "good route" that visits all the points for the sets $\mathcal{T}$ selected by the Point Selecter. On its input, the TSPTW reasoner receives all the generated sets $\mathcal{T}$. After running the TSPTW solver, the reasoner outputs plans P.

The latitude and longitude of POIs is not the only information considered by the TSPTW solver, time related information is also taken into account. As described in Section 2.3.2, this set of problems is the so-called Traveling Salesman Problem with Time Windows (TSPTW). We include the notion of time windows and service times in our application, for multiple reasons. Tourists will want to have dinner at a certain time, some POIs have limited opening times and at some places you will want to spent more time than at others.

In Section 2.3 another specific form of the Traveling Salesman Problem is introduced; the Time Dependent Traveling Salesman Problem (TDTSP). We would like to incorporate this algorithm in our application. However, in this project we choose to focus on the TSPTW and not on the TDTSP for several reasons. The kind of data needed to consider the time dependency is hard to retrieve. For example, traffic data is not available for each city and even if it is, it might not be publicly available. Including the notion of time dependency is also a complex task and implementing this does not outweigh the gains in the scope of this project.

As discussed in Section 2.3.2 there are currently several TSPTW implementations available. Because the focus of our thesis lies with the Point Selecter and the field of the TSPTW is a well explored field resulting in many efficient implementations, we decide to use an existing implementation. Since a large part of the hypothesis is that we are able to generate plans quickly, we need a fast TSPTW solver. The TSPTW solver, as proposed and designed by Rodrigo Ferreira da Silva and Sebastian Urrutia in [17], is the fastest implementation currently available and even though it does not guarantee an optimal solution, the quality of the solutions are good. Therefore, we decide to incorporate this implementation in the TSPTWReasoner plug-in.

4.5.4 Rank Decider

The Rank Decider receives the plans P on its input. Using the grading function depicted in Equation 3.10, each plan is assigned a grade. These grades are used to rank the results from high to low and enable the frontend to present the user with the best obtained city trip plans. On its output the Rank Decider provides the Cartographer with plans with corresponding grades and ranks.

4.5.5 Cartographer

The Cartographer handles the transformation from the list P to a list with detailed routing information appropriate for depicting the route on a map, or navigating a tourist through a city. Basically, this plug-in transforms the results

into a usable format that can be interpreted by the front-end. The current main objective of the Cartographer is to output information which can be used to depict the generated city plans on a OpenStreetMap based browser instance.

4.6 Frontend

The user needs to be consulted at multiple points in the process about:

- The start and end location of the route, POI_{start} and POI_{end}
- Instance preferences $\mathcal{UI}$
- Class preferences $\mathcal{UC}$ and $\mathcal{NUC}$
- Time constraints of the plan $t^{start_{plan}}$ and $t^{end_{plan}}$
- Plan P preference

The first four interactions will take place before the PointSelecter is initiated. The plan preference stage will start at a later point, when the TSPReasoner generated a couple of plans P, from which the user should choose one.

The LarKC platform only accepts queries as input, which for the average tourist is not an appropriate way of communicating with an application. This is why there should be a frontend, enabling regular people to use the application while not being bothered by queries. A frontend can be realised using different techniques, one more complex to implement than the other.

Although creating a mobile phone app as frontend would be an appropriate choice for this application, this is, unfortunately, not in the scope of our master thesis. Creating a solid app takes a lot of time and is not required to show the correct functioning of the implemented algorithms.

Instead, we start with a simple frontend that can be used while developing. While we develop the strategies and the overall LarKC application, we simply add the parameters mentioned above to the code. The user preferences (sets $\mathcal{UI}$, $\mathcal{UC}$ and $\mathcal{NUC}$) are loaded from a text file, to easily load different scenarios. All the generated city trip plans are send to the Cartographer, which will generate files to show the end result.

The end product will comprise a website showcasing the functionality of the application, which will be handling all the user interaction. A simple web-page like the one used to demonstrate the original Alpha Urban LarKC application[2] is, for example, a more natural way of communicating with the user than the use of console commands and files. POI_{start}, POI_{end}, $\mathcal{UI}$, $\mathcal{UC}$, $t^{start_{plan}}$ and $t^{end_{plan}}$ can be specified in a home screen, which using javascript are translated into a user query for the LarKC platform. After processing this query the platform outputs a list of plans P, which are graphically shown to the user. When a user selects one of the plans, it will be displayed on a map using the OpenStreetMap.org API.

[2] `http://seip.cefriel.it/alpha-Urban-LarKC/`

A different possibility would be to create a Java GUI that communicates directly with the platform. However, due to the fact that Java GUIs are most of the time not the most user friendly ways of communicating with the user and that a website would in principle be able to reach out to a much greater audience, we decide not to do this.

Chapter 5

Strategies

This chapter contains a description of the three strategies selecting POIs in order to create sets $\mathcal{T}$, which are later on used as input for the TSPTW-solver. The Baseline strategy is a naive approach that simply enumerates all the possible sets, considering the user preferences. The results of this strategy are optimal, but enumerating all the possible solutions is time consuming. The Distance Times Rating strategy and Radius strategy, take a heuristic approach, considering the rating of points and the distances between points, in order to find good solutions in a short period of time. Table 5.1 contains the objects used by all strategies.

Objects	Description
$\mathcal{S}$	Set containing all the POIs in the dataset.
$\mathcal{T}$	Set which is filled with POIs in order to eventually serve as input for the TSPTW solver.
Q	Queue containing $\mathcal{T}$-sets ready for the TSPTW solver.
$\mathcal{UI}$	Set containing POIs preferred by the user.
$\mathcal{UC}$	Set containing class pairs representing the class user preferences.
$classPair$	Pair of a class and the amount of times the class is preferred.
$\mathcal{NUC}$	Set containing classes the user does not want to visit.

Table 5.1: Objects used by multiple strategies.

5.1 Baseline Strategy

The Baseline strategy of the point selecter is a basic strategy that enumerates every possible combination of POIs that fits the user preferences, as defined in Section 3.3. The function $Enumerate(\mathcal{S},\mathcal{UI},\mathcal{UC},\mathcal{NUC}) \rightarrow Q$ uses the set containing all POIs $\mathcal{S}$ and the user preferences to fill the queue Q with all enumerations, where $Q = \{\mathcal{T}_1, \ldots, \mathcal{T}_n\}$ and each $\mathcal{T}_i \supseteq \mathcal{UI}$. The number of sets

added to Q depends on the number of possible enumerations of class preferences in combination with randomly added POIs while taking an upper bound into account. We provide an elaborate description of the Baseline strategy in Section 5.1.1. The main steps of the Baseline strategy are designed as recursive functions. Due to the complexity of recursive functions, we provide an example in Section 5.1.2, containing a step by step description of the complete process.

5.1.1 Description Of The Baseline Strategy

The strategy consists of two major steps: enumerating all possible combinations of POIs that satisfy the class preferences as well as the instance preferences and for each of these combination extending them with other points to generate $\mathcal{T}$-sets with a length up to the upper bound[1]. Both steps are designed as recursive functions of which an extensive description can be found below. Table 5.2 illustrates all the objects that, in addition to those in Table 5.1, are used in the Baseline strategy.

Objects	Description
$\mathcal{B}_c$	Set of all POIs that are instances of class C or of a subclass of class C.
$\mathcal{B}_{fill}$	Set of all POIs that do not match the user preferences. Used to fill up plans.
icp	Initial classPair that serves as the first class the $enumerate-Pref$ function is going to look for.
$\mathcal{SC}_c$	Set that contains all subclasses of class c.
$\mathcal{CP}$	Incomplete set that satisfies some user preferences. This is the set the $enumeratePref$ function is currently working on.
$\mathcal{CE}$	The set the $enumerateExtension$ function is currently working on.
$\mathcal{PR}$	Set containing all found preference sets.

Table 5.2: Objects used by the Baseline strategy.

In Algorithm 1, we present a description of the first step in the enumeration process. Before the recursive function can be called, we define the initial classPair icp by retrieving a random element of the set $\mathcal{UC}$. Next we call the enumeratePref function with the icp variable, $\mathcal{UI}$ as the basis of each enumeration (the POIs in $\mathcal{UI}$ have to be included in each set), the set of possible POIs that can be used to satisfy the class preference icp using the $listPossiblePois(icp)$ function and the complete set of class preferences.

Using these variables we enter the first level of the recursive function. The first thing that needs to be done is checking the amount of POIs of the current class the algorithm is working on should still be included in the current set. This is stored in a new variable *initialTimesPreferred*. The rest of the process is

[1]See Section 3.3 for an explanation of what the upper bound of a city trip is

repeated for each possible POI of the current class we are working on, hence the for all statement. A copy of the current set is created to prevent the function from altering the original one, since it is still needed in later steps. The same holds for the current set of class preferences that still need to be fulfilled. The POI in question is then added to the current set $\mathcal{CP}_{new}$ and removed from the list of possibilities. Following on this, timesPreferred is lowered by 1. Based on the state of $\mathcal{UC}_{new}$, the algorithm can now continue in different directions, resulting in three cases.

Algorithm 1 Enumerate all possible $Preference$-sets

```
1: icp ← random element e ∈ UC
2: EnumeratePref(icp,UI,listPossiblePois(icp),UC)
3: procedure EnumeratePref(classPair, CP_old, B_class, UC_old)
4:     initialTimesPreferred ← classPair.getTimesPreferred
5:     for all poi ∈ B_class do
6:         CP_new ← CP_old
7:         UC_new ← UC_old
8:         timesPreferred ← initialTimesPreferred
9:         CP_new.add(poi)
10:        timesPreffered ← timesPreffered − 1
11:        B_class.remove(poi)
12:        if timesPreffered = 0 then
13:            UC_new.remove(classPair)
14:            if not UC_new.isEmpty then
15:                newClass ←random classPair∈ UC_new
16:                B_newClass ← listPossiblePois(newClass)
17:                EnumeratePref(newClass, CP_new, B_newClass, UC_new)
18:            else
19:                PR.add(newSet)
20:            end if
21:        else
22:            classPair.setTimesPreferred(timesPreferred)
23:            EnumeratePref(classPair, CP_new, B_class, UC_new)
24:            classPair.setTimesPreferred(timesPreferred + 1)
25:        end if
26:    end for
27: end procedure
```

The first case occurs when $timesPreferred = 0$ and $\mathcal{UC}_{new}$ is not empty. This means that the current class the algorithm was working on is fulfilled, but there are still other class preference that need to be fulfilled. In this case a new classPair is retrieved from $\mathcal{UC}_{new}$ and the corresponding possible POIs are enlisted by the function $listPossiblePois$ and saved in the set $\mathcal{B}_{newClass}$. Now $enumeratePref$ is called, passing along the new classPair, $\mathcal{CP}_{new}$, $\mathcal{B}_{newClass}$ and $\mathcal{UC}_{new}$.

The second case occurs when $timesPreferred = 0$ and $\mathcal{UC}_{new}$ is empty. This is the actual stop condition of the recursive algorithm and means that the current $\mathcal{CP}_{new}$ satisfies all the user preferences. The resulting $\mathcal{CP}_{new}$ is added to $\mathcal{PR}$ and the algorithm returns one step up in the recursion.

The last case occurs when $timesPreferred > 0$, which means that the current class the algorithm is working on is not yet fulfilled. Now the enumeratePref algorithm is called, passing along the old classPair with a lowered rating, $\mathcal{CP}_{new}$, $\mathcal{B}_{class}$ and $\mathcal{UC}_{new}$ to continue finding POIs of the current class.

In Algorithm 2, we show the pseudocode of the second step in the enumeration process: the fill up. When calling the function, there are two objects that are passed along. These are the possible set of fill up points $\mathcal{B}_{fill}$ created by calling upon the function *ListPoisForExtension* and an empty set, because the starting set needs to be empty. The *listPoisForExtension* function returns all POIs i for which holds:

- $i \notin \mathcal{UI}$
- i instanceOf $C \wedge C \notin \mathcal{UC}$, meaning that i should not be an instance of a preferred class or a subclass of a preferred class.
- i instanceOf $C \wedge C \notin \mathcal{NUC}$, meaning that i should not be of an instance of a non preferred class or a subclass of a non preferred class.

The first statement at line 3 in the algorithm is the stop condition. It states that the algorithm should only create sets that are of length lower or equal to the upper bound and that it should also stop when there are no more POIs that can be used. Next, a copy of $\mathcal{B}_{fill}$ is made in order not to modify the original one, because the original one is still used in the iteration.

Algorithm 2 Enumerate all possible T-sets by adding remaining POIs

```
1:  EnumerateExtension(listPoisForExtension(UI,UC,NUC), {})
2:  procedure EnumerateExtension(B_fill, CF_old)
3:      if B_fill.isEmpty ∨ CF_old.size = upperbound then      ▷ Stop condition
4:          return
5:      end if
6:      B_fillNew ← B_fill
7:      for all poi ∈ B_fill do
8:          CF_new ← CF_old
9:          CF_new.add(poi)
10:         B_fillNew.remove(poi)
11:         if CF_new.length ≥ lowerbound then
12:             for all prefSet ∈ PR do
13:                 Q.offer(combine(prefSet, CF_new)
14:             end for
15:         end if
16:         EnumerateExtension(B_fillNew, CF_new)
17:     end for
18: end procedure
```

The rest of the process is repeated for each POI in $\mathcal{B}_{fill}$. A copy of the current set is made in order to preserve the old one, since it might still be needed on a higher level in the recursion. The current set is then expanded with the POI that is currently selected. Next, the POI in question is removed from $\mathcal{B}_{newFill}$ to mark that it can no longer be used. If the length of the current set exceeds the lower bound, it is suitable to include in a $\mathcal{T}$-set. It is combined with each of the preference sets to generate $\mathcal{T}$-sets that are added to Q. Next, the function goes deeper into the recursion, passing along the new set of possible POIs $\mathcal{B}_{fillNew}$ and $\mathcal{CF}_{new}$.

5.1.2 Example

Due to the complexity of the recursive property of the Baseline strategy, we present an example in order to clarify the process.

In our example situation we define the sets $\mathcal{S} = \{m1, m2, m3, c1, c2, r1, r2\}$, $\mathcal{UC} = \{(Restaurant, 1), (Church, 2)\}$ and $\mathcal{UI} = \{m1\}$ where $m1$,$m2$ and $m3$ are instances of class $Museum$, $c1$ and $c2$ instances of class $Church$ and $r1$ and $r2$ instances of class $Restaurant$. An overview can be found in Table 5.3.

Class	**POIs**
Museum	$m1,\ m2,\ m3$
Church	$c1,\ c2$
Restaurant	$r1,\ r2$

Table 5.3: Example classes Baseline

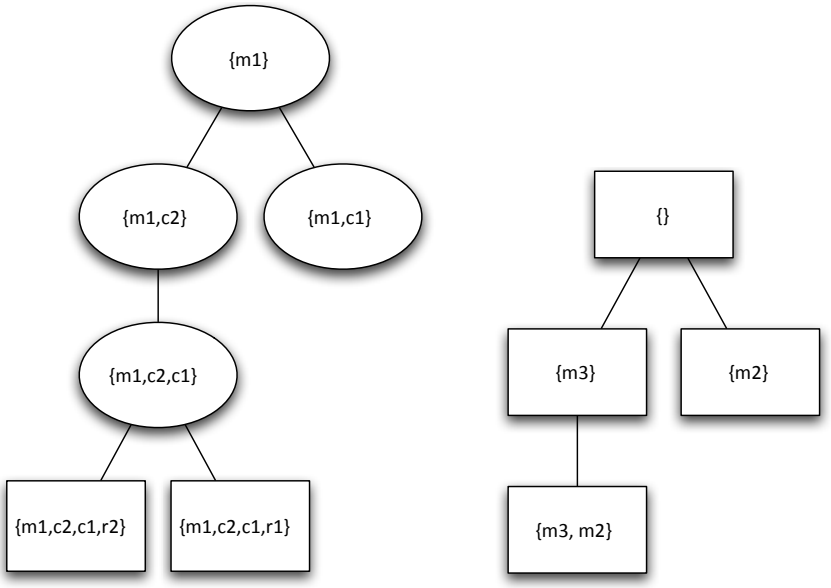
In Figure 5.1 the construction of preference sets and extension sets of the example situation can be found. In both cases, the node at the top of the tree is the first level of the recursion, which means that it represents the base set of the process. For the $enumeratePref$ function this is $\mathcal{UI}$ and for the enumerateExtension function this is an empty set.

Each level of the tree represents a recursion level meaning that for each level the algorithm traverses in the recursion, we also have to traverse a level deeper in the tree. The nodes which are formed like an ellipse in the $enumeratePref$ tree, are nodes that do not satisfy each user preference and thus are not added to $\mathcal{PR}$. The leaf nodes in the form of a rectangle are the sets that satisfy all the user preferences and are added to $\mathcal{PR}$.

In order to initiate the $enumeratePref$ process, we need to retrieve an initial class from the set of preferred classes. Since sets are unordered objects, this will be a random element from the set $\mathcal{UC}$. Next, we initiate the function $enumeratePref$ and pass along the intitialClass, a set of POIs that belong to that class (using the function $listPossiblePois$) and $\mathcal{UI}$.

Let us say that we select the class *church* at random. The set of possible POIs ($\mathcal{B}_{church}$) will now contain $c1$ and $c2$. The algorithm selects $c2$ at random to create the set $\{m1, c2\}$. Since the class preference of *church* is not yet fulfilled, the algorithm traverses one level deeper and, at this level, forms the

set $\{m1, c2, c1\}$, which satisfies the class preference of *church*. At this moment, the class preference of one restaurant still needs to be fulfilled, so the algorithm traverses another level deeper. Here it selects $r2$ as the restaurant and creates the set $\{m1, c2, c1, r2\}$ that satisfies all user preferences and, thus, is added to $\mathcal{PR}$.



(a) EnumeratePref (b) EnumerateExtension

Figure 5.1: Trees visualizing Baseline example.

On this level there is another possibility of adding restaurant $r1$. The algorithm constructs the set $\{m1, c2, c1, r1\}$ and also adds this to $\mathcal{PR}$. Now all the possibilities are explored and the algorithm traverses one level up. Here the possibilities were also explored, so the algorithm traverses another level up.

At this level there was still a possibility of adding $c1$ instead of $c2$, so the set $\{m1, c1\}$ is created. With this set, the user preference of *church* is not yet fulfilled so the algorithm traverses one level deeper. Since there are no possibilities left at this level, the algorithm does not go into the $forAll$ loop and thus traverses up one level and leaves the recursion.

The process for enumerateExtension progresses in a similar fashion, but there is only one set of possibilities ($\mathcal{B}_{fill}$) and each node is a valid extension set, even the empty set. Every generated extension set is combined with each of the found preference sets to generate valid $\mathcal{T}$-sets. This results in the eight $\mathcal{T}$-sets that can be found in table 5.4.

Extension Set	Preference Set	Resulting $\mathcal{T}$-set
{}	$\{m1, c2, c1, r2\}$	$\{m1, c2, c1, r2\}$
	$\{m1, c2, c1, r1\}$	$\{m1, c2, c1, r1\}$
$\{m3\}$	$\{m1, c2, c1, r2\}$	$\{m1, c2, c1, r2, m3\}$
	$\{m1, c2, c1, r1\}$	$\{m1, c2, c1, r1, m3\}$
$\{m3, m2\}$	$\{m1, c2, c1, r2\}$	$\{m1, c2, c1, r2, m3, m2\}$
	$\{m1, c2, c1, r1\}$	$\{m1, c2, c1, r1, m3, m2\}$
$\{m2\}$	$\{m1, c2, c1, r2\}$	$\{m1, c2, c1, r2, m2\}$
	$\{m1, c2, c1, r1\}$	$\{m1, c2, c1, r1, m2\}$

Table 5.4: Resulting $\mathcal{T}$-sets of the Baseline example.

5.2 Distance Times Rating Strategy

The rating function introduced in Section 3.3 considers multiple aspects of a city trip plan to determine whether it is good or not. The plan must include POIs matching the user preferences and it must comply with the overall time window $t^{end_{plan}} - t^{start_{plan}}$, otherwise it is invalid. Major aspects influencing the score of the rating function are the amount of time used for traveling between points, the rating of the chosen POIs and the time spent at these points.

The Distance Times Rating strategy (DTR), described in pseudo code in Algorithm 4, considers these aspects in order to find promising $\mathcal{T}$-sets. Objects introduced specifically for DTR can be found in Table 5.5, whereas additional objects, which are shared with the other two strategies, can be found in Table 5.1.

Objects	Description
$\mathcal{S}_{filtered}$	$\mathcal{S}$ without all the points belonging to classes in $\mathcal{NUC}$ and without the POIs from $\mathcal{UI}$.
classLists	An object containing lists with POIs belonging to a class C.
includedClasses	List containing the classes present in *classLists*.
banQueue	The *banQueue* contains POIs who are going to be banned.
banAmount	The number of POIs that should be banned.
$\mathcal{BP}$	Set containing the POIs which are currently banned.
$\mathcal{CC}$	Set containing the classes which are considered either for adding POIs corresponding to the user preferences to the $\mathcal{T}$-set, or for adding fill POIs to the $\mathcal{T}$-set.
cardinality	The estimated optimal number of POIs to be included in a $\mathcal{T}$-set.
$foundSets$	A set containing all the $\mathcal{T}$-sets already found.
desired	The number of desired $\mathcal{T}$-sets.

Table 5.5: Objects used by the DTR strategy

In order to select appropriate points, DTR adds the POIs of set $\mathcal{S}$ to lists which correspond to their respective class membership. These lists are ordered according to the time it takes to reach the candidate POI from one of the points the tourist is already about to visit combined with the rating of the candidate POIs, thereby attempting to minimize travel distance and maximize the POI ratings. Table 5.6 contains an example of such a classLists object, filled with POIs from the baseline example of Section 5.1.2.

ClassList	POIs and Respective Scores
Museum	**(m2,0.87)** $(m3, 0.63)$ $(m1, 0.21)$
Church	$(c1, 0.79)$ $(c2, 0.33)$
Restaurant	$(r2, 0.70)$ $(r1, 0.66)$

Table 5.6: Example ClassLists object with most promising POI in bold.

The most promising candidate POI is added to the $\mathcal{T}$-set and the distances are updated using this new POI. This process is repeated until the $\mathcal{T}$-set is of the size *optimal cardinality* and can be submitted to the TSPTW process. The optimal cardinality is updated by Algorithm 10 each time a new point is added, in order to determine whether adding another point will violate the time constraints, thereby creating plans with as many points as possible.

While the main algorithm of the strategy, given in Algorithm 4, makes the generation of multiple $\mathcal{T}$-sets possible, the most important algorithm is the one actually creating the $\mathcal{T}$-sets, Algorithm 3. This algorithm keeps adding POIs to the current $\mathcal{T}$-set until the optimal cardinality is reached. Every time a new POI is added, the *classList* objects scores are updated using Algorithm 6. Which POI is added depends on whether there are still user class preferences left. When there are, Algorithm 8 is used to find an appropriate POI, otherwise Algorithm 9 is used to find a fill POI.

Algorithm 3 The creation of $\mathcal{T}$-set using the DTR strategy

```
1:  procedure createTSet(S, UC, UI, POI_start, t^start_plan, t^end_plan)
2:      cardinality ← ∞                                   ▷ Will be refined
3:      T ← new Set(UI)
4:      while T.size() < cardinality do
5:          classList.scoreLists(T)
6:          if noClassPreferences() then
7:              chosenPoi ← findFillPoi(T)
8:          else
9:              chosenPoi ← findUserPoi(T)
10:         end if
11:         if not banQueue.contains(chosenPoi) then
12:             banQueue.add(chosenPoi)
13:         end if
```

Algorithm 3 The creation of $\mathcal{T}$-set using the DTR strategy (continued)

```
14:         T.add(chosenPoi)
15:         cardinality ← determineCardinality(T, t^start_plan, t^end_plan)
16:     end while
17:     return T
18: end procedure
```

When a POI is added to the $\mathcal{T}$-set, it is also added to the *banQueue*. The *banQueue* is used to create a sequence of points to exclude (i.e., ban) from the lists that are used by the processes searching for appropriate POIs to add to the $\mathcal{T}$-sets. When the right points are excluded, additional $\mathcal{T}$-sets nearly as good as the first $\mathcal{T}$-set can be created.

The main method of the DTR strategy, as described in Algorithm 4, starts by generating the *classLists* object, after which it uses a while-loop to continue creating $\mathcal{T}$-sets until the number of found sets matches the amount of sets desired by the user. Due to the constant change of the scores in the *classLists* object, it is not guaranteed that the $\mathcal{T}$-set is not already found. The DTR algorithm, therefore, actively manages the amount of POIs that are banned, resets the *banQueues* when needed and permanently bans POIs when it is impossible to generate new sets using the current considered POIs. This results in three cases when there is a new $\mathcal{T}$ created.

The first case occurs when $\mathcal{T}$ **has never been found before**. At this moment $\mathcal{T}$ is added to the $foundSets$ and Q, which serves as input for the TSP solver. The method *counsel*() is invoked, to grant mercy to POIs banned in the previous iteration and to ban a *banAmount* number of POIs for the next iteration, in order to ensure the creation of new sets. If the current set $\mathcal{T}$ is the best set deducted using the currently known information, the current *banQueue* is saved as the *bestBanQueue*, for later use.

The second case occurs when the **set is already found and too many POIs are banned**. When the number of POIs which have to be banned exceeds the optimal cardinality, the counsel method will attempt to ban more points than available in the *banQueue*. To counter this problem, one POI of the *bestBanQueue* is permanently banned. This will result in new $\mathcal{T}$-sets, all without the banned POI. The iterations start over with new queues and a *banAmount* of 1.

The last case occurs when the **set** $\mathcal{T}$ **is already found and** *banAmount* **does not exceed the** *cardinality*. At this point, it is unlikely to create good new sets, because all the best POIs have already been banned. In order to create new sets, the number of points which should be banned at once is increased by one. A new *banQueue* containing the POIs from the *bestBanQueue* is created.

The *classLists* object used for selecting the most promising POIs, is filled using a filtered set $\mathcal{S}$. The user preferences $\mathcal{UI}$ are removed from $\mathcal{S}$, because they are always added to the initial $\mathcal{T}$ set. These POIs will be visited in any case, otherwise the plan would be invalid. $\mathcal{NUC}$ contains the classes of POIs a user does not want to visit. POIs belonging to these classes and their subclasses

are also removed from the set $\mathcal{S}$.

Algorithm 4 Distance Times Rating Strategy

```
1: procedure DTRStrategy(S, UC, NUC, UI, POI_start, t^start_plan, t^end_plan)
2:     S_filtered ← filterPois(S, UC, NUC, UI, POI_start)
3:     classLists.fillLists(S_filtered)
4:     newBestSet ← true
5:     while foundSets.size() < desired do
6:         T ← createTSet()
7:         if not foundSets.contains(T) then                    ▷ Case 1
8:             Q.add(T)
9:             foundSets.add(T)
10:            BP ← counsel(banQueue, BP, banAmount)
11:            if newBestSet = true then
12:                bestBanQueue.addAll(banQueue)
13:                newBestSet ← false
14:            end if
15:        else if foundSets.contains(T) ∧ banAmount ≥ cardinality then
16:            grantMercy(BP)                                   ▷ Case 2
17:            ban(bestBanQueue, 1)
18:            bestBanQueue ← new Queue
19:            BP ← new Set
20:            banQueue ← new Queue
21:            newBestSet ← true
22:            banAmount ← 1
23:        else if foundSets.contains(T) then                   ▷ Case 3
24:            if not bestBanQueue.isEmpty() then
25:                banQueue ← new Queue
26:                banQueue.addAll(bestBanQueue())
27:                banAmount ← banAmount + 1
28:            end if
29:            BP ← counsel(banQueue, BP, banAmount)
30:        end if
31:    end while
32: end procedure
```

The POIs of the set $\mathcal{S}_{filtered}$ are added to lists containing POIs of the same class. This process is described by Algorithm 5. Whenever there is not a list containing the current class C of a POI present in *classLists*, a new list for that particular C is added. This process is repeated until all $POI \in \mathcal{S}_{filtered}$ are added to the *classLists* object.

Algorithm 5 Fill the classLists object with POIs

```
1: procedure FILLLISTS(S_filtered)
2:     for all poi ∈ S_filtered do
3:         class ← poi.getClass()
4:         if classLists.containsClassList(class) then
5:             classLists.addToList(class, poi)
6:         else
7:             classLists.addClassList(class)
8:             classLists.addToList(class, poi)
9:         end if
10:    end for
11: end procedure
```

In order to be able to easily pick the best POI available at the moment, we decide to rank the POIs using a heuristic. The next process sorts the lists in the *classLists* object using a score obtained by Formula 5.1. The pseudo code of this process is given in Algorithm 6. The rating of each POI is already available in the dataset, represented as $r \in [0, 1]$.

The distance to the nearest point has to be calculated using Algorithm 7, which utilizes a for loop to determine the nearest POI in $\mathcal{T}$. In order to be able to scale the distance score from 0 to 1, a measure of maximum distance has to be known. This is the highest found distance between two POIs in the dataset. After the $heuristicScore(POI)$ is calculated for each POI in each list in the *classLists* object, all the lists are sorted using the scores.

$$heuristicScore(POI_i) = r_i \cdot (1 - \frac{minDistance(\mathcal{T})}{maximumDistance}) \tag{5.1}$$

Algorithm 6 Assign scores to the POIs present in the classLists object

```
1: procedure SCORELISTS(T, POI_start)
2:     for all class ∈ includedClasses do
3:         list ← classLists.getList(class)
4:         for all poi ∈ list do                    ▷ Rate each poi in list
5:             nearest ← determineNearestPoint(poi, T, POI_start)
6:             rating ← poi.getRating()
7:             score ← (rating · (1 − (nearest / maxDistance)))
8:             POI.setScore(score)
9:         end for
10:        list.sort()                    ▷ Sort the list using obtained scores
11:    end for
12: end procedure
```

Algorithm 7 Determine nearest point in $\mathcal{T}$

```
1: procedure DETERMINENEARESTPOINT(poi, T, POI_start)
2:     nearest ← ∞
3:     for all tPoi ∈ T ∪ POI_start do
4:         distance ← calculateDistance(tPoi, poi)
5:         if distance < nearest then
6:             nearest ← distance
7:         end if
8:     end for
9:     return nearest
10: end procedure
```

Algorithms 8 and 9 illustrate the process of finding appropriate points for inclusion in the $\mathcal{T}$-set. Algorithm 8 is concerned with finding the POIs necessary for fulfilling the user class preferences $\mathcal{UC}$. First a set $\mathcal{CC}$ is created, containing all the classes which are preferred at least once. For each of these classes, the highest ranking POI in the object *classLists* is retrieved. Only the POIs which are not banned or already included in $\mathcal{T}$ are considered. The Boolean *endListReached* is used to make sure that no more points are considered than there are available in the list. The POI with the highest overall ranking is returned.

Algorithm 8 Find the most promising POI considering the $\mathcal{UC}$

```
1: procedure FINDUSERPOI(T, UC)
2:     CC ← retrievePreferredClasses(UC)
3:     for all class ∈ CC do
4:         list ← classLists.getList(class)
5:         position ← 0
6:         poi ← list.get(position)
7:         endListReached ← false
8:         while poi.getStatus() = banned ∨ poi ∈ T ∧
9:                 not endListReached do
10:            position ← position + 1
11:            if position > list.size() − 1 then
12:                endListReached ← true
13:            else
14:                poi ← list.get(position)
15:            end if
16:        end while
17:        score ← poi.getScore()
18:        if currentScore > bestScore then
19:            bestPoi ← poi
20:            bestScore ← score
21:        end if
22:        UC.lowerAmountPreferred(bestPoi.getClass)
23:    end for
24:    return bestPoi
25: end procedure
```

Algorithm 9 is in essence the same as Algorithm 8. There are some minimal differences though: the amount of times the class is preferred is not lowered and the possible fill classes are used in the candidate class set $\mathcal{CC}$. The previous algorithm deducted $\mathcal{CC}$ from the preferred classes in $\mathcal{UC}$.

Algorithm 9 Find the most promising POI for filling the $\mathcal{T}$ set

```
1:  procedure findFillPoi(T, UC)
2:      CC ← retrievePossibleFillClasses(UC)
3:      for all class ∈ CC do
4:          list ← classLists.getList(class)
5:          position ← 0
6:          poi ← list.get(position)
7:          endListReached ← false
8:          while poi.getStatus() = banned ∨ poi ∈ T ∧
9:                  not endListReached do
10:             position ← position + 1
11:             if position > list.size() − 1 then
12:                 endListReached ← true
13:             else
14:                 poi ← list.get(position)
15:             end if
16:         end while
17:         score ← poi.getScore()
18:         if currentScore > bestScore ∧ not endListReached then
19:             bestPoi ← poi
20:             bestScore ← score
21:         end if
22:     end for
23:     return bestPoi
24: end procedure
```

New points are added to the $\mathcal{T}$-sets until the estimated optimal cardinality of the set is reached. Finding a number for this optimal cardinality is the responsibility of Algorithm 10. This algorithm calculates an estimation of the time required to visit the POIs in the current $\mathcal{T}$-set, by determining the sum of the service times in addition to an estimation of the travel time, calculated using the number of included points. This is turned into a prospect by the addition of the maximum duration d and an estimation of the travel time, thereby leaving room for a good choice considering the rating function, namely a POI with a high duration d.

When this prospect does not exceed the overall time window, the optimal cardinality is increased. The returned cardinality is somewhat pessimistic regarding the total time, due to the use of $maxService$. Otherwise it would be likely for the strategy to generate a lot of sets which are unsolvable, due to the imposed time restrictions. When the overall strategy is compared to a baseline,

the cardinality should be equal or less than the upperbound of that baseline, otherwise it is not possible to compare the results.

Algorithm 10 Determine the optimal cardinality for the $\mathcal{T}$ sets

```
1:  procedure CARDINALITY(T, t^start_plan, t^end_plan)
2:      availableTime ← t^end_plan − t^start_plan
3:      cardinality ← T.size()
4:      totalServiceTime ← 0
5:      for all poi ∈ T do
6:          totalServiceTime ← totalServiceTime + poi.getServiceTime
7:      end for
8:      estimatedTime ← totalServiceTime + (30 · T.size)
9:      while estimatedTime < availableTime do
10:         cardinality ← cardinality + 1
11:         estimatedTime ← estimatedTime + maxService + 30
12:     end while
13:     if cardinality > upperBound then
14:         return upperBound
15:     end if
16:     return cardinality
17: end procedure
```

Based on an identical *classLists* object, Algorithm 3 will produce the same $\mathcal{T}$-set over and over again. A sound strategy should however produce a *desired* amount of unique sets. The method *counsel*, as described in Algorithm 11, is used to accomplish this. It grants *mercy* to the POIs banned in the previous iteration and uses the *banQueue* to find the next POIs to ban. The banned POIs are returned, so they can be granted mercy in the next iteration.

The *grantMercy* procedure, as described in Algorithm 11, uses the *previouslyBanned* set to grant mercy to every POI which was banned in the last iteration. It set the status of the POI to free and increases the number of available POIs of the class.

Because every time at least one POI has to be banned to be able to create a new set, the *ban* procedure starts by banning the first POI of the *banQueue*. It could be that just banning one POI would not create a new set $\mathcal{T}$ anymore, so a *peekList* is created. This list is used to peek a few spots ahead in the *banQueue* and if the size of the *peekList* allows it, ban a *banAmount* points from the dataset.

The *banPoi* procedure is used for the actually banning of a POI. It considers the times the class of the POI is needed in order to be able to fulfill the user preferences. If this amount is the same or exceeding the available amount, the number of POIs to be banned is temporarily increased in order to skip this POI and ban another one. If it is possible to ban the POI, the status of the POI is set to banned, the number of POIs from this class available is decreased by one and the POI is added to the set of banned POIs. This set of bannedPois is

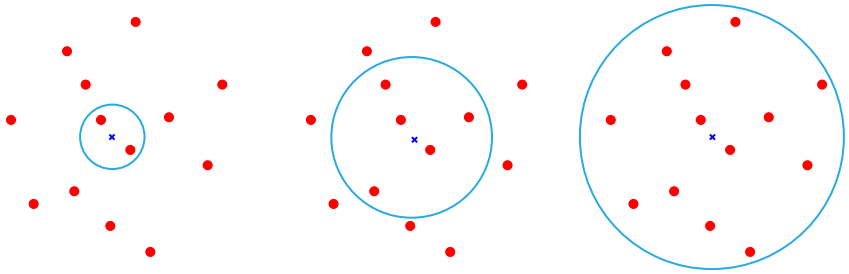
returned.

Algorithm 11 Grant mercy to the previously banned and ban number of POIs

```
1: procedure COUNSEL(banQueue, previouslyBannedPois, banAmount)
2:     grantMercy(previouslyBannedPois)
3:     BP ← ban(banQueue, banAmount)
4:     return BP
5: end procedure
6:
7: procedure GRANTMERCY(previouslyBannedPois)
8:     if not previouslyBanned = null then
9:         for all poi ∈ previouslyBannedPois do
10:            poi.setStatus(free)
11:            poi.getCategory().increaseNumberAvailable()
12:        end for
13:    end if
14: end procedure
15:
16: procedure BAN(banQueue, amountToBan)
17:    banAmount ← amountToBan
18:    BP ← banPoi(banQueue.poll(), BP)
19:    peekList.addAll(banQueue)
20:    for i ← 0, banAmount − 1 do
21:        if i < peekList.size() then
22:            BP ← banPoi(peekList.get(i), BP)
23:        end if
24:    end for
25:    return BP
26: end procedure
27:
28: procedure BANPOI(banPoi, BP)
29:    class ← banPoi.getClass()
30:    timesDesired ← class.getTimesPreffered()
31:    available ← getClassList(category).getNumberAvailable()
32:    if timesDesired >= available then
33:        banAmount ← banAmount + 1
34:        return BP
35:    else
36:        banPoi.setStatus(banned)
37:        getClassList(category).decreaseNumberAvailable()
38:        BP.add(banPoi)
39:    end if
40:    return BP
41: end procedure
```

5.3 Radius Strategy

The DTR strategy is based on ratings of POIs as well as on the distances between the POIs. Here, we introduce an entirely different approach that also considers both aspects, but is formalized as a trade-off between them. The basic idea of the Radius strategy is to select highly rated POIs within an increasing radius, starting from the *center* of the urban environment, to form suitable $\mathcal{T}$-sets. An illustration of the idea can be seen in Figure 5.2. The small cross represents the center of the urban environment, the blue circle represents the radius and the POIs are represented by a red dot.



(a) step 1 (b) step 2 (c) step 3

Figure 5.2: Process of Radius strategy

The Radius Strategy forms set containing POIs that lie within the radius. If no suitable sets can be found or if the user asks for more plans, the radius is increased. The main advantage of the Radius strategy, in contrary to the DTR strategy, is the fact that it is able to function without the assumption of needing a finite dataset. As long as the center is known, one could impose a geographical query that retrieves all POIs that lie within the radius. Since we use many different objects within the Radius strategy, we provide a clear overview in table 5.7.

Objects	**Description**
radius	Radius in meters: the distance from the center in which the strategy searches for suitable points.
radiusIncrease	Increase of the radius in meters after each enumeration step. Also starting value of the radius.
center	The center of the radius is determined as the weighted average position of the POIs.
minRating	The minimal rating a POI should have in order to be included in $\mathcal{T}$-sets.
desired	The number of desired $\mathcal{T}$-sets.
produced	Amount of $\mathcal{T}$-sets that are found.

Continued on next page

Objects	Description
R	Sorted list of all POIs that lie within the current radius and have a rating above $minRating$.
R_{fill}	Sorted list of all POIs that lie within the current radius and have a rating above $minRating$ that do not match the user preferences, used to fill up plans.
R_c	Sorted list of all POIs that lie within the current radius and have a rating above $minRating$ that contains POIs that are of a certain class c $\in \mathcal{UC}$.
icp	Initial classPair that serves as the first class the $enumeratePref$ function is going to look for.
$\mathcal{SC}_c$	Set that contains all subclasses of class c.
$\mathcal{CP}$	Incomplete set that satisfies some user preferences. This is the set the $enumeratePref$ function is currently working on.
PR	Sorted list containing all currently found preference sets within the current radius.
FI	Sorted list containing all currently found fill sets within the current radius.
$allSets$	The set containing all sets that already have been added to Q. Used in following iterations to check whether found $\mathcal{T}$-sets are new.

Table 5.7: Objects for Radius strategy.

A high level description of the Radius Strategy can be found in Algorithm 12. The strategy starts by determining the centre of the radius, executed by a separate function that can be found in Algorithm 13. At the start of the process, we set the radius to 10, meaning that the strategy looks for suitable points within a radius of 10 meters from the center. Within the while-loop we call upon the function that does the actual enumeration of all the 'good sets' that can be found inside the radius. Finally, we expand the radius after each enumeration and rerun the entire enumeration process in order to calculate additional results. This is repeated until the desired number of $\mathcal{T}$-sets is found.

Algorithm 12 Radius Strategy

```
1: procedure RadiusStrategy(S, UC, UI, NUC)
2:     center ← determineCenter(S)
3:     radiusIncrease ← 10
4:     radius ← radiusIncrease
5:     minRating ← 0.5
6:     while produced < desired do
7:         enumerateInRadius(radius, minRating, S, UI, UC, NUC)
8:         radius ← (radius + radiusIncrease)
9:     end while
10: end procedure
```

The center of the radius is calculated in Algorithm 13. We define the center of the radius as the "average most interesting point", which can be calculated using Equation 5.2 and Equation 5.3. The basic idea behind the "average most interesting point" is taking the average position of the POIs within an urban environment as the center. However, due to the fact that certain POIs are more interesting than others (represented by the rating), the higher rated POIs should have more influence on the position of the center. Therefore, the geographical coordinates of each POI in $\mathcal{S}$ are multiplied by the rating of the POI to scale its influence on the position according to its rating. The sum of these numbers is then divided by the sum of all ratings to calculate the coordinates of the "average most interesting point".

$$latCenter = \frac{\sum_{i=1}^{n} lat_i * r_i}{\sum_{i=1}^{n} r_i} \tag{5.2}$$

$$lonCenter = \frac{\sum_{i=1}^{n} lon_i * r_i}{\sum_{i=1}^{n} r_i} \tag{5.3}$$

Algorithm 13 Determine the Center of the Radius

```
1:  procedure DetermineCenter(S)
2:      sumLat ← 0
3:      sumLon ← 0
4:      sumRating ← 0
5:      for all poi ∈ S do
6:          sumLat ← sumLat + (poi.getLat * poi.getRating())
7:          sumLon ← sumLon + (poi.getLon * poi.getRating())
8:          sumRating ← sumRating + poi.getRating()
9:      end for
10:     center.lat ← (sumLat/sumRating)
11:     center.lon ← (sumLon/sumRating)
12:     return center
13: end procedure
```

In order to be able to determine suitable $\mathcal{T}$-sets, we first need to determine which POIs have an acceptable rating and lie within the radius. Therefore, we call upon the function *getPOIsInRadius*, which returns the sorted list R containing all the POIs within the radius that have a rating above the minimum rating threshold. This function is described in more detail in Algorithm 15. Setting the minimum rating threshold is exactly the trade-off mentioned earlier. On the one hand, if we set the value of *minRating* too high, the rating is considered to be more important than distance, since less points will be found and thus increasing the radius will lead to a higher distance between the different points. On the other hand, if we set the value of *minRating* too low, more points will be found in a smaller radius making the rating subordinate to the distance.

Now that we have the sorted list R, we execute $enumeratePref$ to retrieve sorted list PR containing all possible sets within the radius that satisfy all the user preferences. Next, we need to determine which POIs can still be used to fill up the preference sets to generate $\mathcal{T}$-sets that fill up the users entire day by calling upon the $getFillPOIs$ function which is, on its turn, explained in Algorithm 16. Using this sorted list R_{fill}, we can call the $enumerateFill$ function to enumerate all extension sets, starting with the highest rated, and combining them with each of the preference sets. The enumerateFill function is described in more detail in Algorithm 18.

Algorithm 14 Enumerate the Possibilities Within the Radius

```
1: procedure EnumerateInRadius(radius, minRating, 𝒮, 𝒰ℐ, 𝒰𝒞, 𝒩𝒰𝒞)
2:     R ← getPOIsInRadius(𝒮, 𝒰ℐ, 𝒩𝒰𝒞, radius, center, minRating)
3:     enumeratePref(radius, icp, 𝒰ℐ, R, R_c, 𝒰𝒞)
4:     R_fill ← getFillPOIs(R, 𝒰𝒞, 𝒰ℐ, radius)
5:     enumerateFill(R_fill, PR)
6: end procedure
```

Retrieving all suitable points within the radius is done for each POI by calculating the distance between the POI and the center of the radius. This process can be seen in more detail in Algorithm 15. If this distance is lower than the radius, the POI in question is not of a non preferred class or a subclass of a non preferred class and the rating of the POI is greater or equal to the minimum rating threshold, the POI in question is added to list R. The calculation of the distance in meters is executed by the *calculateDistance* function that uses Pythagoras' theorem in combination with the geographical coordinates.

Algorithm 15 Return All POIs Within the Radius

```
1: procedure GetPOIsInRadius(𝒮, 𝒰ℐ, 𝒩𝒰𝒞, radius, center, minRating)
2:     for all poi ∈ 𝒮 do
3:         distance ← calculateDistance(center, poi)
4:         nonPref ← false
5:         for all subclass ∈ poi.getClass().getSubClasses() do
6:             if 𝒩𝒰𝒞.contains(subclass) then
7:                 nonPref ← true
8:             end if
9:         end for
10:        if distance ≤ radius ∧ not 𝒰ℐ.contains(poi)
11:          ∧ not 𝒩𝒰𝒞.contains(poi.getClass())
12:          ∧ not nonPref ∧ poi.rating > minRating then
13:            R.add(poi)
14:        end if
15:    end for
16:    return R
17: end procedure
```

The function enumerateExtension needs a list of POIs that can be used to fill up the sets from PR and create $\mathcal{T}$-sets. In order to provide the function with this list, we define Algorithm 16. POIs that can be used to fill up the $\mathcal{T}$-sets are POIs that do not belong to a class in $\mathcal{UC}$, not to a subclass of a class in $\mathcal{UC}$ (subsume match), not to a class in $\mathcal{NUC}$, not to a subclass of a class in $\mathcal{NUC}$ and are not in the set $\mathcal{UI}$. Since the latter three were already filtered by using Algorithm 15, we only have to filter using $\mathcal{UC}$.

Algorithm 16 Return All POIs Within the Radius Not Matching Preferences

```
1:  procedure GetFillPOIs(R, UC, radius)
2:      for all poi ∈ R do
3:          exact ← false
4:          subsume ← false
5:          for all classPair ∈ UC do                    ▷ Check exactmatches
6:              class ← classPair.getClass()
7:              if class = poi.getCategory then
8:                  exact ← true
9:              end if
10:             SC_class ← class.getSubClasses()
11:             for all subclass ∈ SC_class do           ▷ Check subsume matches
12:                 if subclass = poi.getCategory then
13:                     subsume ← true
14:                 end if
15:             end for
16:         end for
17:         if not exact ∧ not subsume then
18:             R_fill.add(poi)
19:         end if
20:     end for
21:     R_fill.sort()                                    ▷ Sort descending
22:     return R_fill
23: end procedure
```

In Algorithm 17 one can find the $enumeratePref$ function, which enumerates all the Preference sets that can be found given R. The function is recursive and is stated in a similar fashion as the $enumeratePref$ function of the baseline strategy in Section 5.1.1. However, due to the fact that we are dealing with sorted lists as input, there are some minor differences considering the iteration through the lists R_c. Instead of randomly iterating through them, we use a for loop that runs from 0 until the size of R_c.

Algorithm 17 Enumerates All Preference Sets In Radius Starting With Best Rated

```
1: procedure ENUMERATEPREF(radius, classPair, CP_old, R, R_c, UC_old)
2:     initialTimesPreferred ← classPair.getTimesPreferred()
3:     for i ← 0 until R_c.size() do
4:         UC_new ← UC_old
5:         timesPreferred ← initialTimesPreferred
6:         CP_new ← CP_old
7:         CP_new.add(R_c.get(i))
8:         timesPreferred ← timesPreferred − 1
9:         R_c.remove(i)
10:        if timesPreferred = 0 then
11:            UC_new.remove(classPair)
12:            if not UC_new.isEmpty() then
13:                classPair ← UC_new.next()
14:                R_c ← getPrefPOIs(radius, R, classPair, CP_new)
15:                enumeratePref(radius, classPair, CP_new, R, R_c, setClass_new)
16:            else
17:                PR.add(CP_new)
18:            end if
19:        else
20:            classPair.setTimesPreferred(timesPreferred)
21:            enumeratePref(radius, classPair, CP_new, R, R_c, UC_new)
22:            classPair.setTimesPreferred(timesPreferred + 1)
23:        end if
24:    end for
25: end procedure
```

Algorithm 18 describes the extension of the found preference sets with highly rated points. Using the list of fill points R_{fill} and PR, $\mathcal{T}$-sets are created in an orderly fashion. The basic construction of the function is a triple for-loop that enumerates each possible extension set and combines this with each preference set to calculate all possible $\mathcal{T}$-sets. This is done by combining each $POI_i \in R_{fill}$ with each fill set in FI, containing all the previous found extension sets, and add the result to FI. This way, each new $POI_i \in R_{fill}$ is combined with all previous found extension set and thus generates all possible fill sets. The found fill sets are on their turn combined with the ordered list of preference sets PR. Since sets containing more points are more likely to result in a plan with a higher grade, only the generated $\mathcal{T}$-sets of length upper bound are added to Q.

Algorithm 18 Enumerates All Fill Sets In Radius Starting With Best Rated

```
1: procedure enumerateFill(R_fill, PR)
2:     for i ← 0 until R_fill.size do
3:         for j ← 0 until FI.size() do
4:             if FI.get(j).size() < upperbound then
5:                 fillSet ← combine(FI.get(j), R_fill.get(i))
6:                 FI.add(fillSet)
7:                 for k ← 0 until PR do
8:                     T_new ← combine(fillSet, PR.get(k)
9:                     if not allSets.contains(T_new) then
10:                        allSets.add(T_new)
11:                        if T_new = upperbound then
12:                            Q.offer(T_new)
13:                        end if
14:                    end if
15:                end for
16:            end if
17:        end for
18:    end for
19: end procedure
```

Chapter 6

Data

The data used by the city trip planner is an important factor considering the quality of its results. A discussion on which data sources to use can be found in Section 6.1. In Section 6.2, we describe the process of pre-processing this data in order to make it usable for our application. We continue with the data analysis in Section 6.3 where we take a look at the contents of the different datasets. Finally, we conclude with Section 6.4, in which we discuss the quality of the different datasets.

6.1 Data Sources

Given our problem setting, there are many different kinds of data that can be considered. Tourists might want to visit all sorts of interesting sights in a city ranging from going to museums and visiting churches to eating at a restaurant and visiting a club. In order to acquire all of this information, many different sources of data are needed.

One can imagine that in order to retrieve information about restaurants a different data source is invoked than for retrieving information about monuments in a certain area. Besides using different sources for different kinds of data, it can also be the case that multiple sources can be invoked to retrieve the same kind of data. Take for example the retrieval of monuments. Information about monuments is provided by DBPedia as well as by LinkedGeoData. Even though it is likely that there is overlap between the data sources, some monuments that are contained within the LinkedGeoData dataset might not be present in DBPedia or the other way around.

If we were to use multiple sources for one kind of POI, this would imply the need for instance matching. We would have to check whether an instance from one source actually represents the same entity as an instances from the other source. This process could be facilitated by `owl:sameAs` relations, the Semantic Web way of indicating that concepts represent the same thing. However, Semantic Web data sources are not yet very common, making most instance

matching processes very complex and costly. It is hard to judge whether two things are the same, because the description of the two instances might differ. One can think of missing data in one source, conflicting data concerning the location, multiple POIs with the same name etc.

The above implies that we have to make two decisions: which sources to include and whether to include multiple sources per POI category. Since the main focus of this thesis is on effective POI selection and TSP solving and we do not want to focus on data retrieval, we decide to include one data source that can provide us with all sorts of information. We choose to use the LinkedGeoData dataset, because it includes a lot of different kinds of POIs, including: museums, sights, churches, shops, bars and restaurants.

Using only one dataset also has its drawbacks. First of all, the LinkedGeoData dataset does not contain information about every kind of POI, for example, squares. Also, the dataset might not be complete. When considering only one dataset we are highly dependable on this source and even though the LinkedGeoData dataset is very comprehensive, it is highly plausible that it does not contain, for example, every monument on the planet. However, we believe that the LinkedGeoData dataset is extensive and trustworthy enough to fit our needs within this project.

6.2 Pre-processing

In order to effectively use the data from LinkedGeoData, we have to do some pre-processing. We discuss the preference of using localized datasets that contain POIs of a single city over using a global dataset that contains all POIs in Section 6.2.1. In Section 6.2.2 we discuss the mapping of the LinkedGeoData dataset categories to our own ontology. We continue with Section 6.2.3, containing a description of additional properties that we decide to add to the dataset.

6.2.1 Localized Datasets

The LinkedGeoData dataset is a global dataset that contains POIs from all over the world. Naturally, the first idea that comes to mind is to feed the global dataset to the city trip planner and base the city trips on this single dataset. However, this makes the point selection very complex and the amount of possible sets $\mathcal{T}$ immense. When using a very naive method of selecting POIs, it might even occur that one selects the Empire State building in New York and the Colosseum in Rome in the same set $\mathcal{T}$. Of course, the selection strategies do take the notion of distance into account, but they still have to consider the position of every POI and with the size of this dataset (currently 66 million triples[1]) this is a costly process.

Even though the selection strategies will not select POIs from all over the world, we have to find a way to consider only POIs from the city a user requests. In order to this, we have several options. The first option is to divide the dataset

[1] `http://linkedgeodata.org/Datasets`

into a tree structure, where each node of the tree represents a grid. This grid represents a small part of the total geospatial area of the dataset. This could, for example, be done by using R-Trees [26]. The second option we consider is to (semi)-automatically create a dataset per city. We can compare the points in the dataset with the city boundaries, adding the points within the boundary to the dataset corresponding to the city.

Using methods that automatically generate the tree-structures raise some problems. Some cities might be spread over multiple grids, which means that we cannot simply only select a single grid and select POIs from this grid. Another bigger problem is the fact that we do not know which city is in which grid. This means that we have to find a way of keeping track of which city is where. This information is generally not available in R-trees or other spatial indexes.

Because of these problems we choose to use the second method: creating a separate dataset for each city. Of course, this method also has its drawbacks. We have to specify bounding boxes for each city and write a small piece of software that, based on a bounding box, creates a dataset. Since the focus of our thesis does not lie with the data, we choose to create datasets in this way for two cities: Amsterdam and Milan. The reason to choose these two cities is that the quality and the size of the two datasets is very different, which makes it interesting to compare the performance of the selection strategies on both datasets. We elaborate on these differences in Section 6.3 and Section 6.4.

6.2.2 Ontology Mapping

As described in Section 4.4, we have designed an ontology in order to structure our data. The LinkedGeoData dataset is structured according to its own ontology. Therefore, we have to do a mapping that matches the classes from the LinkedGeoData ontology to our own ontology. This could, for example, be matching the TourismHotel class from LinkedGeoData to our own Hotel class.

We only match classes that we believe are interesting for the city trip planner, thereby removing POIs that are instances of other classes from the dataset. The complete matching schema can be found in Appendix B.

6.2.3 Added Properties

While the LinkedGeoData dataset provides us with all kinds of information about POIs, there are some properties that are missing:

- Rating
- Opening time
- Closing time
- Service time

For each POI we enter the rating manually with the help of review sites such as Tripadvisor[2], Hostelworld[3] and Booking.com[4]. For each class of our ontology, we determine a suitable opening and closing time. For example, a pub should have different opening times (21:00 - 0:00) than a museum (10:00 - 16:00). The service time is also based on the class and to ensure some variety in the data we add a random factor to it.

<http://linkedgeodata.org/triplify/node1012644715>
<http://www.w3.org/2003/01/geo/wgs84_pos#geometry>
"POINT(2.54237 48.8299)"

Figure 6.1: Redundant triple removed from the LinkedGeoData dataset.

Figure 6.1 shows a redundant triple included in the LinkedGeoData dataset. This is a concatenation of the latitude and longitude values of a point. According to the LinkedGeoData website[5], this triple is used by the Virtuoso universal server software[6]: "The node positions use of a Virtuoso-specific data type that is not supported by the open source edition". We can deduct this information from the latitude and longitude triples and therefore we decide to remove these triples from our dataset. In addition to saving storage space, this will prevent later problems with loading Virtuoso specific data types, in non Virtuoso systems. When we add additional data sources, with richer information about objects, we can choose to filter even more triples.

6.3 Data Analysis

In this section we describe the two datasets used for evaluating the strategies discussed in Section 5. We start with the Milan dataset, because this dataset is related to the scenario introduced in Section 3.1. The second dataset is the city of Amsterdam. The distribution of points over classes C is given in addition to maps of the chosen area.

In Figure 6.2 one can find the distribution of POIs over the different classes present in the Milan dataset. This figure shows that the points of interest are not evenly distributed over the classes. For example, there are many more instances of the class hotel, church and café than of others. Still, the exact numbers are not very high. One can imagine that the number of restaurants in the city of Milan is much higher than 10. The total number of POIs in this dataset adds up to 70. From this number we can conclude that many POIs in Milan are missing from this dataset.

[2] http://www.tripadvisor.com
[3] http://www.hostelworld.com
[4] http://www.booking.com
[5] http://linkedgeodata.org/Datasets
[6] http://virtuoso.openlinksw.com/

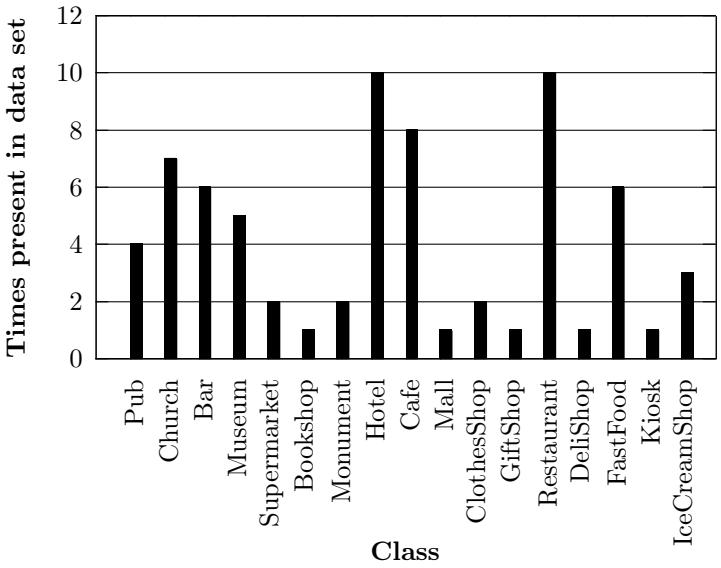


Figure 6.2: The amount of instances per class in Milan

The map of the area of Milan we choose to include in our dataset and all the POIs in this area can be found in Figure 6.3. Here we can see that the highest density of POIs lies around the square of the Duomo, but still there are POIs scattered all over the city. Note that this figure contains all the POIs and not only POIs of the classes that we selected as described in Section 6.2.2.

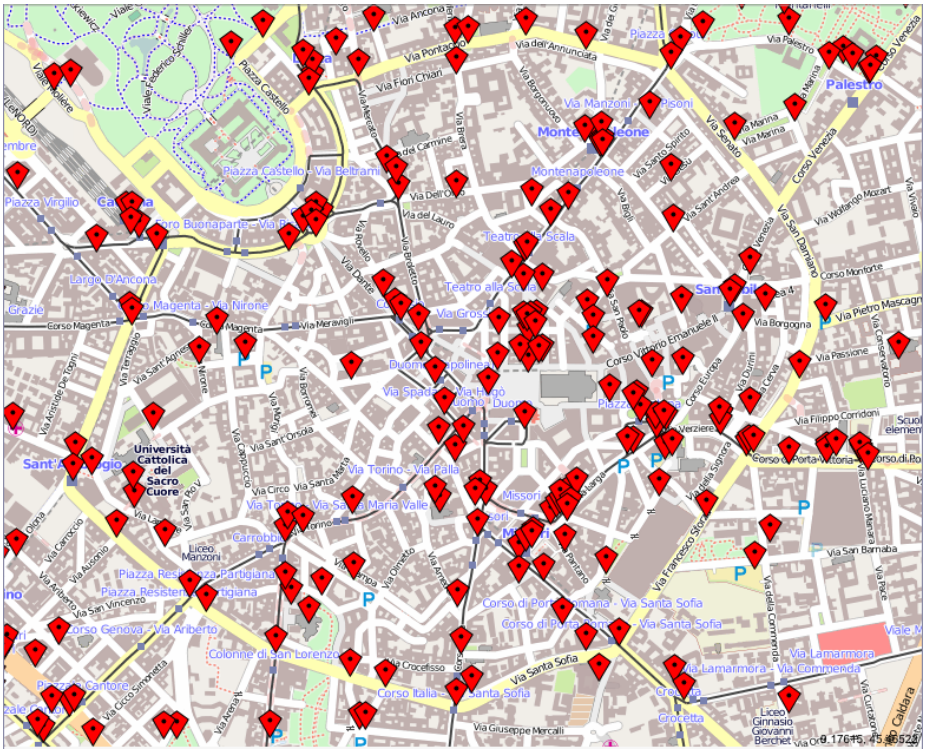


Figure 6.3: Map representing the Milan dataset, each pointer represents a POI.

In Figure 6.4 the distribution of POIs over the different classes included in the Amsterdam dataset can be found. One thing that immediately catches the eye is the fact that the amount of POIs in this dataset (667) is much higher, while the surface of the selected area is about the same. This can mean two things. Either Amsterdam is a much more interesting city or there are more people working on adding POIs in the region of Amsterdam. Even though we believe Amsterdam is a nice city, the second option is more likely.

Another thing that stands out is the distribution over the different classes. As with the Milan dataset, the distribution of POIs over the classes is not equal. The differences in the amount of POIs per class is even higher. Another point that can be made is that there are many more classes represented in this dataset than in the Milan dataset.

It is also worthwhile mentioning that location specific classes appear in both the datasets. In the Italian dataset the ice cream shop class can be found, whereas in the Netherlands, where cheese is very popular, cheese shops can be found.

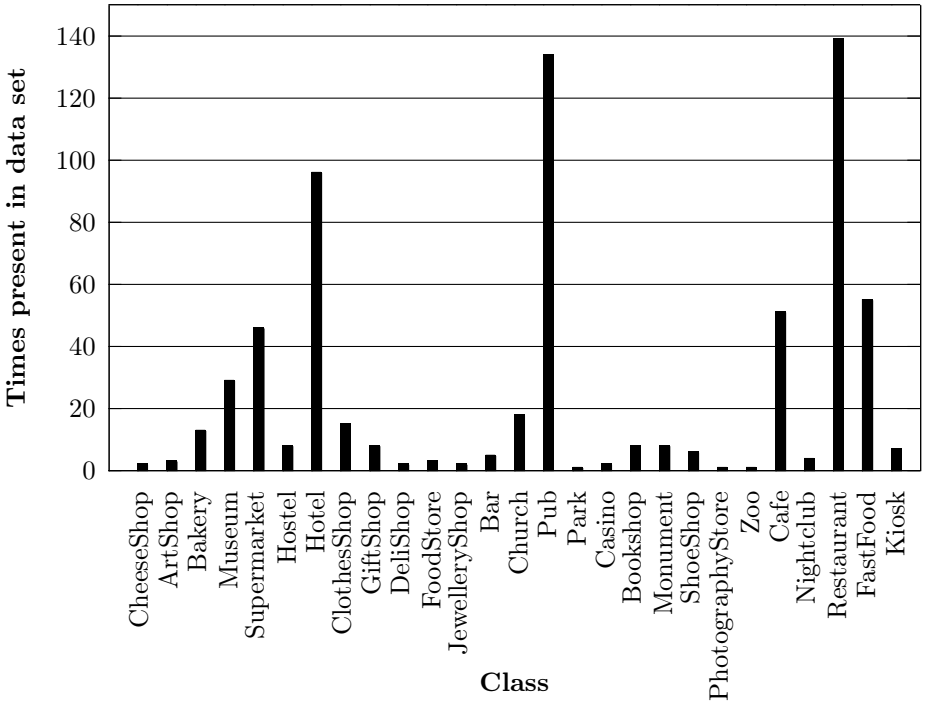


Figure 6.4: The amount of instances per class in Amsterdam

The map of the area of Amsterdam, which we consider to cover an interesting part of the city, in addition to the POIs located inside this area, can be found in Figure 6.5. This figure shows an even higher concentration of POIs in certain

areas than in the case of the Milan map. Especially the area near the canals, the so-called "grachtengordel", contains lots of POIs. Since this is the most lively and interesting part of the city, it is interesting to see that this is also represented in te distribution. Another interesting area is near the Vondelpark, at the bottom left, where a lot of museums reside.

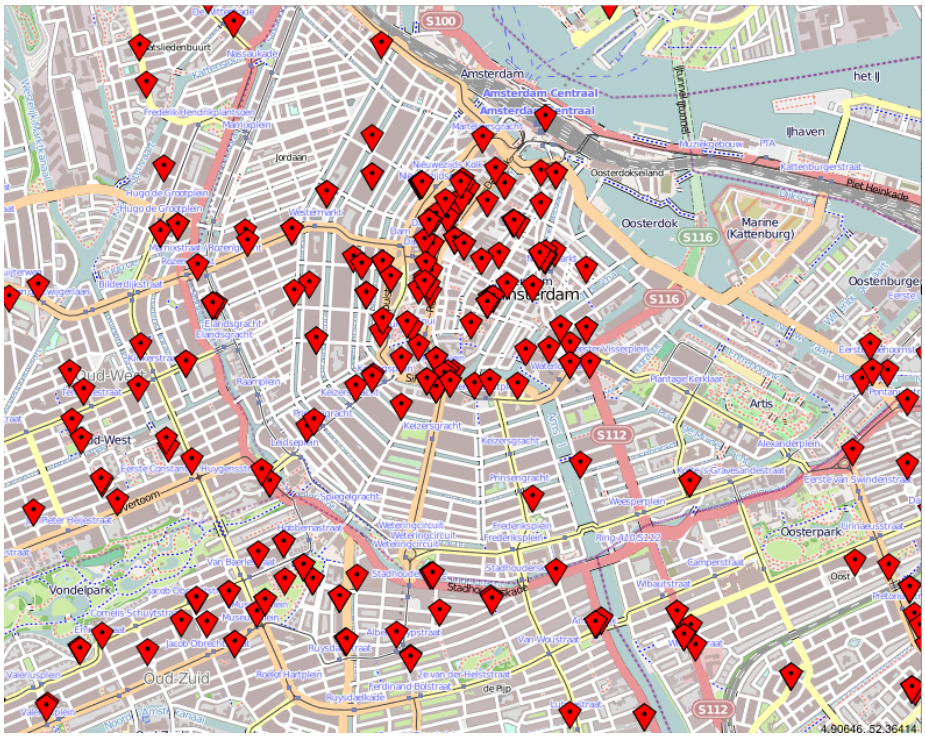


Figure 6.5: Map representing the Amsterdam dataset, each pointer represents a POI.

6.4 Data Quality

We conclude this chapter with a discussion of the quality of the datasets. As discussed above, especially in the Milan dataset, a lot of POIs are missing. Besides missing POIs, it can also happen that information of POIs that are present in the dataset is missing, or is wrong.

Concerning the Milan dataset, the most famous POIs are not present in the dataset. The Duomo, which can be considered to be the the major touristic spot of Milan, is missing as well as the famous theater La Scala. Also a lot of labels of POIs are missing, which makes the POIs unidentifiable. Overall, the quality of the information differs per POI, some POIs are described extensively by using lots of properties and others contain only the coordinates, a class and

a URI.

The quality of the Amsterdam dataset is much higher. We have not identified any major touristic spots that are missing and almost all POIs contain at least a label, coordinates, class and URI. Also, many POIs contain additional information such as a website and for example the kind of cuisine, when considering restaurants.

Chapter 7

Implementation

In this chapter, we describe the implementation process of the city trip planner, as specified by to the problem description introduced in Chapter 3 and the proposed architecture in Chapter 4. We start with Section 7.1 by discussing the implementation of the two most important data objects, the point of interest object and the class object. Section 7.2 discusses the implemented testing environments, which are used for evaluating the quality of the selection strategies. We conclude this chapter by discussing the end product of this project in Section 7.3: the implementation in LarKC. Besides being the working application, the LarKC implementation is also used to carry out experiments evaluating the time it takes to successfully generate city trip plans.

7.1 Implementation Of Data Objects

The implementation of the different objects is not a straight translation of the conceptual design describing all objects as stated in Section 4.2. Only the objects class and POI have been modelled as separate objects in the Java implementation. The other objects are, for the sake of simplicity, modelled as properties of these two objects. In Section 7.1.1 we elaborate on the translation of the POI object from the conceptual model into the one that is implemented. In Section 7.1.2 we repeat this process for the class object.

7.1.1 Point Of Interest

The implementation of the POI object incorporates all the information presented in Figure 4.2 except for the reviewer, because we make no use of reviewers in the current version of the application. The other objects, except for class, are modeled as properties of the POI object. The point object containing the latitude as well as the longitude is modelled as two separate properties: latitude and longitude, both of type float. The rating is modelled as a property of type double and is a value between 0 and 1. The duration description object

is modelled as an integer property, representing minutes. The opening hours object is modelled as two separate properties representing opening time and closing time in the form of strings in the format "hh:mm".

The reasons for these simplifications are practical. If we were to implement all the properties as objects, it would be much harder to maintain a clear and concise overview of where things are stored. This would make the job of programming much harder and the code unreadable. While it is debatable, we make the assumption that each of the objects that have been turned into properties are unique for a single POI. This means that they are not reused by other POIs, removing the absolute need to save information in separate objects. We realize that, especially for the Point object, this assumption is radical, since multiple POIs can be at the exact same location. For example, a museum shop is located inside a museum, but they are both separate POIs. However, POIs that are at the same location are not common and for the few cases in which it happens it is possible to store their locations separately.

Besides the properties that are described in Figure 4.2, Section 4.2, we have added another property: the unique resource identifier (URI). The URI is used to denote a unique POI and is conform the Semantic Web, where every unique entity is denoted by a URI. For each POI, we use the URIs that are already present in the LinkedGeoData dataset.

7.1.2 Class

The implementation of the class object differs less from the conceptual model than the POI. In fact, the only real difference is its name. Since Java already uses the term class for internal purposes, it is not possible to create an object named class. Instead we changed its name to category, which, in our eyes, represents the same concept.

As with the POI object, we also added a URI property to the class object in order to denote unique classes. Since we created our own OWL ontology that contains all the classes we need, we already have these URIs available.

7.2 Testing Environments

Instead of using LarKC to evaluate the quality of different strategies, we develop a test environment which sole goal is to run the strategies on different datasets, in order to get a reliable and good measure of the quality of the generated city trip plans. The main reason for developing this test environment is to be able to effectively run a baseline. This baseline enables us to compare the two developed strategies to the optimal and worst situation. Since we let the baseline generate all possible sets $\mathcal{T}$ of a scenario set in Milan (which sums up to 4.379.340 sets), we have to efficiently utilize the TSPTW solver, which is the slowest component in the process. The test environment is therefore multi-threaded, which enables it to solve multiple $\mathcal{T}$-sets at the same time. In our test

environment we immediately write all our results to files in order to keep the amount of memory used to a minimum.

In Section 7.2.1 we discuss the test environment that is used to actually run the strategies and save the solutions generated by the TSPTW solver. We continue with the grading environment in Section 7.2.2. This environment utilises the grading function as introduced in Equation 3.10 to evaluate the results obtained by the test environment. We finish with the evaluation environment in Section 7.2.3. This environment compares the results from strategies using the chosen evaluation measure, the cumulative gain.

7.2.1 Test Environment

The main process that controls all other processes can be found in the *TestEnvironment.java* file. The other important files are the *SelectionThread* files, which represent the selection strategies, and the *TSPThread.java* file. In order to provide ourselves with a fast test environment, we execute each distinct main process in a separate thread to create a multi-threaded Java process. Such a process can take full advantage of the multi-core architecture of a testing machine.

It is hard to keep track of which Java threads are running, which is why we channel the communication between the separate parts using a queue. The queue is filled with $\mathcal{T}$-sets by the selection strategy. Within the main program there is a while loop that checks whether there are sets present in the queue and, if so, it pops the first element and calls a TSP thread to execute the TSPTW solver on this particular $\mathcal{T}$-set.

The test environment requires as input the dataset, the created ontology, a set of user preferences and a specified selection strategy. The output is a text file, with a different plan on each line. In front of each line there is a 1 or a 0, representing whether the TSPTW solver is successful (in case of the 1) or unsuccessful (in case of a 0). The rest of each line is an ordered sequence of URIs where each URI represents a POI.

7.2.2 Grading Environment

The grading environment is executed after the test environment. It takes the output of the test environment as input, grades the city trip plans and outputs the graded city trip plans into a new file. It uses almost the same format for its output as the test environment, but instead of a 1 or a 0, it puts the grade of the plan in front of each line. Besides this, one can also request to sort the output, which can be used to generate the optimal/worst solution.

The main file of the grading environment is the *gradingEnvironment.java* file. This file executes the whole process ranging from reading the file to calculating the ratings and outputting the result. The only process that is carried out by another file is the sorting process of the results. This is done by the *Sort.java* file that uses an adapted version of a merge sort implementation that was found

on www.codeodor.com[1]. We were forced to implement this peculiar sorting method in order to avoid *out of memory errors* while handling the big result files generated by the baseline strategy.

7.2.3 Evaluation Environment

The final part that completes our testing environments is the evaluation environment. The evaluation environment takes multiple files as input where each file contains the results from a different strategy. The evaluation environment uses the Discounted Cumulated Gain evaluation measure (DCG) as introduced in [29] to compare the different results.

DCG is an evaluation method originating from the field of information retrieval and is developed in order to credit information retrieval methods for their ability to retrieve highly relevant documents. In our case, the information retrieval methods are the selection strategies and the documents the city trip plans. We will provide a more elaborate description of the discounted cumulated gain in Section 8.1.1.

Using the results obtained by the DCG method, the evaluation method outputs a comma separated values file (CSV) that contains the results of this comparison. This CSV file can, on its turn, be used to generate the graphs that we use in the evaluation in Section 8.2.

7.3 LarKC

As described in Section 2.2, LarKC applications consist of multiple small building blocks that are called plug-ins. In Chapter 4, one can find an extensive description of the plug-ins and their interactions. In Chapter 4 we also expressed the desire for parallel execution one of the plug-ins. While we did succeed in making the Test Environment execute the TSP Threads in parallel, we do not incorporate this in the LarKC workflows.

In cooperation with our supervisor, Emanuele Della Valle, we have made an effort in trying to figure out ways for getting parallel execution of plug-ins working in LarKC. An e-mail discussion between our supervisor and people from the LarKC Consortium resulted in an example, which was not fully functional. For this reason in addition to a lack of good documentation, we are at the moment not able to include parallel execution in the LarKC workflows.

In the remaining of this section we discuss the implementation of the plug-ins and what problems we encountered.

7.3.1 Planning Decider

The planning decider plug-in is the core of the application, it controls the process and prepares the execution of the city trip generation workflow, by loading

[1] `http://www.codeodor.com/index.cfm/2007/5/14/Re-Sorting-really-BIG-file---the-Java-source-code/1208`

all the needed data into the LarKC datalayer. The workflow loading the PlanningDecider can be found in Figure 7.1. When the PlanningDecider is initialized, parameters of the LarKC workflow description are read. In Figure 7.1 at line 19, one can find a parameter indicating the location of the file containing the POI data. Line 20 represents the parameter describing which strategy to use.

```
1: @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
2: @prefix larkc: <http://larkc.eu/schema#> .
3: @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
4:
5: # Define a plug-in
6: _:PlanningDecider a <urn:eu.larkc.plugin.PlanningDecider> .
7:
8: # Define a path and set the input and output of the workflow
9: _:path a larkc:Path .
10: _:path larkc:hasInput _:PlanningDecider .
11: _:path larkc:hasOutput _:PlanningDecider .
12:
13: # Connect an endpoint to the path
14: <urn:queryendpoint> a <urn:eu.larkc.endpoint.sparql> .
15: <urn:queryendpoint> larkc:links _:path .
16:
17: # Define plug-in specific parameters
18: _:PlanningDecider larkc:hasParameter _:param .
19: _:param larkc:filePath "../data/Milan.nt" .
20: _:param larkc:strategy "RadiusStrategy" .
```

Figure 7.1: The workflow loading the PlanningDecider plug-in, written in *Notation 3*.

The LinkedGeoData dataset comes in the N-Triples RDF format, which is an advantage because this makes it straightforward to load into the LarKC data layer. The file path provided by the workflow is used to indicate which dataset to load, one containing information about points in Milan or one with information about points in Amsterdam. LarKC comes with a N3 parser, which is used to load the data into the data layer at the moment plug-in is initialized.

In addition to the information concerning POIs, the PlanningDecider also loads the user preferences. We choose to store these preferences in a single text file that contains $\mathcal{UC}$, $\mathcal{UI}$ as well as $\mathcal{NUC}$. We choose to use files, because in the future these files could easily be generated by a front-end according to the preferences a user enters.

Also important is the loading and storage of the parameters that are used throughout the application. At the moment the parameters are non customizable, which means they are the same for every user and are loaded into the datalayer in a hard-coded fashion. Of course, this is not a desirable situation, but for the moment it is sufficient in order to prove or disprove our hypothesis.

Parameters that are stored in the data layer are:

- minimumLat: the minimum latitude present in the dataset.
- minimLon: the minimum longitude present in the dataset.
- startTime: the start time of plan.
- endTime: the end time of plan.
- speedMs: the travel speed in meters per second.
- desired: the desired amount of resulting plans.
- city: the city in which the application searches for plans. This parameter is determined based on loaded dataset.

The user can also specify his or her preferred selection strategy. It is only possible to choose between the Radius or the DTR selection strategy, since running the Baseline takes over 3 days on the Milan dataset. The Baseline will take decades to finish on the Amsterdam dataset. Based on the parameter found in Figure 7.1 at line 20, the PlanningDecider is able to load two distinct workflows: one containing the DTR strategy for point selection, the other containing the Radius strategy as point selection method. The sequence of plug-ins of the Radius strategy workflow, is loaded by the Notation3 workflow description of Figure 7.2.

```
1: @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
2: @prefix larkc: <http://larkc.eu/schema#> .
3: @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
4:
5: # Define plug-ins
6: _:PointSelecter a <urn:eu.larkc.plugin.PointSelecter> .
7: _:TSPTWReasoner a <urn:eu.larkc.plugin.TSPTWReasoner> .
8: _:RankDecider a <urn:eu.larkc.plugin.RankDecider> .
9: _:Cartographer a <urn:eu.larkc.plugin.Cartographer> .
10:
11: # Define a path and set the input and output of the workflow
12: _:path a larkc:Path .
13: _:path larkc:hasInput _:PointSelecter .
14: _:path larkc:hasOutput _:Cartographer .
15: _:PointSelecter larkc:connectsTo _:TSPTWReasoner .
16: _:TSPTWReasoner larkc:connectsTo _:RankDecider .
17: _:RankDecider larkc:connectsTo _:Cartographer .
```

Figure 7.2: The workflow loading the plugins of the Radius workflow, written in *Notation 3*.

After the workflow is finished and the Planning Decider receives a resulting set of RDF statements containing the plans, it executes on these statements the user query it received on its input and returns the results to the query endpoint.

7.3.2 Point Selecter

Because we have designed and developed two different selection strategies, we have also created two different point selecter plug-ins. These are the Radius Point Selecter and the DTR Point Selecter plug-ins. In this section we describe the points that they have in common considering their implementation. The algorithms used for finding points are discussed in depth in Section 5.

Each Point Selecter plug-in needs to interact with the LarKC datalayer in order to function correctly. The sets $\mathcal{UC}$, $\mathcal{NUC}$ and $\mathcal{UI}$ are needed in order to be able to select POIs corresponding to the user preferences. These preferences are retrieved using a SPARQL query. The number of desired sets is also fetched from the datalayer, so the point selection process can continue creating sets until the desired number is reached. A set is represented by triples including the URIs of the selected POIs. The found sets are added to a *SetOfStatements* object. Figure 7.3 depicts an example set containing four selected points.

```
1: @prefix lgd: <http://linkedgeodata.org/triplify/>
2: @prefix set: <set>
3:
4: set:set1 set:contains lgd:node725239190,
5:                       lgd:node432860311,
6:                       lgd:node442827340,
7:                       lgd:node261631356;
```

Figure 7.3: Representation of a set including four points, selected by a Point Selecter.

7.3.3 TSPTW Reasoner

As stated in Chapter 4, we decide to use a TSPTW implementation developed by Rodrigo Ferreira da Silva and Sebastian Urrutia [17]. We received the source code from the authors, but whereas LarKC and our Test Environment are all coded in Java, this TSPTW solver is coded in the C++ language.

C++ is a fast, platform specific language. Since solving the TSPTW problem is a demanding process, we decide not to port the TSPTW solver to Java. Instead we use the provided C++ implementation and compile three different versions from the source code, a Linux version, a Mac version and a Windows version. The Test Environment and LarKC plug-in check on which operating system the application is currently running, before executing one of the three programs, from within Java.

The provided TSPTW implementation uses text files as input. Since especially the baseline creates a large number of input sets, the use of files would

slow down the process, because of the heavy load on the file system. This is why we decide to modify the source code, so that the information needed by the solver can be communicated using a commandline argument, thereby bypassing the file system.

The input text files included x and y coordinates of the points, a demand value which was always set to 0, a ready time and due date defining the time window and a service time. The solver requires non common units for these properties and therefore the data of the LinkedGeoData dataset has to be converted. This is done using a *TSPData.java* object, which stores the data of a current set and converts it to a for the solver usable format. When the solver finishes, a plan P is created from the stored POI data, using the order obtained through the TSPTW solver. The resulting plans are added to the LarKC datalayer. Each triple on the output of the TSPTW plug-in links the URI of a POI to its position in the sequence of the plan. Figure 7.4 shows the plan resulting from the four points of Figure 7.3.

```
1: @prefix lgd: <http://linkedgeodata.org/triplify/>
2: @prefix plan: <plan>
3: @prefix notExecutable: <notExecutable>
4:
5: plan:set1 plan:poi1 lgd:node442827340;
6:           plan:poi2 lgd:node261631356;
7:           plan:poi3 lgd:node725239190;
8:           plan:poi4 lgd:node432860311.
9: plan:set2 plan:message notExecutable:timeOut
```

Figure 7.4: Representation of a plan including four points ordered by the TSPTW reasoner and a second plan, for which the TSPTW solver was unable to find a solution.

The solver is not always able to find a solution for a given set of points $\mathcal{T}$, but will keep on trying until it is stopped. In order to keep computational resources available, we decide to stop the solver after a second. After more then a second of trying to solve the problem, the algorithm is unlikely to find a solution at all. At this point a triple with as predicate plan:message and object notExecutable:timeOut is created and added to the datalayer. This triple indicates that the solver was unable to solve this particular set.

7.3.4 Rank Decider

The Rank Decider plug-in uses Equation 3.10 to assign grades to plans P. These plans have to be reconstructed from the datalayer, which is done using the predicate of the triples used to indicate the sequence of the points (see Figure 7.4). For each point in the plan some properties have to be known in order to be able to judge the plan: the class the POI belongs to, the rating, the latitude, the longitude and the service time. This is achieved by firing a SPARQL query at

the datalayer for each POI, retrieving the corresponding properties. In addition to the plans, also the user preferences, the parameters and the ontology are loaded.

The grade which is assigned to a plan, is added to a *plan.java* object. The plans are sorted using their assigned grades, using the sort method provided by the *Collections* class of Java. Triples with the respective grade and rank of a plan P are added to a *SetOfStatements* object and send to the Cartographer plug-in. An example of these triples can be found in Figure 7.5.

```
1: @prefix lgd: <http://linkedgeodata.org/triplify/>
2: @prefix plan: <plan>
3:
4: plan:set1 plan:hasGrade "1.276146584398369";
5:           plan:hasRank "2".
6: plan:set2 plan:hasGrade "1.2878256525927898";
7:           plan:hasRank "1".
8: plan:set3 plan:hasGrade "1.2643784764454862";
9:           plan:hasRank "3".
```

Figure 7.5: Representation of grades and ranks of three sets, determined by the RankDecider.

7.3.5 Cartographer

The Cartographer plug-in creates files which can be used to display the obtained city trip plans in a human readable format. The first file created is a GPS Exchange Format (GPX) file. GPX files contain information about waypoints and routes. This information can be loaded by web pages or mobile devices and are rendered as lines, depicting a route. Figure 7.6 shows an example of a generated GPX file, of a route in Milan.

```
1: <?xml version="1.0" encoding="UTF-8" ?>
2: <gpx version="1.1">
3: <trk><name>Plan#1</name>
4:     <trkseg>
5:         <trkpt lat="45.4913216" lon="9.2953455"><ele>0.0</ele>
6:         <time>2011-11-11T11:11:01Z</time></trkpt>
7:         <trkpt lat="45.456426" lon="9.1734675"><ele>0.0</ele>
8:         <time>2011-11-11T11:11:02Z</time></trkpt>
9:         <trkpt lat="45.4885364" lon="9.1654404"><ele>0.0</ele>
10:        <time>2011-11-11T11:11:03Z</time></trkpt>
11:    </trkseg>
12: </trk>
13: </gpx>
```

Figure 7.6: Example of a GPX file representing a route of three points in Milan.

At the moment the Cartographer adds points to the GPX files, which describe straight lines from one POI to the next. In the future, routing mechanisms could be used to add more waypoints, thereby providing users with more detailed info. The point of interest data we use does not include the elevation of the points, which is why we set the <ele> element of the GPX file to 0.0. The TSPTW Reasoner outputs only the sequence of POIs, not the related time information. For this reason we use fictive time elements, which retain the sequence information of the city plan generated by the system.

The second file is a text file containing information about the points of interest included in the city plan. The values include a short description and the label of the POI. This data is used to enrich the information displayed on the OpenStreetMaps depiction of the route.

Chapter 8

Evaluation

In this chapter we evaluate the city trip planner and the developed selection strategies using the test environments and the LarKC application. We perform multiple experiments in order to evaluate the performance levels.

As described in Chapter 6, we evaluate the strategies on two different urban environments: Amsterdam and Milan. Where the Milan dataset contains only 70 POIs, the Amsterdam dataset contains 667 POIs. As a consequence, much more possible solutions can be found in the Amsterdam environment than in the Milan environment. In table 8.1, one can find the number of possible sets for the two datasets given a maximum number of points in a $\mathcal{T}$-set (upper bound) of 8.

dataset	**# of Possibilities**
Milan	4.379.340
Amsterdam	over $75 \cdot 10^9$

Table 8.1: Amount of possible sets with an upper bound of 8.

We cannot execute the baseline selection strategy on the Amsterdam dataset, because of its size. Given the fact that the upper bound is set to 8 and considering the tourist scenario introduced in Section 3.1, the Baseline takes around 39 hours to complete on the Milan dataset. Using these numbers, we can roughly calculate that the Baseline would take over 75 years to complete on the Amsterdam. This makes it impossible for us to run the Baseline on the Amsterdam dataset within our current time-frame, leaving us without an ideal situation to compare our strategies with on this dataset. However, the experiments on this dataset are still worthwhile, because results can indicate that a suitable plan can be found within a short period of time, given the fact that there is an extreme amount of possible solutions.

In Section 8.1 we describe the method of evaluation by discussing the used evaluation measure and the experimental setup. We present the results of these experiments in Section 8.2.

8.1 Method

In this section we describe the method of evaluation. We start in Section 8.1.1 by elaborating on the evaluation measure we use, which is the Cumulated Gain. Next, we discuss the minimal performance we have to obtain in order to confirm the hypothesis in Section 8.1.2. Finally, we discuss the different settings for the experiments in Section 8.1.3.

8.1.1 Cumulated Gain

In order to evaluate the results from the experiments, we use the Discounted Cumulated Gain (DCG) method. DCG originates from the field of Information Retrieval and is based upon two principles [29]:

- Highly relevant documents are more valuable than marginally relevant documents.
- The greater the ranked position of a relevant document, the less valuable it is for the user, because it is less likely that the user will ever examine the document.

In order to apply this evaluation method to our experiments we have to view the problem as an information retrieval problem. This means that the selection strategies can be seen as strategies that select plans from a finite set of possibilities and plans can be seen as documents. This is not a problem, since an urban environment with a finite set of POIs always has a finite set of possible day plannings, even though this number can be very high. DCG uses a ranked list as a basis for the evaluation, where the order of the ranked list represents the order in which the plans are found.

The first principle can be rewritten to the following: highly rated plans are more valuable than low rated plans. In the Cumulated Gain (CG) evaluation method, the rating of each plan is used as a gained value measure for its ranked position in the result list. Based on this principle the general cumulated gain method is formulated in Equation 8.1:

$$CG_i = \begin{cases} G_1 & \text{if i} = 1 \\ CG_{i-1} + G_i & \text{otherwise} \end{cases} \tag{8.1}$$

The second principle can be rewritten to the following: the greater the ranked position of a relevant plan, the less valuable it is for the user, because it is less likely the user will ever look at the plan. This means that the greater the rank of a plan, the smaller the share of the rating of the plan is added to the cumulated gain. Based on these two principles, the Discounted Cumulated Gain of a plan i is formulated in Equation 8.2:

$$DCG_i = \begin{cases} CG_i & \text{if i} < \text{b} \\ DCG_{i-1} + G_i/^b log_i & \text{if i} \geq \text{b} \end{cases} \tag{8.2}$$

The DCG is the main measure we use, but since we have an ideal situation on the Milan experiment, we can use a measure that judges the different strategies relatively to the ideal situation. This measure is called the Normalized Discounted Cumulated Gain (nDCG) and is calculated by dividing the DCG_i value of a strategy with the DCG_i of the ideal situation.

8.1.2 Success Criteria

Based on the hypothesis "The LarKC platform is able to generate 10 city trip plans of good quality within a minute, using strategies for selecting points of interest from the web of data combined with a TSPTW solver" we can formulate the success criteria as following:

- The developed strategies should be able to generate 10 city trip plans in less than a minute.
- The 10 city trip plans should be of good quality, meaning that the discounted cumulated gain after 10 plans should be closer to the optimal situation than to the worst situation.

The first point is an evaluation based on execution time. The execution time of the application is very important. Since finding the best solution using a baseline strategy takes two days on the Milan dataset and more than a few decades on the Amsterdam dataset, a lot of progress can be gained in this area. If we are able to find 10 plans within a minute we make sure that the speed of the application is satisfactory. In general, users want to see results very fast, but also like the opportunity to still have a choice, hence the 10 plans within a minute.

The second point is an evaluation based on quality. Execution time is important, but finding 10 plans within a minute that are of poor quality is not a desirable result. However, judging the quality of the plans is hard, because no related work has been done using selection strategies similar to ours. This means that the only possibility is comparing the results of the 2 strategies with each other and, on the experiments on the Milan dataset, with the optimal situation and the worst situation. We consider ourselves to be successful in the case where the DCG results after 10 found plans lie closer to the optimal solution than to the worst situation.

8.1.3 Experimental Settings

Here we discuss the different experiments that we perform on the two urban environments. We start by describing the experiments that are performed on the Milan dataset.

Milan

On the Milan dataset we run 3 different experiments. The first experiment is to either confirm or reject the execution time part of the hypothesis. We run

the strategies using the LarKC application until they have retrieved 10 plans and measure the time it takes to do this. Because measuring execution time is highly variable due to all sorts of processes that can interfere with the execution of the application, the experiment is run 10 times in order to be able to make valid conclusions. The independent variable we use is the selected strategy and the dependent variable is the execution time. The upper bound (maximum length of a plan) is set to 8, because this is a reasonable amount of POIs that can be visited in the current time window of 10 hours. For this experiment the following setup is used:

```
Apple Macbook pro 13"
Intel Core2Duo 2.4 GHz
4GB 1067MHz DDR3
Mac OS X 10.6 (Snow Leopard)
```

Because of earlier mentioned reasons in Chapter 7, we run the quality experiments using the developed test environments. After generating 10 plans, we calculate the DCG of both strategies as described in Section 8.1.1 and compare these results with the DCG of the optimal 10 plans and the worst 10 plans. Again, the selected strategy is the independent variable and the dependent variable is the DCG of the resulting plans. Using this experiment we can confirm or disprove the quality statement of the hypothesis. This experiment is run on the following setup:

```
Intel Quad Core 2.4GHz
2GB DDR3
Debian GNU/Linux 6.0
```

The experiments concerning quality are executed on a different setup, because of the excessive amount of time it takes to run a baseline. Therefore, a server machine running Linux equipped with four CPU cores was used in order to speed up the process.

In order to measure the performance of the strategies on the long term we perform a third experiment by letting the strategies retrieve 200 plans and compare the resulting DCG measures with the optimal and worst situation.

All quality experiments are run with the constants $K_{eagerness}$ and $K_{laziness}$ set to 1. The travel speed is set to 30 kilometers per hour. Another logical choice would have been to set it to walking speed (around 5 km/h), but we assume that tourists also take public transport to get around.

Amsterdam

On the Amsterdam dataset we perform the same three experiments. In this case we remove the upper bound of 8 in order to give the DTR selection strategy the chance to determine this intelligently. The results of the experiment concerning the plan quality are less useful, since we cannot calculate the optimal situation and the worst situation. This is why the experiment cannot be used for confirming or rejecting the quality statement of the hypothesis. However, since the

Amsterdam dataset is a much more extensive dataset, it is very useful to see if the strategies are still able to produce results quickly.

8.2 Results

In this section we present the results of the experiments. We start by presenting the results of the experiments concerning the execution time in Section 8.2.1 and continue by presenting the results concerning the quality of the retrieved plans in Section 8.2.2. Section 8.2.3 contains an in-depth analysis of the best results obtained by the city trip planner.

8.2.1 Results Of The Execution Time Experiments

In figure 8.1 one can find the results from the execution time experiments in the form of a box-and-whisker plot. Here we can see that the results are close to each other. The execution time for running the city trip planner on the Milan dataset are similar for both strategies. However, on the Amsterdam dataset the result of running the city trip planner with the DTR selection provides more variable results, including some high outliers. We have enlisted the standard deviation and the mean of each experiment in Table 8.2.

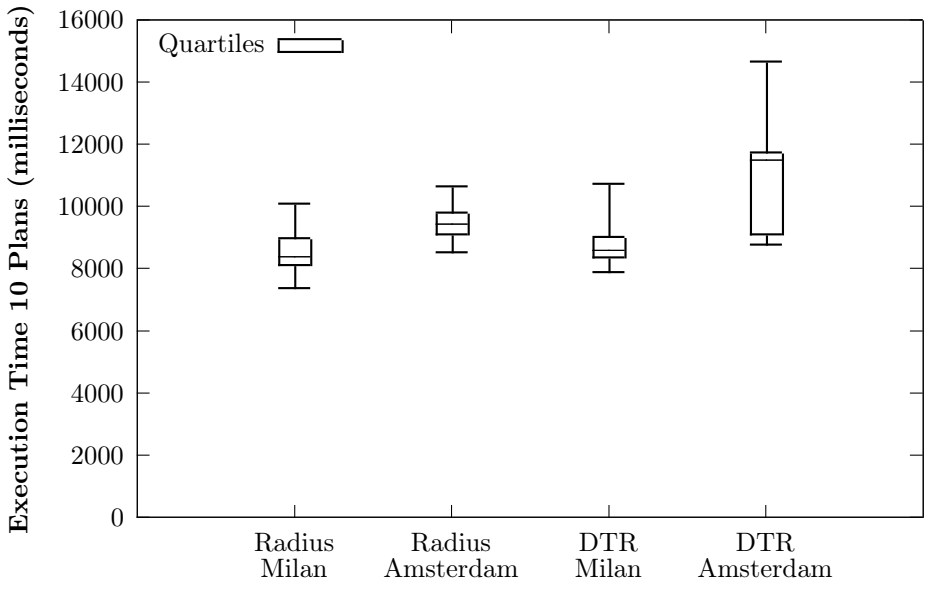


Figure 8.1: Box-and-whisker plot of 10 LarKC executions generating 10 plans. Bottom whisker is the fastest run, top whisker is the slowest run and the bar in the box is the median.

	DTR		Radius	
	mean	std. dev.	mean	std. dev.
Milan	8789.7 ms	785.45 ms	8502.1 ms	774.49 ms
Amsterdam	10933 ms	1843.47 ms	9483.4 ms	589.51 ms

Table 8.2: The mean and standard deviation of each experiment.

In order to be able to draw conclusions from our results we run a factorial ANOVA statistical test, where we have the two independent variables *dataset* and *selection strategy* and the dependent variable *execution time*. For the main effects, which are the effects of the dataset variable and the strategy, the degree of freedom is 1. For the interaction effect, dataset * strategy, the degree of freedom is also 1. For all effects, the degrees of freedom for the residuals are 36. We can, therefore, report the three effects from this analysis as follows:

- There was a significant main effect of the chosen dataset on the total time the city trip planner was running,
  $F(1,36) = 6.081$, $p<0.05$.
- There was a significant main effect of the strategy chosen for selecting POIs on the total time the city trip planner was running,
  $F(1,36) = 19.673$, $p<0.01$.
- There was not a significant interaction effect between the strategy selected and the dataset selected, on the the total time the city trip planner was running, $F(1,36) = 2.721$, $p=0.108$. This indicates that the DTR strategy as well as the Radius strategy are not proven to be reacting differently on the dataset chosen. Specifically, the execution time in milliseconds was similar for DTR (M=8789.70, SD=785.453) and Radius (M=8502, SD=774.491) on the Milan dataset; the execution time in milliseconds was higher for DTR (M=10933.00, SD=1834.47) than for Radius (M=9483.40, SD=589.508), but proved to be insignificant.

8.2.2 Results Of The Quality Experiments

When looking at the results in Figure 8.2, one can see that running the city trip planner with either one of the strategies produces similar results, where running it with the Radius strategy seems to perform a little better. In this graph we can also see that the results from both strategies are much closer to the optimal situation than to the worst possible situation.

In Figure 8.3 one can find the normalized discounted cumulated gain results. Here we can see that both strategies lead to results very close to the optimal situation. We run an independent t-test using the normalized discounted cumulated gain results in order to be able to conclude whether the city trip planner running one strategy performs significantly better than running it with the other. From the t-test that runs with range 10 we can conclude that on average, the city trip planner running with the Radius selection strategy performs significantly better

(M=0.949849580, SE=0.001711546) than running it with the DTR selection strategy (M=0.938511729, SE=0.003969151, $t(9)$=-8.295, p<0.01)

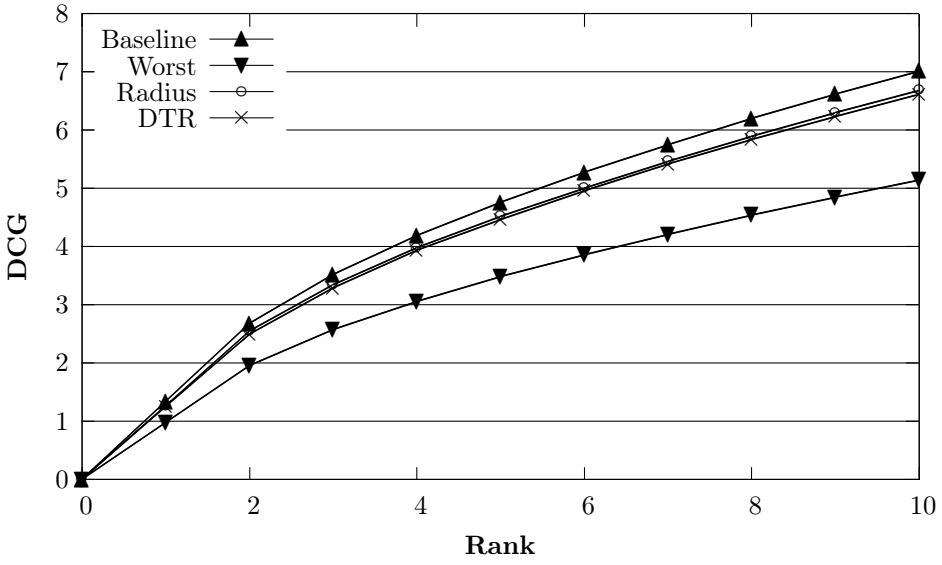


Figure 8.2: The discounted cumulated gain curves for the different selection strategies in relation to the optimal situation (baseline) and the worst possible situation (worst) on the dataset of Milan with range 10.

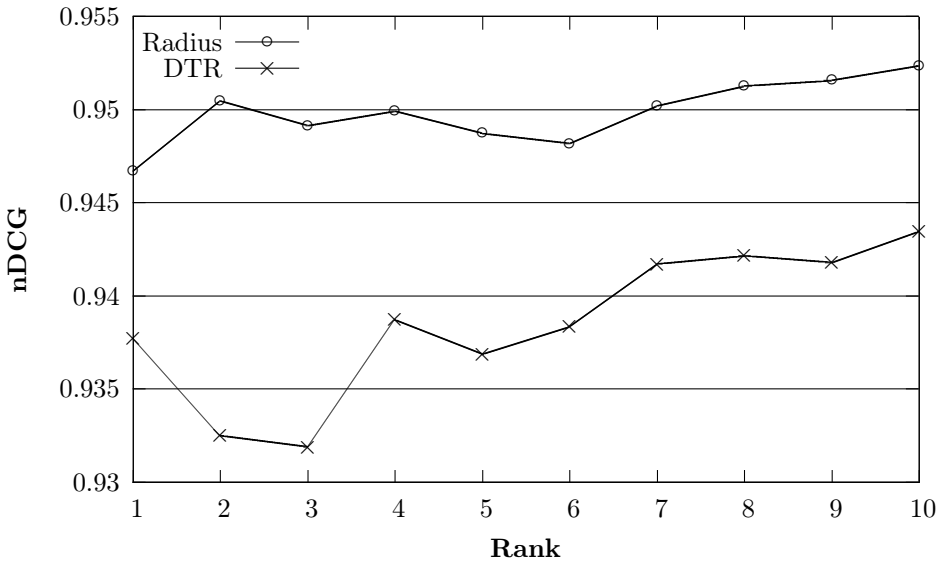


Figure 8.3: The normalized discounted cumulated gain curves for the different selection strategies on the dataset of Milan with range 10.

In Figure 8.4 one can find the performance of running the city trip planner with the different strategies on the long term. Here we can see that the results for both strategies are, again, very close to each other and also close to the optimal situation. However, on the long term it looks like the difference between the DCG of the optimal situation and the strategies seems to increase a little.

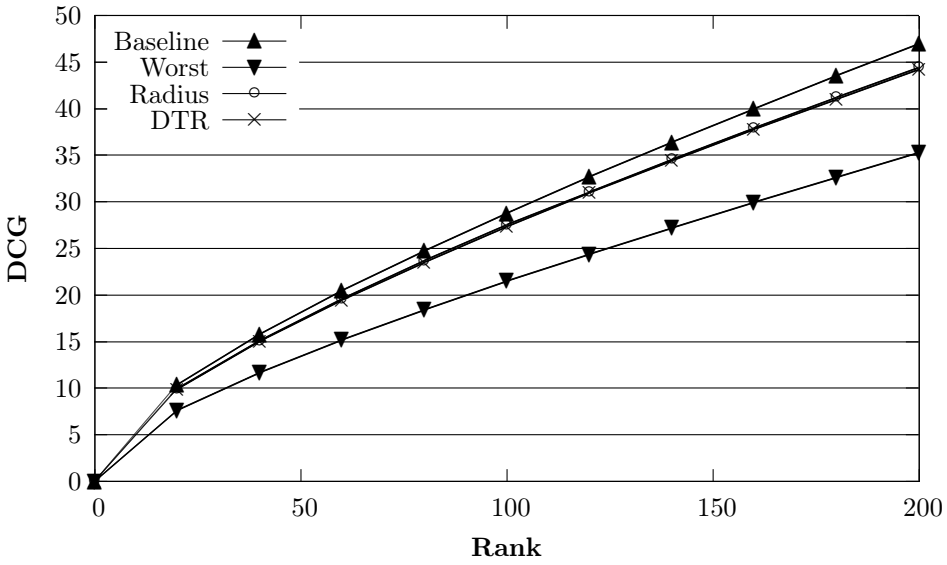


Figure 8.4: The discounted cumulated gain curves for the different selection strategies in relation to the optimal situation (baseline) and the worst possible situation (worst) on the dataset of Milan with range 200.

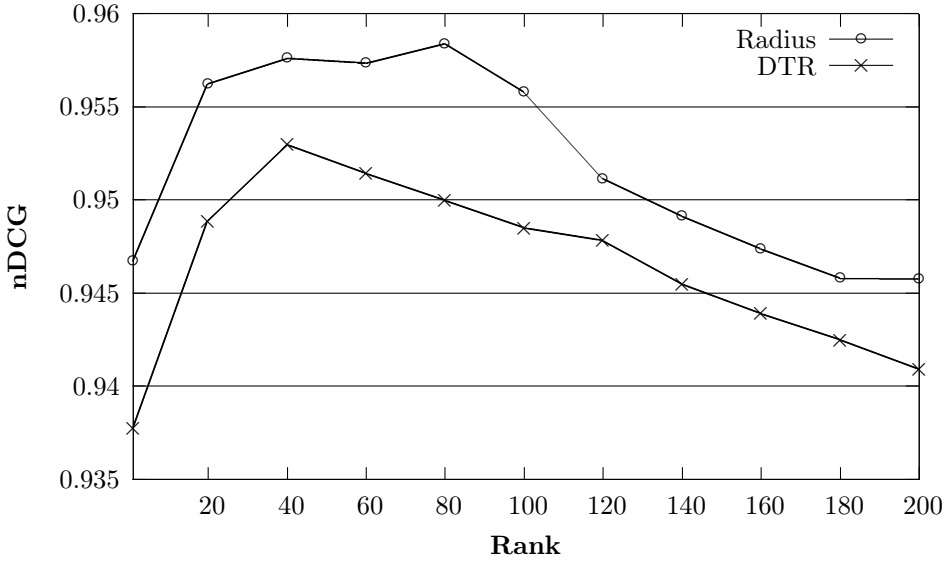


Figure 8.5: The normalized discounted cumulated gain curves for the different selection strategies on the dataset of Milan with range 200.

This can also be seen in Figure 8.5, where the normailzed DCG steadily decreases as more plans are generated. For these results we also run an independent t-test to be able to conclude whether running the city trip planner with one of the strategies performs better than running it with the other strategy on the long haul. From the t-test that runs with range 200 we can conclude that on average, the city trip planner run with the Radius selection strategy also performs significantly better (M=0.952631955, SE=0.000330793) than running it with the DTR selection strategy (M=0.947008774, SE=0.000287033, t(199)=-12.839, p<0.01)

In Figure 8.6 one can see the performance of running the city trip planner on the Amsterdam dataset for both strategies. Where in the Milan experiment the city trip planner running the radius strategy outperformed the one running the DTR strategy, the roles are now reversed. Since there is no optimal and no worst possible situation, we cannot draw hard conclusions about the quality of the performance, but when looking at the absolute numbers of the DCG, we can definitely see that plans found using the DTR selection strategy get a little higher grades than the plans found using the Radius strategy.

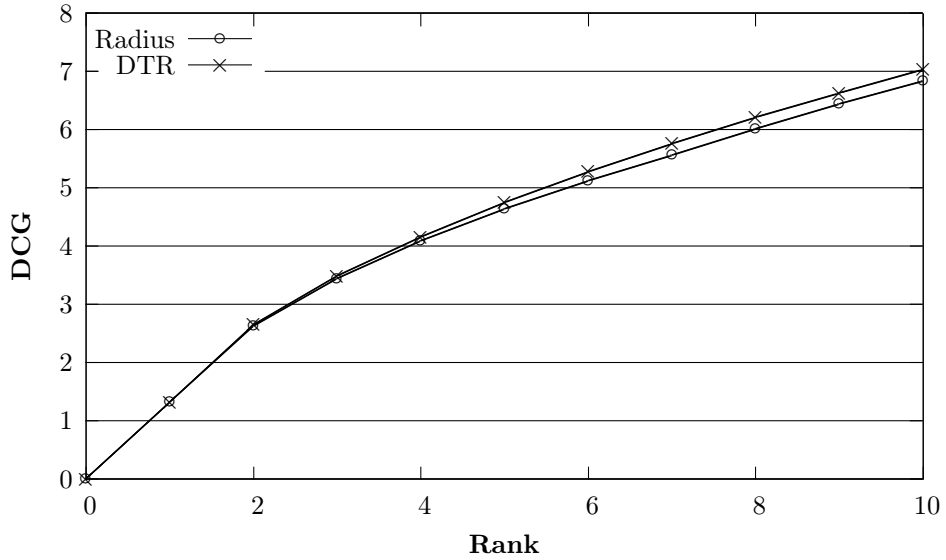


Figure 8.6: The discounted cumulated gain curves for the different selection strategies on the dataset of Amsterdam with range 10.

In Figure 8.7 we can see the performance on the Amsterdam data for 200 plans. Again we cannot draw hard conclusions, but also on the long haul running the city trip planner with the DTR strategy seems to produce better results.

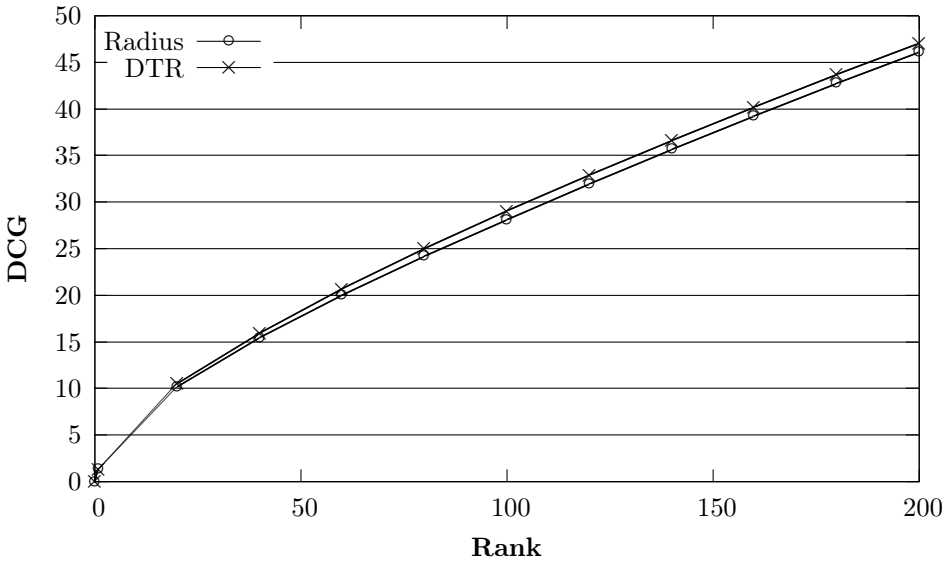


Figure 8.7: The discounted cumulated gain curves for the different selection strategies on the dataset of Amsterdam with range 200

8.2.3 Resulting City Trip Plans

In this section we discuss the content of the plans resulting from using the different strategies. For both datasets we discuss the first found plan by the Radius selection strategy as well as the first plan found by the DTR selection strategy. On the Milan dataset we also discuss the best possible plan, resulting from the sorted baseline strategy.

City trip plans Milan

In Figure 8.8 one can find the best possible plan of the Milan dataset. Each marker represents a selected POI and the number inside the marker shows the order in which the POIs are visited. The start point is denoted by a marker without a number. Each path between two POIs is represented by a straight line, since the euclidian distance was used to measure the distance between two POIs.

In order to check whether this plan is actually a good plan we provide the data of the POIs in table 8.3. From this table we can see that POIs that are selected all have a rating above 0.7 and that the POIs with later time windows are also visited later. However, POI number 6 shows an impossible situation. While it starts at 18:30, the visit of this restaurant lasts 94 minutes, meaning that the earliest possible time a tourist is supposed to leave this restaurant is at 20:04. Leaving at this time makes visiting Zucca in galleria and Monumento a Vittorio Emanuele impossible. We elaborate on this problem in Section 9.3 of the discussion.

In Figure 8.9 one can find the visual representation of the first found plan by the DTR strategy on the Milan dataset. From this map and Table 8.4 we observe that most of the ratings of the POIs that are selected by the DTR strategy are higher than the ones from the best possible plan. However, the duration of the visits from the best possible plan are much longer, which influences the grade in a positive manner.

In Figure 8.10 the first found plan of the Radius strategy on the Milan dataset can be found. The first thing that catches the eye is the distance between the POIs. In comparison with the DTR and the best found plan, the POIs found by the Radius strategy are much closer to each other.

However, as can be seen in Table 8.5, the ratings are lower. Even though the ratings are high in general, the first and last POI selected have a much lower rating in comparison to the other POIs. Another notable thing is the fact that the last POI has no label, which illustrates the sometimes incomplete information of the LinkedGeoData dataset.

As with the best plan found on the Milan dataset, this plan also suffers from the fact that it is impossible to execute. Again, Ristorante Savini is set at the sixth position, which makes it impossible to visit Zucca in galleria and the last POI.

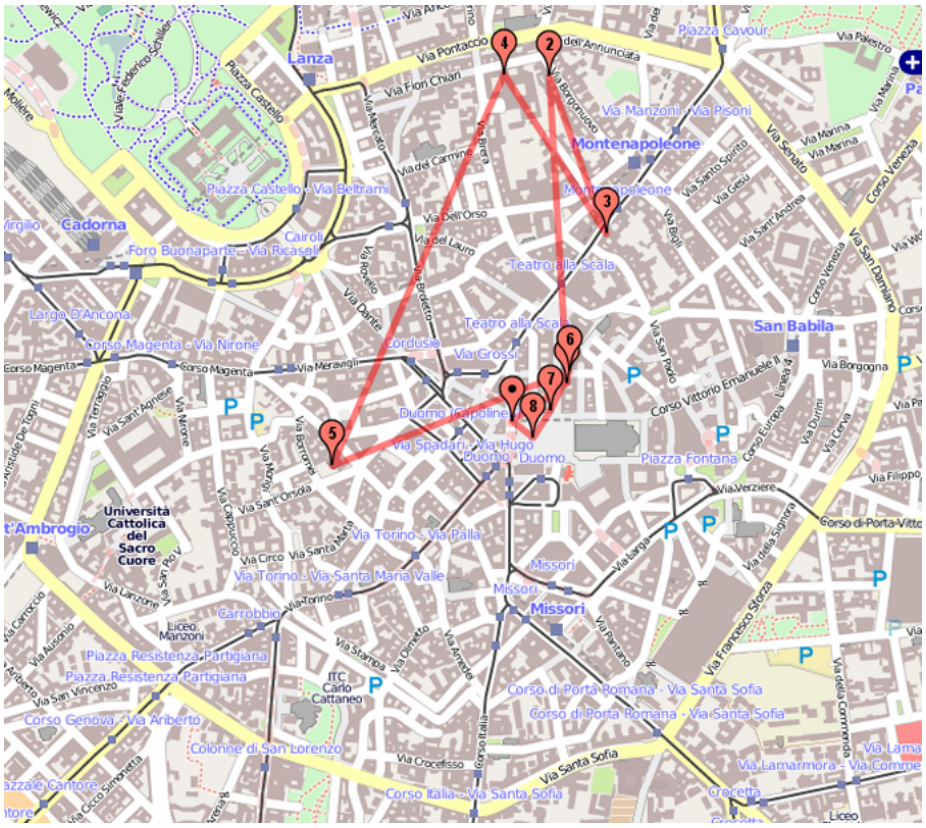


Figure 8.8: The map representing the best plan that can be found on the dataset of Milan, where each pointer represents a POI that is visited and the numbers in the marker represent the order in which the POIs are visited.

Order	Name	Rating	Time Window	Duration
1	Beit Shlomo	0.887	10:00 - 17:00	33
2	Museo del Risorgimento	0.712	10:00 - 16:00	92
3	Museo Poldi Pezzoli	0.764	10:00 - 16:00	90
4	Pinacoteca di Brera	0.801	10:00 - 16:00	100
5	S. Maria del Podone	0.749	10:00 - 17:00	39
6	Ristorante Savini	0.833	18:30 - 22:00	94
7	Zucca in Galleria	0.892	7:00 - 20:00	24
8	Monum. Vitt. Emanuele	0.874	8:00 - 20:00	22

Table 8.3: POI information for the best Milan plan.

Figure 8.9: The map representing the first plan found by the DTR strategy on the dataset of Milan, where each pointer represents a POI that is visited and the number in the marker represent the order in which the POIs are visited.

Order	Name	Rating	Time Window	Duration
1	Sans Egal	0.934	7:00 - 20:00	20
2	S. Maria del Podone	0.749	10:00 - 17:00	39
3	Monum. Vitt. Emanuele	0.874	8:00 - 20:00	22
4	Café Victoria	0.863	7:00 - 20:00	39
5	S. Maria del Podone	0.749	10:00 - 17:00	39
6	Zucca in Galleria	0.892	7:00 - 20:00	24
7	Beit Shlomo	0.887	10:00 - 17:00	33
8	Ristorante Savini	0.833	18:30 - 22:00	94

Table 8.4: POI information for the first found DTR plan on Milan.

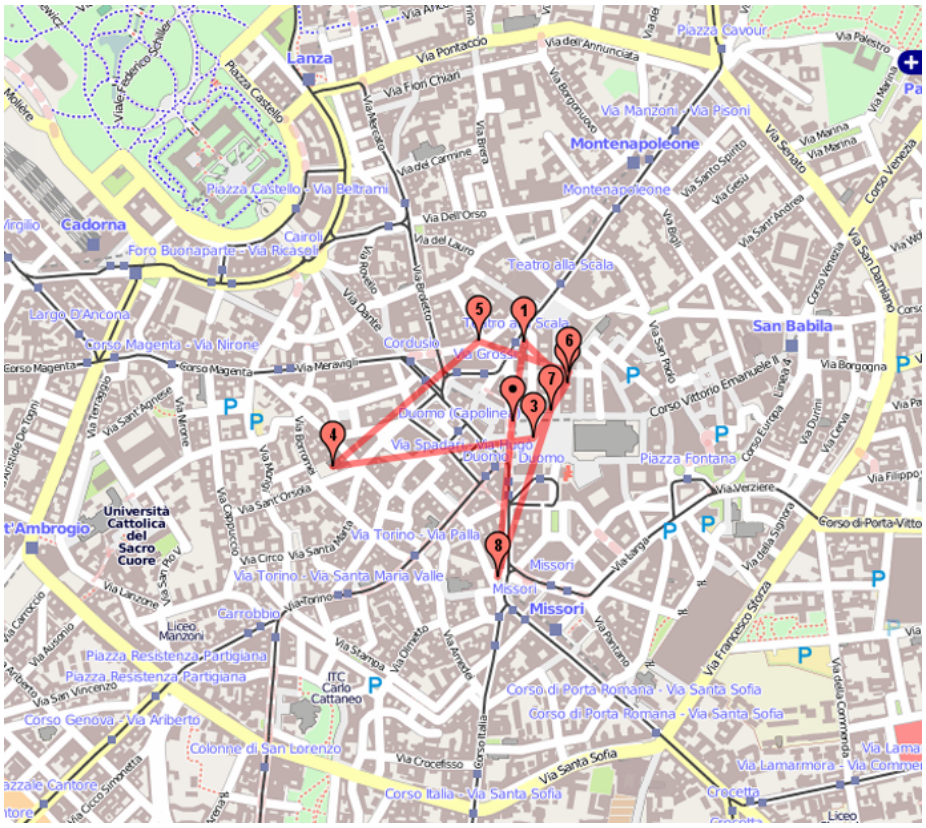


Figure 8.10: The map representing the first plan found by the Radius strategy on the dataset of Milan, where each pointer represents a POI that is visited and the numbers in the marker represent the order in which the POIs are visited.

Order	Name	Rating	Time Window	Duration
1	Grom	0.698	10:00 - 20:00	20
2	Beit Shlomo	0.887	10:00 - 17:00	33
3	Monum. Vitt. Emanuele	0.874	8:00 - 20:00	22
4	S. Maria del Podone	0.749	10:00 - 17:00	39
5	Café Victoria	0.863	7:00 - 20:00	39
6	Ristorante Savini	0.833	18:30 - 22:00	94
7	Zucca in Galleria	0.892	7:00 - 20:00	24
8	-	0.625	8:00 - 20:00	19

Table 8.5: POI information for the first found Radius plan on Milan.

City trip plans Amsterdam

In Figure 8.11 one can find the visual representation of the first found plan by the DTR strategy on the dataset of Amsterdam. From this map and Table 8.6 we observe that most of the ratings of the POIs that are selected by the DTR strategy are above 0.9. When we look at the route, the tourist is send all the way through the city and back. However, this makes sense when looking at the time windows, since the restaurant Nam Kee cannot be visited before 18:30 and the last two POIs cannot be visited before 21:00. Another thing that catches the eye is the length of the plan. The plan only contains 7 POIs, which can happen due to the variable upper bound of the DTR strategy on the Amsterdam dataset.

In Figure 8.12 one can find the visual representation of the first found plan by the Radius strategy on the Amsterdam dataset. Again we can see that the POIs are much closer to each other than in the plan resulting from the DTR strategy. However, from Table 8.7 we can see that also in this case, the ratings of the plan resulting from the Radius strategy are a little lower and the duration of the POIs are also a little shorter than is the case with the DTR strategy.

Figure 8.11: The map representing the first plan found by the DTR strategy on the dataset of Amsterdam, where each pointer represents a POI that is visited and the numbers in the marker represent the order in which the POIs are visited.

Order	Name	Rating	Time Window	Duration
1	Theo Thijssen monument	0.953	8:00 - 20:00	21
2	Noorderkerk	0.862	8:00 - 20:00	40
3	Portugees-Israelitische gem.	0.861	10:00 - 17:00	34
4	Nam Kee	0.749	18:30 - 22:00	95
5	Nationaal Monument	0.988	8:00 - 22:00	22
6	Wynand Fockink	0.993	21:00 - 23:59	95
7	De Vriendschap	0.892	21:00 -23:59 00	92

Table 8.6: POI information for the first found DTR plan on Amsterdam.

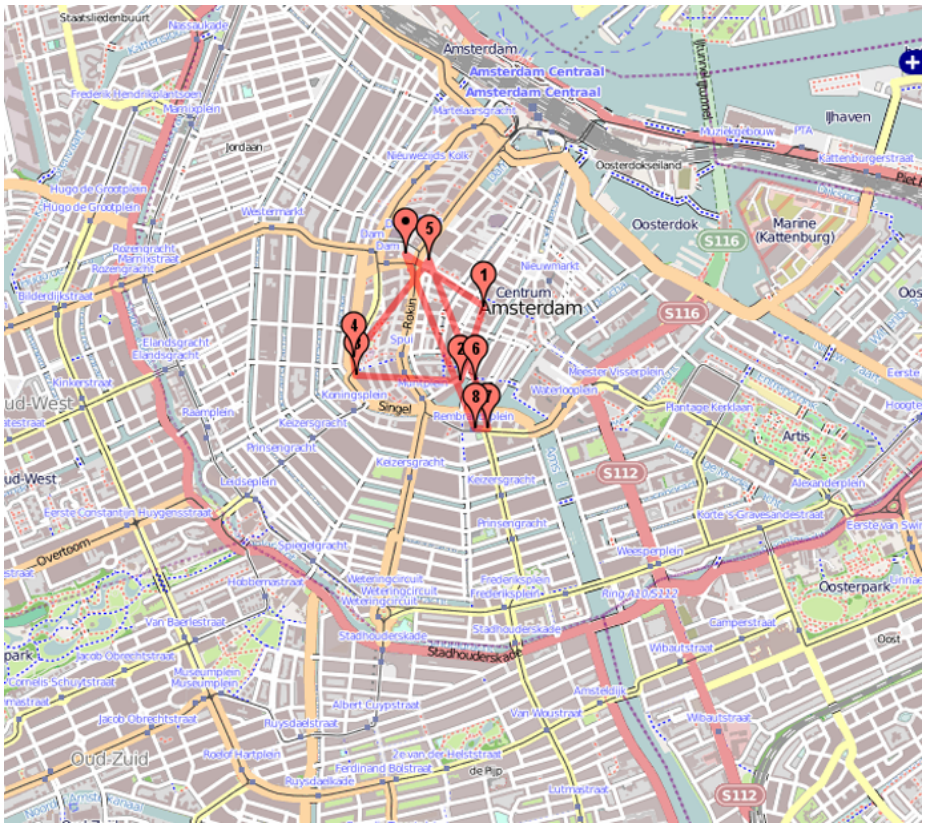


Figure 8.12: The map representing the first plan found by the Radius strategy on the dataset of Amsterdam, where each pointer represents a POI that is visited and the numbers in the marker represent the order in which the POIs are visited.

Order	Name	Rating	Time Window	Duration
1	Waalse kerk	0.759	10:00 - 17:00	36
2	Coffee Company	0.873	7:00 - 20:00	25
3	Oude Lutherse Kerk	0.757	10:00 - 17:00	39
4	Kantjil ed Tijger	0.941	18:30 - 22:00	94
5	Nationaal Monument	0.988	8:00 - 22:00	22
6	Staalmeersters	0.946	21:00 - 23:59	100
7	Cooldown Café	0.925	23:00 - 23:59	34
8	Studio 80	0.859	23:00 - 23:59	39

Table 8.7: POI information for the first found Radius plan on Amsterdam.

Chapter 9

Discussion & Future Work

This chapter discusses the results of the experiments. In Section 9.1, the results of the speed experiment are discussed. We continue with the results of the quality experiment in Section 9.2. Since some plans that were constructed during the experiments violated time constraints, we discuss the quality of the TSPTW solver in Section 9.3. The whole process of developing the city trip planner has been done keeping the theoretical aspects of the Semantic Web in mind. We discuss conformity to the Semantic Web in Section 9.4. We conclude this chapter with Section 9.5, containing possible improvements and additions.

9.1 Execution Time Experiment

The success criterion considering time was stated in Section 8.1.2 as: *The developed strategies should be able to generate 10 city trip plans in less than a minute.* As can be seen in Figure 8.1, both strategies clearly pass this criterion, none of the 40 executions of the city trip planning process in LarKC surpasses the fifteen seconds.

As discussed in the Section 6, the Amsterdam dataset comprises many more points of interest than the Milan dataset. The POI data has to be loaded in the LarKC datalayer and significant parts of it have to be retrieved by the plug-ins. At some points, the strategies check every point of interest available in the datalayer. For example, the DTR strategy loads all of them in lists. For this reason, it takes more time when there are more POIs in a dataset. As a result of this, a significant faster execution of the application was achieved on the Milan dataset, than on the larger Amsterdam dataset.

Overall, the application including the Radius strategy is significantly faster than the one with the DTR strategy. Further analysis of the results points to an interaction effect on the strategies when the datasets are alternated. Using the DTR strategy seems to result in a higher increase of execution time when we switch from the Milan to the Amsterdam dataset than when using the Radius strategy for point selection. Although it did not prove significant, there could be

a reason for this observation. The DTR strategy has a method for dynamically adjusting the amount of points included in a set, which could lead to surpassing the upperbound. Including more points in the $\mathcal{T}$-set will result in a TSPTW problem which is harder to solve and thus will take more time.

The standard deviation of the experiment using the DTR strategy on the Amsterdam dataset is larger than the standard deviations of the three other experiments. The DTR strategy produces the exact same $\mathcal{T}$-sets each execution, so the probable cause of this increase of the standard deviation is the TSPTW solver. The used TSPTW solver implementation has random elements, it starts with a random plan and keeps improving this. Although most of the time this results in the optimal route, the amount of passed time to get to this solution may differ due to the randomness, especially when the $\mathcal{T}$-sets contain more points.

9.2 Quality Experiments

The success criterion concerning the quality of the city trip plans was stated in Section 8.1.2 as: *The 10 city trip plans should be of good quality, meaning that the results should be closer to the optimal situation than to the worst situation.* This success criterion can only be evaluated on the Milan dataset. It is not possible to run a complete baseline on the Amsterdam dataset because of the time it will take to complete. Visual inspection of The Cumulated Gain graphs show that both strategies are closer to the optimal curve than the worst curve, so the quality criterion is met by both strategies.

Another observation made in Section 8.2 concerns the performance of the city trip planner given a certain selection strategy. On the Milan dataset, the DTR strategy is outperformed by the Radius selection strategy, while on the Amsterdam dataset it is the other way round. On the Milan dataset we prove that the performance of the Radius strategy was significantly better, where on the Amsterdam dataset there is no way of proving whether the difference is significant, due to a lacking baseline.

As observed in Section 8.2.2, the grades obtained by the strategies are very close to each other. Nevertheless, an in-depth analysis of the resulting plans in Section 8.2.3 show some differences. The POIs of the best plans of the Baseline have, on average, a lower rating than the POIs selected by the two strategies, but the time spent at those POIs is higher. As stated in Section 3.3, the service time is taken into account by the grading function. Plans containing POIs with higher service times will obtain higher overall grades than plans containing POIs with lower service times, while they might contain the same amount of POIs with similar ratings. For this reason, selecting POIs with longer service times will lead to a higher overall grade.

While the Radius strategy did not take the notion of service times into account, the DTR strategy did, but indirectly. As described in Section 5.2, the DTR strategy adjusts its upper bound according to the sum of the service times of all POIs currently in the selected set, in combination with an estimation

of the the time needed for adding another POI. By doing this, the strategy does take notion of service times, which can be seen reflected in practice by the Amsterdam plan which contains only 7 POIs instead of the normal amount of 8. At the Milan dataset the strategy is still limited to selecting at most 8 points, otherwise it would not be possible to compare the strategy with the results of the Baseline.

Both strategies select points based on rating and distance. Besides the service time, temporal aspects are disregarded. This will become a problem when multiple points of interest have small time windows. Although the selected points might be close to each other, it is not possible to visit them in the optimal sequence due to the time restrictions imposed by the specified time windows. As can be seen in Section 8.2.3, for this reason some of the routes take rather strange detours. As a result of not considering the time windows, sometimes the travel distance will increase.

Overall the quality of the plans is very good. The routes makes sense when considering the time windows, but some of the plans are invalid. This is due to the quality of the TSPTW solver, on which we will elaborate in the next section.

9.3 Quality TSPTW Solver

The only plug-in not solely designed and implemented by ourselves is the TSPTW Reasoner, which utilizes the code of a Traveling Salesman Problem with Time Windows solver, provided by Rodrigo Ferreira da Silva and Sebastian Urrutia. This C++ solver is very fast, it enables us to generate many city trip plans in a short period of time. However, over time, some irregularities concerning the quality of the solutions occurred.

The solver does not guarantee to find the optimal solution, but the article describing the solver made clear that the overall solutions where of good quality [17]. When one runs the LarKC implementation multiple times, the grades of the plans will not be constant, although the point selection strategies produce the same sets over and over again. Each run, the grades obtained by the plans are slightly different, with variations in the range of [0 - 0.03]. Still, the high ranking plans will consequently be high ranking, but it does indicate variability in the outputted solutions by the TSPTW solver.

After close inspection of multiple text files containing the results, this appears to be indeed the case. Although in many cases the results do differ, similar sequences of POIs can still be distinguished. The occurrence of variation in the output is not so strange considering the random elements of the TSPTW solver. As mentioned in Section 2.3.2, the TSPTW starts with a random generated solution and uses heuristics to improve the solution.

A more severe problem came to light while describing the best plans in Section 8.2.3. Every POI has a time window and a service time. The test files used to evaluate the solver in [17] all had a service time of 0. Before using the implementation we manipulated these values and the results changed, leading to

the conclusion that service times were considered by the solver. While analyzing the resulting plans, some of them appeared to be invalid, because a tourist would not be able to reach the next POI in time, considering the service time of the current POI. This points to incorrect handling of service times by the solver.

Although we are Rodrigo Ferreira da Silva and Sebastian Urrutia very grateful for sending us the source code of their TSPTW solver, we believe that their implementation could be improved. The period of time needed to find a solution is short, but the solutions are of varying quality. The use of the LarKC platform for our application is a good choice. In case a more reliable solver is found, it can easily be switched with the current one, through workflow manipulation.

9.4 Conformation to The Semantic Web

The application is developed while keeping the theoretical aspects of the Semantic Web into mind. One of these aspects is the open world assumption. For practical purposes we, however, choose to consider two POI classes to be disjoint. When at a certain point the dataset contains points with two different classes, these are two different POIs. The Rijksmuseum is, for example, a museum in Amsterdam, which is located inside a historic building. This will result in a historicBuilding POI and a museum POI. When a POI could have had multiple classes, this would have created problems concerning the time related concepts, the rating and deciding whether to include the POI in the plan P.

The time and rating will be different depending on the class of a POI. You can walk around the Rijksmuseum all day (a = 7:00, b = 22:00) but the museum inside will have a different time window (a = 9:00, b = 18:00). It also takes less time to admire the building ($d = 15$ *minutes*), than visiting the museum ($d = 120$ *minutes*). The rating might also be different, one aspect of the point could be more appreciated than the other.

During the point selection process, problems would occur if classes were not disjoint. If a tourist indicates, using the $\mathcal{NUC}$, that one class is not preferred and the tourist would still like to visit other things (which is hopefully the case), this could lead to a contradiction. For example, a tourist expressing the preference to not visit museums and the preference to visit historic buildings, would lead to a contradiction in case of the earlier Rijksmuseum example. In the future, a shift from the POI concept to the notion of an activity, could solve this problem. We could then state that multiple activities can take place at the POI Rijksmuseum.

9.5 Future Work

In this section we present several points on which improvements can be made alongside possible additions and extensions to the to the currently implemented city trip planner.

As discussed in Section 9.3, the results generated by the **TSPTW solver** are

of varying quality. Using a more stable, or even a TSPTW algorithm guaranteeing the generation of the optimal solution, would greatly improve the stability of the results obtained by the city trip planning process. This could be done by modifying the current solver, developing our own TSPTW solver or finding a new solver.

A point of possible improvement is the **POI information** provision. Currently, the city trip planner provides basic information for all the POIs in the resulting plan such as name, location, time related information and rating. We could enhance the experience of using our application by providing additional information such as pictures or a Wikipedia description. This could be achieved by developing a separate LarKC plug-in that obtains the information, given the URIs of the POIs in the selected set.

Besides enriching the information of the POIs, the amount of considered POIs could be increased by **incorporating additional data sources**. As described in Section 6.1, we use LinkedGeoData as our only data source. In Section 6.4 we analyzed the data and concluded that the LinkedGeoData dataset was far from complete. In the case of Milan, very important POIs were missing. In Section 6.1, we also discussed the complications of incorporating multiple data sources. Still, the data of the application needs improving in order to provide our users with the best plans possible and the best way of doing this is by incorporating multiple data sources. Examples of data sources which could be a valuable addition are DBPedia, FourSquare and Booking.com.

Furthermore, a major point that can be improved is the procedure which is used to **assign ratings to POIs**. Currently the ratings of a POIs are entered manually, based on looking them up on websites such as TripAdvisor and Booking.com. By automatically determining these grades, we can save ourselves lots of manual labor. Another advantage of using an automated grading system, is that we can incorporate as many sources as we want. Also, ratings of POIs are subject to change. For example, the quality of restaurant can deteriorate or the current collection of a museum can be of less quality than a previous one. Automatic rating determination can handle these changes by, for example, modifying the ratings of POIs once in a while, by taking new reviews into account.

Even though the **selection strategies** performed well and confirmed the hypothesis, they can be improved at some points. As stated in Section 9.2, both strategies do not take all the temporal aspects of POIs into account. The DTR strategy selects POIs until it is unlikely that adding another POI will result in a solvable set, due to the sum of service times of the already selected POIs. The Radius strategy could be extended with such a construction. Also, the consideration of time windows through preferring POIs with partially overlapping time windows could prevent undesired detours.

Major headway can also be made in the area of **routing**. The current version of the city trip planner does routing based on euclidean distance, meaning that it draws straight lines between POIs. If we were to include actual routing algorithms, this would be a major improvement in the usability of the application. One way of doing this, is by using the Traveling Salemsan Ap-

plication[1], which contains a Java library that accomplishes routing based on OpenStreetMap data.

Nowadays, many tourists are in the possession of a smartphone. By creating a **mobile version** of the city trip planner, a much larger audience can be addressed. The usability would greatly improve, rerouting could take place on the fly and tourists could use GPS to determine their position relative to the POIs. The scenario sketched in Section 3.1 already includes the notion of a mobile application and we believe our application is suitable for such a transformation.

Even though the computation performed by the selection strategies is intensive, smartphones should be able to handle this. The processors of the current smartphones are fast and with normal use of the application, they will have to generate no more than 10 plans. This is reached by our dual core setup within 15 seconds and should not take more than a minute using a state of the art smartphone containing a CPU of around 1GHz.

As described in Section 2.5, many applications developed for assisting tourists, require an Internet connection to function. This will not hold for our application. Our datasets can easily be stored on the mobile device, since the files are really small: the Milan dataset is 147Kb and the Amsterdam dataset is 1.5Mb.

In the future, a modified version of the application could also be applied in **other areas** than city trip planning or even tourism. For example, the system could be applied in museums. Here, the displayed objects would serve as POIs. In this case $\mathcal{UI}$ contains art objects a user definitely wants to visit. $\mathcal{UC}$ contains the kind of art objects a user certainly wants to visit. One can think of renaissance paintings as an example. $\mathcal{NUC}$ would contain the kind of art objects the user does not want to visit, for example, sculptures. The result would be the order in which to visit the art objects along with a route.

Another example is disaster management. In a disaster scenario, the people in need of help would be the POIs and the result the order in which to visit these people to give them help or food. In this case priorities can be given to certain people or areas by putting them in $\mathcal{UI}$. $\mathcal{UC}$ and $\mathcal{NUC}$ would be less useful, or even discriminating.

[1] `http://wiki.openstreetmap.org/wiki/Traveling_Salesman`

Chapter 10

Conclusion

This master thesis revolves around the process of planning a city trip. In the introduction three research questions were formulated. The first research question was stated as: *How can we effectively select a set of points of interest which will result in a good city trip plan?* Two point selection strategies were developed in order to select appropriate POIs for tourists from a data set: the Distance Times Rating strategy (DTR) and the Radius strategy. The DTR strategy utilizes ordered lists that are constantly updated while the Radius strategy selects highly rated POIs within a circle with a continuously expanding radius.

After multiple sets of points were selected, a TSPTW solver was used to find the shortest route visiting all the selected POIs in a set while considering time windows and service times. The resulting city trip plans were evaluated and assigned a grade, which enabled the ranking of the plans. The best plans were presented to the user on a map, displaying the sequence of the to be visited points.

The functionalities mentioned above were implemented by the following LarKC plug-ins: the DTR Point Selecter, the Radius Point Selecter, the TSPTW Reasoner, the Rank Decider and the Cartographer. A Planning Decider was used to conduct and control the overall process.

The second research question considers the data needed by the application to function correctly: *Which datasets do we use for the extraction of points of interest and how to structure this data?* We chose to use one dataset, because using multiple datasets would most probably entail complex instance matching. The utilized Semantic Web data originates from the LinkedGeoData project, which is an abstraction of the OpenStreetMap project. A separate dataset was created for points in Milan and points in Amsterdam. A taxonomy of classes was developed to structure the data.

The third and final research questions is concerned with the manner of evaluating the city trip plannings: *How to evaluate the quality of a city trip plan?* A grading function was introduced, which considers the ratings of the selected POIs, the time spent at the POIs, whether the POIs match the user preferences and the amount of time needed for travelling between the POIs.

In order to confirm or reject the hypothesis *"The LarKC platform is able to generate 10 city trip plans of good quality in less than a minute, using strategies for selecting points of interest from the web of data combined with a TSPTW solver"*, we had to evaluate the quality of the city trip plans and the speed of the application. To accomplish this, the LarKC application was used to test the execution time, while a separate multithreaded Test Environment was used to conduct the experiments evaluating the quality of the city trip plans.

The Test Environement was used to create a baseline: all the solutions of a certain scenario on the Milan data set. The two strategies were compared to the sorted baseline using the Discounted Cumulated Gain method. This method is based on two principles: highly rated plannings are more valuable than low rated plannings and the greater the ranked position of a relevant plan, the less valuable it is for the user, because it is less likely the user will ever look at the plan.

On the Milan data set the Discounted Cumulated Gain results of the Radius strategy were significantly better than the results of the DTR strategy. Both of them were closer to the optimal results than the worst possible results. We were unable to run a baseline on the Amsterdam data set, due to the amount of time needed to finish it. Lacking the baseline, no significance measures could be applied, although on the Amsterdam data set the DTR strategy appeared to be performing better than the Radius strategy.

Experiments to measure the execution time were conducted using both the Amsterdam and Milan data sets. City trip plans generated using the two developed strategies were available within fifteen seconds, although the Radius strategy was faster than the DTR strategy. The Amsterdam data set contains more points than the Milan data set. Therefore, it takes significantly more time to create city trip plans for Amsterdam.

Based on the results from these experiments we can confirm the hypothesis. The city trip plans are of sufficient quality, they are closer to the optimal result than the worst result and the execution times stay below fifteen seconds. Although these results are very promising, the TSPTW solver appeared not as reliable as we would like it to be, creating results of varying quality and some invalid solutions. This is a component of the application which could be replaced by a more stable plug-in. Additional future work can concern other areas of implementation, improvements of the selection strategies, adding more data sources and making the application available on mobile platforms.

Bibliography

[1] D. Allemang and J. Hendler. *Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[2] G. Antoniou and F. van Harmelen. *A Semantic Web Primer*. The MIT Press, 2004.

[3] D.L. Applegate, R.E. Bixby, V. Chvátal, and W.J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2007.

[4] G. Ashworth and S.J. Page. Urban tourism research: Recent progress and current paradoxes. *Tourism Management*, 32(1):1 – 15, 2011.

[5] M. Assel, A. Cheptsov, B. Czink, C. Fuchs, N. Lanzanasto, V. Momtchev, S. Kotoulas, L. Bradeško, B. Fortuna, and I. Toma. Final larkc architecture and design, 2011. Deliverable D5.3.3.

[6] M. Assel, A. Cheptsov, G. Gallizo, I. Celino, D. Dell'Aglio, L. Bradeško, M. Witbrock, and E. Della Valle. Large knowledge collider: a service-oriented platform for large-scale semantic reasoning. In *Proceedings of the International Conference on Web Intelligence, Mining and Semantics*, WIMS '11, pages 41:1–41:9, New York, NY, USA, 2011. ACM.

[7] S. Auer, J. Lehmann, and S. Hellmann. Linkedgeodata: Adding a spatial dimension to the web of data. *Lecture Notes in Computer Science*, 5823:731–746, 2009.

[8] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.

[9] D.P. Bhattacharyya. Mediating india: An analysis of a guidebook. *Annals of Tourism Research*, 24(2):371 – 389, 1997.

[10] L. Bigras, M. Gamache, and G. Savard. The time-dependent traveling salesman problem and single machine scheduling problems with sequence dependent setup times. *Discrete Optimization*, 5(4):685 – 699, 2008.

[11] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, 2009.

[12] D. Brickley and R.V. Guha. *RDF Vocabulary Description Language 1.0: RDF Schema*. W3C Recommendation, 2004. Available at `http://www.w3.org/TR/2004/REC-rdf-schema-20040210/`.

[13] B. Brown and M. Chalmers. Tourism and mobile technology. In *Proceedings of the eighth conference on European Conference on Computer Supported Cooperative Work*, pages 335–354, Norwell, MA, USA, 2003. Kluwer Academic Publishers.

[14] R.W. Calvo. A new heuristic for the traveling salesman problem with time windows. *Transportation Science*, 34(1):113–124, 2000.

[15] W.B. Carlton and J.W. Barns. Solving the traveling-salesman problem with time windows using tabu search. *IIE transactions Science*, 28(8), 1996.

[16] K. Cheverst, N. Davies, K. Mitchell, A. Friday, and C. Efstratiou. Developing a context-aware electronic tourist guide: some issues and experiences. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '00, pages 17–24, New York, NY, USA, 2000. ACM.

[17] R.F. da Silva and S. Urrutia. A general vns heuristic for the traveling salesman problem with time windows. *Discrete Optimization*, 7(4):203 – 211, 2010.

[18] M. Dorigo, V. Maniezzo, and A. Colorni. Positive feedback as a search strategy, 1991.

[19] Y. Dumas, J. Desrosiers, E. Gelinas, and M.M. Solomon. An optimal algorithm for the traveling salesman problem with time windows. *Operations Research*, 43(2):pp. 367–371, 1995.

[20] D. Fensel, F. van Harmelen, B. Andersson, P. Brennan, H. Cunningham, E. Della Valle, F. Fischer, Z. Huang, A. Kiryakov, T.K. Lee, L. Schooler, V. Tresp, S. Wesner, M. Witbrock, and N. Zhong. Towards larkc: A platform for web-scale reasoning. In *Proceedings of the 2008 IEEE International Conference on Semantic Computing*, pages 524–529, Washington, DC, USA, 2008. IEEE Computer Society.

[21] F. Focacci, A. Lodi, and M. Milano. A hybrid exact algorithm for the tsptw. *Informs Journal on Computing*, 14:403–417, October 2002.

[22] D. Fogel. An evolutionary approach to the traveling salesman problem. *Biological Cybernetics*, 60:139–144, 1988. 10.1007/BF00202901.

[23] K.R. Fox. *Production scheduling on parallel lines with dependencies*. PhD thesis, John Hopkins University, 1973.

[24] M. Gendreau, A. Hertz, G. Laporte, and M. Stan. A generalized insertion heuristic for the traveling salesman problem with time windows. *Operations Research*, 46(3):pp. 330–335, 1998.

[25] L. Gouveia and S. Voss. A classification of formulations for the (time-dependent) traveling salesman problem. *European Journal of Operational Research*, 83(1):69 – 82, 1995.

[26] A. Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14:47–57, June 1984.

[27] K. Hagen, R. Kramer, M. Hermkes, B. Schumann, and P. Mueller. Semantic matching and heuristic search for a dynamic tour guide. In Andrew J. Frew, editor, *Information and Communication Technologies in Tourism 2005*, pages 149–159. Springer Vienna, 2005.

[28] M. Haklay and P. Weber. Openstreetmap: User-generated street maps. *IEEE Pervasive Computing*, 7:12–18, October 2008.

[29] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20:422–446, October 2002.

[30] M. Kenteris, D. Gavalas, and D. Economou. An innovative mobile electronic tourist guide application. *Personal and Ubiquitous Computing*, 13:103–118, 2009.

[31] T. Kindberg, M. Chalmers, and E. Paulos. Guest editors' introduction: Urban computing. *IEEE Pervasive Computing*, 6:18–20, 2007.

[32] T. Kinoshita, M. Nagata, Y. Murata, N. Shibata, K. Yasumoto, and M. Ito. A personal navigation system for sightseeing across multiple days. In *Proc. of 3rd International Congress on Mobile Computing and Ubiquitous Networking (ICMU2006)*, pages 254–259, 2006.

[33] A. Langevin, M. Desrochers, J. Desrosiers, S. Gélinas, and F. Soumis. A two-commodity flow formulation for the traveling salesman and the makespan problems with time windows. *Networks*, 23(7):631–640, 1993.

[34] C. Lee, Y. Chang, and M. Wang. Ontological recommendation multi-agent for tainan city travel. *Expert Systems with Applications*, 36(3, Part 2):6740 – 6753, 2009.

[35] A.A. Lew. A framework of tourist attraction research. *Annals of Tourism Research*, 14(4):553 – 575, 1987.

[36] A. Maruyama, N. Shibata, Y. Murata, K. Yasumoto, and M. Ito. A personal navigation system for sightseeing across multiple days. In *Proc. of 11th World Congress on ITS*, pages 18–21, 2004.

[37] J. Miguens, R. Baggio, and C. Costa. Social media and tourism destinations: Tripadvisor case study. *Advances in Tourism Research*, 2008.

[38] M. Nagata, Y. Murata, N. Shibata, K. Yasumoto, and M. Ito. A method to plan group tours with joining and forking. In Tzai-Der Wang, Xiaodong Li, Shu-Heng Chen, Xufa Wang, Hussein Abbass, Hitoshi Iba, Guo-Liang Chen, and Xin Yao, editors, *Simulated Evolution and Learning*, volume 4247 of *Lecture Notes in Computer Science*, pages 881–888. Springer Berlin / Heidelberg, 2006.

[39] J.W. Ohlmann and B.W. Thomas. A compressed-annealing heuristic for the traveling salesman problem with time windows. *Informs Journal on Computing*, 19(1):80–90, 2007.

[40] W3C OWL Working Group. *OWL 2 Web Ontology Language: Document Overview*. W3C Recommendation, 2009. Available at `http://www.w3.org/TR/owl2-overview/`.

[41] G. Pesant, M. Gendreau, J. Potvin, and J. Rousseau. An exact constraint logic programming algorithm for the traveling salesman problem with time windows. *Transportation Science*, 32(1):12–29, 1998.

[42] J. Picard and M. Queyranne. The time-dependent traveling salesman problem and its application to the tardiness problem in one-machine scheduling. *Operations Research*, 26(1):pp. 86–110, 1978.

[43] S. Schmitz, A. Zipf, and Neis P. New applications based on collaborative geodata - the case of routing. In *XXVIII INCA International Congress on Collaborative Mapping and SpaceTechnology*, Gandhinagar, Gujarat, India, 2008.

[44] N. Shadbolt, W. Hall, and T. Berners-Lee. The semantic web revisited. *Intelligent Systems, IEEE*, 21(3):96 –101, jan.-feb. 2006.

[45] W. Souffriau and P. Vansteenwegen. Tourist trip planning functionalities: State of the art and future. In Florian Daniel and Federico Facca, editors, *Current Trends in Web Engineering*, volume 6385 of *Lecture Notes in Computer Science*, pages 474–485. Springer Berlin / Heidelberg, 2010.

[46] C. Stadler, J. Lehmann, K. Höffner, and S. Auer. Linkedgeodata: A core for a web of spatial open data. Submitted to the Semantic Web Journal, 2011.

[47] E. Della Valle, I. Celino, and D. Dell'Aglio. The experience of realizing a semantic web urban computing application. In *Proceedings of the Terra Cognita Workshop 2009, colocated with ISWC 2009 - the 8th International Semantic Web Conference, 25-29 October 2009 - Washington, DC*, 2009.

[48] E. Della Valle, I. Celino, and D. Dell'Aglio. The experience of realizing a semantic web urban computing application. *Transactions in GIS*, 14(2):163–181, 2010.

[49] P. Vansteenwegen and D. Van Oudheusden. The mobile tourist guide: An or opportunity. *OR Insight*, 20(3):21–27, 2007.

[50] P. Vansteenwegen, W.r Souffriau, G. Vanden Berghe, and D. Van Oudheusden. Iterated local search for the team orienteering problem with time windows. *Computers & Operations Research*, 36(12):3281 – 3290, 2009. New developments on hub location.

[51] B Wu, Y. Murata, N. Shibata, K. Yasumoto, and M. Ito. A method for composing tour schedules adaptive to weather change. In *Intelligent Vehicles Symposium, 2009 IEEE*, pages 1407 –1412, june 2009.

Glossary

$\mathcal{S}$-set
: The set of all POIs contained in the problem space.

$\mathcal{UC}$-set
: Set containing the class preferences, represented by pairs of C_i and X_i, where X_i is the amount of times a tourist likes to visit POIs categorized with C_i.

$\mathcal{UI}$-set
: This set is used to give the user the option to indicate specific POIs that he or she will definitely visit.

$\mathcal{NUC}$-set
: With the help of this set a user can specify which classes he or she definitely would not like to visit.

$\mathcal{T}$-set
: A set of selected points for which the TSPTW solver will estimate a short route visiting all the nodes in the given set.

Class C
: A class is a set of elements which some POIs have in common, each POI is an instance of a class.

Discounted Cumulated Gain (DCG)
: An evaluation method originating from the field of information retrieval, is developed in order to credit information retrieval methods for their ability to retrieve highly relevant documents.

Large Knowledge Collider (LarKC)
: A platform for massive distributed incomplete reasoning that will remove the scalability barriers of currently existing reasoning systems for the Semantic Web.

LarKC plug-in
: The building blocks of a LarKC application, can be reused or created from scratch.

LarKC workflow
Workflows are used within LarKC to create a sequence of plug-ins.

LinkedGeoData (LGD)
The LinkedGeoData project aims to be the geographical counterpart of DBPedia, providing a dataset consisting of geographical data in the RDF format, abstracted from the OSM project.

Normalized Discounted Cumulated Gain (nDCG)
An evaluation method originating from the field of information retrieval that shows the performance of an information retrieval method relative to the ideal situation, is calculated by dividing the DCG value of a strategy with the DCG of the ideal situation.

Ontology
A common conceptualization, ontologies define the data and relations between concepts.

OpenStreetMap (OSM)
A community effort, aiming to create a set of map data which is free to use.

Plan P
A plan for a city trip, represented by a list indicating the optimal sequence of visiting the selected POIs.

Point Of Interest (POI)
Geographical points which might be of interest to a tourist.

RDF Schema (RDFS)
RDF Schema extends RDF and provides a minimal ontology representation language.

Resource Description Framework (RDF)
The Resource Description Framework is used to make statements about resources.

Semantic Web
The Semantic Web extends the World Wide Webs infrastructure with techniques making represented information not only readable for humans, but also interpretable and operable for machines.

Service Time
The time in minutes a tourist will probably spend at a sight.

SPARQL Protocol and RDF Query Language
Query language used to retrieve information from RDF data sources.

Time Dependent Traveling Salesman Problem (TDTSP)
: The problem of finding the shortest route between a set of points, while the travel costs between the points are dependent on their position in the tour.

Time Window
: Sets the period of time in which a POI is relevant to a user. Time windows can also be applied to a plan, indicating the period of time a tourist wants to spend visiting POIs.

Tourist Trip Design Problem (TTDP)
: The combination of the processes of choosing which POIs to visit and in what order.

Traveling Salesman Problem (TSP)
: The problem of finding the shortest route between a set of points, where each point is visited exactly once.

Traveling Salesman Problem with Time Windows (TSPTW)
: The problem of finding the shortest route between a set of points, while considering time restrictions of the points.

Triple
: A single statement about a resource, consisting of a subject, a predicate and an object.

Universal Resource Identifier (URI)
: Universal Resource Identifiers are used to identify resources.

Appendix A

Taxonomy

Appendix B

Ontology Mapping

Table B.1: Mapping of LinkedGeoData ontology to City Trip Planner ontology

LinkedGeoData class	**City Trip Planner class**
Museum	Museum
TourismMuseum	Museum
Artgallery	Artgallery
Monument	Monument
PlaceOfWorship	Church
HistoricChurch	Church
Castle	Castle
Fort	Fort
HistoricRuins	Ruins
Casino	Casino
Nightclub	Nightclub
Pub	Pub
Park	Park
NaturalBeach	Beach
TourismZoo	Zoo
TourismHotel	Hotel
Hotel	Hotel
Hostel	Hostel
TourismApartments	Apartment
Apartment	Apartment
Restaurant	Restaurant
Cafe	Cafe
Bar	Bar
IceCream	IceCreamShop
FastFood	FastFood

Continued on next page

Table B.1 – continued from previous page

LinkedGeoData class	City Trip Planner class
Food	FoodStore
Bakery	Bakery
Supermarket	Supermarket
Deli	DeliShop
Delicatessen	DeliShop
Cheese	CheeseShop
Antique	AntiqueShop
Antiques	AntiqueShop
Books	Bookshop
Art	ArtShop
LiquorStore	Alcohol
WineryShop	WineryShop
Clothes	ClothesShop
Fashion	ClothesShop
Foto	PhotographyStore
Photo	PhotographyStore
Souvenir	SouvenirShop
Souvenirs	SouvenirShop
Gift	GiftShop
Gifts	GiftShop
Jewelry	JewelleryShop
Kiosk	Kiosk
Mall	Mall
Market	Market
Music	MusicShop
Perfume	Perfumery
Shoes	ShoeShop
Toys	ToyShop