Study of mobility patterns

Steven Engels

promotor :
Prof. dr. Bart KUIJPERS

Eindverhandeling voorgedragen tot het bekomen van de graad master in de informatica databases

universiteit hasselt

Acknowledgements

This thesis took time, work and dedication, but also the support and patience of those around me. Without them, this thesis would have been difficult to realize.

I wish to thank my parent, family and friends, for their encouragement this last year, but also all previous years of my studies. Thank you for believing in me, and being there when I needed you.

Also, I wish to thank Prof. dr. Bart Kuijpers for giving me the opportunity to work on this interesting subject. Your confidence, guidance and suggestions helped me succeed in making this thesis.

Special thanks goes out to Bart Moelans, I really appreciate the time and effort he invested in my thesis. I also wanted to thank dr. Alejandro Vaisman, whose suggestions and remarks were valued and helpful.

Abstract

Recent technologic developments in mobile location aware devices allow the movement of objects to be captured and stored in databases. The type of data describing this movement is called a trajectory.

A trajectory can be observed from a geometric point of view, a curve through space, but also from a semantic point of view, by using background information to interpret its course. This allows us to define semantically enriched trajectories [dMR05, SPD+08], which are annotated with background information resulting in a list of stops and moves, stops being locations of importance to understand the meaning of trajectories, and moves describing the transitions between these stops.

Essentially, these semantically enriched trajectories show movement behavior. As any type of behavior shows patterns, it is interesting to find these in movement. One option to analyze trajectories is association analysis, a group of techniques with the purpose of finding rules describing found patterns.

Apriori [AIS93] is one of the most important association analysis tools. This algorithm has already been adapted to process sequential data, a class of data models trajectories belong to, resulting in AprioriAll [AS95].

The goal of this work is to describe an Apriori implementation that is adapted to the specifications of semantically enriched trajectories. Our algorithm focuses on the specific nature of trajectories by defining a special data model, inspired on the Mobility Patterns introduced in [dMR05]. This model will include the use of wildcards in defining frequent sequences as to increase its flexibility and semantics.

Nederlandstalige samenvatting

Inleiding

De ontwikkeling van GPS en tracking systemen heeft in combinatie met database technologie de mogelijkheid gecreëerd om bewegingen van objecten te observeren en op te slaan. De bewegingen, beschreven als *trajecten*, worden op zulk een grote schaal vergaard, dat het moeilijk is om deze te analyseren. Er bestaat echter een geheel van technieken dat toegespitst is op het analyseren van en zoeken van patronen in een grote hoeveelheid data, namelijk *data mining*.

Beweging wordt beschreven door middel van trajecten. Deze kunnen vanuit verschillende standpunten worden bekeken, vanuit een geometrisch standpunt, maar ook vanuit een semantisch standpunt, waarop meer gefocust wordt op het motief of gedrag dat de beweging veroorzaakt.

Data mining is een onderdeel van Knowledge Discovery in Databases [HK00, TSK05], een erg uitgebreid geheel van technieken, met als doel grote hoeveelheden data te onderzoeken zodat patronen kunnen worden gevonden. Eén groep technieken dat deel uit maakt van data mining is associatie analyse, dewelke associaties of correlaties zoekt in data.

In deze thesis trachten we een associatie analyse techniek te definiëren die gericht is op het zoeken van patronen in een semantische benadering van trajecten en zich daartoe richt op het specifieke karakter van dit type data.

Mobiliteitsdata

Mobiliteitsdata, bestaande uit trajecten, beschrijven de beweging van objecten. Trajecten worden geformuleerd als een continue functie die voor elk tijdstip coördinaten geeft in de beschouwde dimensies, zoals beschreven in

Definitie 1 en getoond in Figuur 2.1. Wanneer we echter deze trajecten willen beschrijven in een database, is dit model van een continue functie moeilijk bruikbaar.

Daarom wordt een traject meestal voorgesteld als een reeks van *traject samples*, voorgesteld in Definitie 2 en Figuur 2.2. Deze punten zijn van de vorm (x_i, y_i, t_i), dewelke coördinaten en een timestamp bevatten die een momentopname van het traject beschrijven. Op basis van deze traject samples kan met behulp van lineaire interpolatie een benadering worden opgesteld van het traject, zoals Figuur 2.3 weergeeft.

Een traject kan vanuit verschillende standpunten worden beschouwd, vanuit een strikt ruimtelijk standpunt, waarbij enkel de geometrische aspecten van het traject worden in acht genomen, maar ook vanuit een semantisch standpunt, waar de motieven die tot het traject leiden het traject worden beschouwd. Door gebruik te maken van achtergrondinformatie kan een traject in een zekere context worden 'geïnterpreteerd'. Een mogelijk voorbeeld is het traject van een toerist in een stad, waar met behulp van kennis van bezienswaardigheden en andere toeristische locaties het traject niet wordt aanzien als een geheel van coördinaten, maar eerder een opeenvolging van locaties.

Het is deze semantische interpretatie van trajecten waar we op concentreren. Er werden reeds algoritmen beschreven die toelaten een traject bestaande uit traject samples om te zetten naar een semantisch verrijkt model [dMR05, SPD+08]. Dit model bestaat uit een lijst van *stops* en *moves* waarin in volgorde de locaties worden opgesomd. We beschouwen twee technieken gericht op het verrijken van trajecten: SMoT en CB-SMoT.

Het SMoT algoritme [ABK+07], omschreven in Listing 2.1 en gedemonstreerd in Figuur 2.4, tracht een traject te interpreteren met behulp van een *applicatie*, dewelke in essentie een opsomming is van *candidate stops*, locaties die voor deze specifieke toepassing interessant zouden kunnen zijn en toelaten het traject te interpreteren. Elk van deze locaties wordt voorzien van een minimum duur, deze laat toe een onderscheid te maken tussen een 'bezoek' of een 'toevallig voorbij gaan' van een candidate stop. Het resultaat is een lijst van stops en een lijst van moves.

CB-SMoT, voorgesteld in [PBKA08] en beschreven in Listing 2.2, is een gelijkaardig algoritme, met het verschil dat deze de verantwoordelijkheid van

het bepalen van de candidate stops niet meer volledig bij de gebruiker plaatst, maar ook zelf naar potentiële stops op zoek gaat. Dit wordt mogelijk door er vanuit te gaan dat indien een bewegend object een interessante locatie benadert of heeft bereikt, deze zal vertragen, en dat bijgevolg de sample punten dichter bij elkaar zullen liggen. Door clustering toe te passen op deze punten kunnen gebieden met een vertraagde snelheid worden herkend, zoals Figuur 2.5 weergeeft. Deze kunnen dan worden herkend als een gedefinieerde candidate stop, of als een nieuwe 'unknown' stop, waarbij het aan de gebruiker wordt overgelaten om te beslissen of deze 'unknown' stop al dan niet als een echte stop beschouwd moet worden.

In het verdere verloop van de thesis, en ook de implementatie, zullen we enkel gebruik maken van SMoT, gezien deze sneller is dan CM-SMoT, en we geen behoefte hebben aan de mogelijkheden van CB-SMoT.

Sequentiële patronen zoeken

Een volgende stap is de notie van semantisch verrijkte trajecten gebruiken als basis voor het zoeken van patronen.

Associatie analyse heeft, vanuit een traditioneel standpunt, het doel *frequente itemsets* te vinden. Door analyse van transacties, dewelke worden omgezet naar itemsets, kunnen we groepen van *items* identificeren die samen regelmatig voorkomen. Het doel van deze analyse en het zoeken van de frequente itemsets, is het formuleren van *associatie regels*, van de vorm

$$A \rightarrow B(sup = x, con = y),$$

die de ontdekte kennis omschrijft. Een regel omschrijft dat een niet-lege itemset A in een transactie, de aanwezigheid van een niet lege itemset B impliceert. Elke regel wordt geassocieerd met een *support sup* en *confidence con*, dewelke respectievelijk uitdrukken hoe interessant en hoe betrouwbaar deze regel is. Het definiëren van een minimum support en confidence laten de gebruiker toe het analyseproces te manipuleren. Minimum support wordt dan ook gebruik om te bepalen of een itemset al dan niet frequent is, gezien deze aangeeft in hoeveel transacties de itemset minimaal aanwezig moet zijn.

Een bekend algoritme dat toelaat frequente itemsets in transacties te zoeken is Apriori [AIS93], waarvan de pseudocode beschreven is in Listing 3.1. Indien men frequente itemsets probeert te vinden, is een naïeve optie alle mogelijke

itemsets te genereren en dan voor elke itemset te bepalen of deze al dan niet frequent is. Dit zou echter betekenen dat tot $2^k - 1$ itemsets moeten worden gegenereerd. Het Apriori algoritme probeert dit echter te vermijden door gebruik te maken van het Apriori Principe, wat toelaat stapsgewijs kandidaat frequente itemsets van grootte k op te bouwen met behulp van de frequente itemsets van grootte $k - 1$. Deze kandidaten worden dan geëvalueerd opdat de infrequente itemsets worden verwijderd, en de frequente als basis kunnen worden gebruikt voor het bepalen van de kandidaten van grootte $k + 1$. Uiteindelijk, wanneer geen kandidaten meer kunnen worden gegenereerd, is de collectie van frequente itemsets bepaald.

De principes die we omschreven hebben voor itemsets, zouden we willen toepassen op onze trajecten. Het datamodel dat we hiervoor gebruiken, het equivalent van de itemset, is de *sequentie*. Een sequentie kan beschouwd worden als set van itemsets waarbij een volgorde op deze itemsets wordt gedefinieerd. Net zoals bij itemsets bestaat er ook een notie van *subsequentie*, waarbij een sequentie in een andere sequentie vervat zit.

We hebben dus aangegeven dat we onze semantisch verrijkte trajecten omzetten naar sequenties, om hierop associatie analyse op toe te passen. Dit omzetten leidt tot een sequentie van singleton-itemsets, waarbij elke itemset een stop bevat. Het doel van deze analyse is het zoeken van *frequente sequenties*, dewelke voldoen aan de door de gebruiker opgegeven minimum support.

Er werden reeds enkele associatie analyse algoritmen ontwikkeld die toegespitst zijn op het zoeken van frequente sequenties, bijvoorbeeld AprioriAll en GSP. AprioriAll, voorgesteld in [AS95] en omschreven in Listing 3.2, bouwt verder op de principes van het voorgestelde Apriori. Ook deze werkt met een stapsgewijze generatie van kandidaat frequente sequenties, waarvan dan wordt gecontroleerd of deze al dan niet frequente zijn. Tussen de kandidaat generatie en de support controle moet er echter een nieuwe stap worden gevoegd, die er voor te zorgen dat het Apriori Principe wordt gerespecteerd. De kandidaat generatie laat namelijk toe dat kandidaten worden gecreëerd die infrequente subsequenties bevatten, wat natuurlijk het Principe tegenspreekt.

Het GSP algoritme [SA96] bouwt verder op het bestaande AprioriAll, maar voegt enkele uitbreidingen toe die de mogelijkheden van de gebruiker

vergroten. Zo definieert GSP 'sliding window', 'time constraints' en 'taxonomies'.

AprioriAll laat reeds toe patronen te vinden, maar de mogelijkheden zijn beperkt, gezien deze niet de volle semantiek van trajecten kunnen bevatten. We zullen een eigen sequentie definiëren, geïspireerd op [dMR05], een werk dat zich concentreert op het formuleren en toepassen van reguliere expressies om sequenties te evalueren. Het introduceren van wildcards, vergelijkbaar met deze gebruik in reguliere expressies, laat toe een zekere flexibiliteit te introduceren in de sequenties, bijvoorbeeld $A.\$.B$ wat aangeeft dat een object van A naar een willekeurige locatie beweegt om van daaruit naar B te bewegen. Gebaseerd op dit type zullen we vervolgens een eigen Stops en Moves Apriori voorstellen.

De eerste stap is het definiëren van een eigen sequentie datamodel, Definitie 10. In dit model beschouwen we een sequentie niet langer als een opeenvolging van itemsets, maar als een opeenvolging van items. Ook beschouwen we een striktere definitie van een subsequentie 11. Deze twee aanpassingen zorgen ervoor dat onze sequenties vergelijkingen vertonen met strings, en dat dit ons toelaat algoritmen te gebruiken die hierop gedefinieerd zijn. Zoals vermeld zullen we ook wildcards introduceren, die toelaten het karakter van semantisch verrijkte trajecten te benadrukken en ook enige flexibiliteit bij het omschrijven van sequenties toe te laten. Een voorbeeld van zulk een traject is

$$Hotel_1.Museum_1.Monument_1.Hotel_1.$$

In eerste instantie maken we bij het beschrijven van Stops and Moves Apriori, waarvan de pseudocode in Listing 3.3 wordt getoond, geen gebruik van wildcards, zodat een vergelijking met AprioriAll mogelijk wordt. Omdat ons datamodel sterk aanleunt met het begrip van strings, kunnen we gebruik maken van bestaande string technieken om ons algoritme te versterken. We volgen het bestaande AprioriAll als een basis voor Stops and Moves Apriori, met als gevolg dat ook hier een stapsgewijze generatie plaats vindt van kandidaat frequente sequenties en frequente sequenties. Een eerste observatie is dat in Stops and Moves Apriori het Apriori Principe niet moet worden opgelegd, gezien dit blijft gelden voor alle genereerde kandidaten. Een tweede observatie is de mogelijkheid om het Boyer-Moore algoritme, beschreven in Listing 3.4 te gebruiken voor het bepalen van substrings, dewelke een vermindering van het aantal berekeningen oplevert.

Het nadeel van deze techniek, is dat het aantal gevonden frequente sequenties vrij beperkt is, gezien de strikte notie van substring die gehandhaafd wordt. Vandaar de introductie van de *enkelvoudige wildcard* \$, zoals beschreven in Definitie 13, dewelke in kandidaat frequente sequenties kan worden opgenomen en semantisch kan worden beschouwd als 'een willekeurige stop'. Het introduceren van deze wildcard laat toe meer flexibiliteit te importeren in de frequente sequenties. Deze flexibiliteit komt echter tegen een bepaalde prijs, we kunnen namelijk Boyer-Moore niet langer gebruiken voor het zoeken van subsequenties, maar moeten inspiratie zoeken bij het evalueren van regulier expressies.

Een volgende stap is het introduceren van een *meervoudige wildcard*, omschreven in Definitie 14, namelijk $\$^+$, dewelke één of meerdere willekeurige items vertegenwoordigt. Het gebruik van deze wildcard laat een nog grotere flexibiliteit toe. Daarnaast kunnen we ook de wildcard $\$^*$ introduceren, om zo aan te geven dat deze kan worden geïnterpreteerd als nul, één of meerdere willekeurige items.

Welke meerwaarden brengen deze technieken met zich mee? Zonder het gebruik van wildcards hebben we een algoritme dat een striktere klasse van resultaten oplevert, maar dat hiervoor een kleiner aantal berekeningen moet uitvoeren. Het gebruik van wildcards laat een grotere flexibiliteit toe, met als gevolg dat alle resultaten die AprioriAll zou vinden ook gevonden worden, en dit aan een vergelijkbaar aantal berekeningen. Het grote voordeel echter is de grotere semantische uitdrukkingskracht van de gevonden frequente sequenties, gezien we met het gebruik van wildcards ook informatie krijgen over het aantal willekeurige items dat zich tussen de andere items bevinden, maar ook dat we nu de mogelijkheid hebben om aan te duiden dat twee items elkaar onmiddellijk opvolgen.

Wanneer alle frequente sequenties werden bepaald, kunnen we deze als uitvoer aannemen, maar het is ook mogelijk om de kennis van deze sequenties in de vorm van regels uit de drukken. Een voorwaarde is natuurlijk dat de uitdrukkingskracht van deze regels de semantiek van onze sequenties volledig weergeeft. Door het definiëren van een predicaat *passes()* dat het direct opvolgen van items weergeeft en een sequentiële conjunctieve connector $\vec{\wedge}$ die de volgorde aangeeft, kunnen regels worden gevormd die de semantiek van een regel uitdrukt.

Implementatie

Om de werking van Stops and Moves Apriori te demonstreren, wordt deze geïmplementeerd, zowel in een versie die niet gebruik maakt van wildcard, als deze met enkelvoudige en deze met meervoudige wildcards.

De data die als invoer zal dienen voor onze algoritmen bestaat uit een collectie van trajecten opgeslagen in de vorm van traject samples. Deze bevatten echter nog geen semantische waarde en zullen bijgevolg door een reeds bestaande implementatie van het SMoT algoritme, in combinatie van een applicatie, worden omgezet naar een lijst van stops. In de implementatie zal deze lijst worden ingelezen en worden omgezet naar een sequentie, het data type waar onze implementatie is op gericht.

Zoals aangegeven, maakt de implementatie gebruik van het principe achter het Boyer-Moore algoritme om subsequenties te vinden in sequenties. De implementatie van Boyer-Moore zak uiteindelijk een aangepaste versie zijn zodat deze de verrijkte trajecten kan verwerken.

Vervolgens wordt Stops and Moves Apriori met enkelvoudige wildcards geïmplementeerd. Deze kan echter geen gebruik meer maken van Boyer-Moore, waarbij dan ook het grootste verschil met de vorige implementatie wordt gegeven. We opteren voor een aangepaste versie van reguliere expressie validatie die dan wel een groter aantal berekeningen nodig heeft.

Ten slotte wordt de implementatie met meervoudige wildcards voorgesteld. Ook hier hebben we vooral aandacht voor het bepalen of een sequentie al dan niet een subsequentie is, gezien dit het grootste verschil beschrijft met de vorige implementatie.

Conclusie en toekomstig onderzoek

De omschrijving en implementatie geven de werking van onze Stops and Moves Apriori aan. We zijn erin geslaagd een algoritme te definiëren dat het karakter van semantisch verrijkte trajecten niet enkel weergeeft, maar ook gebruikt als middel om het aantal berekeningen te verminderen. Onze drie mogelijkheden laten de gebruiker toe om zelf de klasse van resultaten te bepalen.

In de toekomst kunnen we proberen de uitbreidingen voorgesteld in GSP ook te introduceren in Stops and Moves Apriori, indien deze een meerwaarde brengen. Daarnaast zou het ook mogelijk zijn om niet enkel te richten op associatie analyse, maar ook andere types van data mining en deze toe te passen op semantisch verrijkte trajecten.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the development of global positioning systems, tracking systems and digital mapping technology, combined with existing database technology, it has become possible to record and store large amounts of mobility data. However, because of the scale on which this data has been gathered and also its nature, it has become difficult to analyze and study them.

When it comes to analyzing large amounts of data, database technology already has an answer, namely data mining. However, this framework is mainly focused on analyzing traditional data, and not mobility data. Therefore the need has arisen to expand this existing framework in such a way, as to develop and improve tools which allow that mobility data to be processed and analyzed.

This thesis focuses on the combination of both the trajectories and data mining. Before we describe our goal more precisely, an analysis of these two subjects can help define our purpose and motivation.

1.1 Background

1.1.1 Moving Object Data

Essentially, a *trajectory* is data on the movement of an object. It describes the movement of a single object in the real world as a continuous function of time. Given a time instant t, the function returns the position at time t of the object in a d-dimensional space. This can be modeled as a function $o : \mathbf{R} \rightarrow \mathbf{R}^d$.

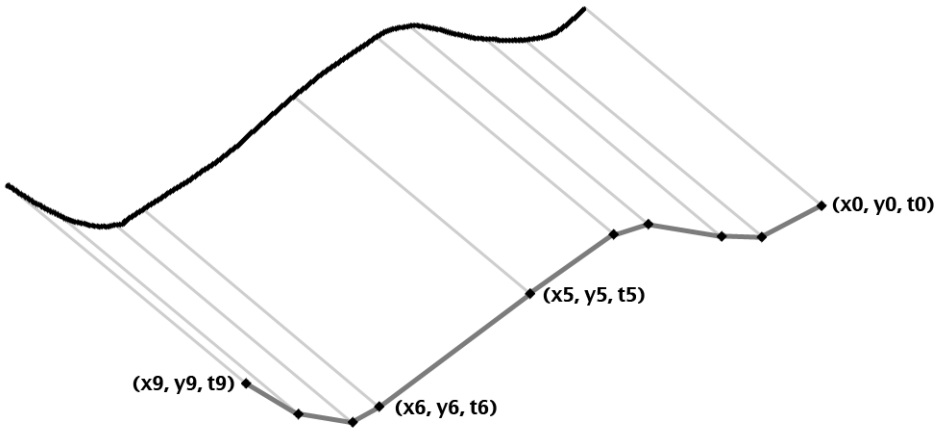

Figure 1.1: Demonstration of how a two-dimensional real-world trajectory can be transformed into trajectory samples.

However, when used in applications, instead of a continuous function, movement is often transformed into a data model where it is represented as a set of observations or control points, with every observation described as a tuple (x, y, t) where combining x and y gives the coordinate, and t the timestamp at which instant in time the coordinate was recorded. As a consequence of this model, we do not have the exact coordinate for every instant of time. However, we are able to interpolate between two known coordinates, and get an approximation of the trajectory. We can conclude that the use of control points to describe a trajectory does not reduce the quality of the trajectory, as observed in [KMdW06].

Numerous applications exist that generate and process moving object data. Examples of these applications can be found in biology, ecology, telecommunications, city planning and tourism. For example, in the field of tourism, we can observe the movement of tourists and how they move. By analyzing the collected data we can discover information that would give us perspective on transportation needs, marketing opportunities or background information on city development.

Applications and techniques that focus on mining these trajectories already exist. However, these techniques have one major drawback: in many cases, the result of the mining process lacks the desired level of information. In other words, although we have results, they are low in semantic contents. Therefore, after the analysis of the data, the results of this process need to

be translated to a semantically richer model.

This lack of semantics was the inspiration to describe a semantically enriched model [ABK+07] for representing trajectories. Instead of transforming the results after the analysis, the input can de transformed prior to mining in order to concentrate more on the actual semantics during the process. This model is a keystone for the input of the mining process described in this thesis, as we try to emphasize and focus on the semantics of the trajectories.

1.1.2 Knowledge Discovery in Databases

As a guideline for the following descriptions and definitions of important notions and concepts, we use the books written by Han and Kamber [HK00] and Tan, Steinbach and Kumar [TSK05].

Data mining is the process of automatically analyzing large amounts of data, with the purpose of finding new, non-trivial and useful information, often in the form of patterns or prediction models. It is part of the Knowledge Discovery in Databases process, referred to as KDD, a process with the purpose of transforming raw data in useful information.

As Figure 1.2 shows, it is possible to divide the KDD process into three phases:

- preprocessing
- data mining
- postprocessing

We will now give a brief description of every phase.

Preprocessing

The first phase is the preprocessing phase, with the purpose of preparing the data for processing. It entails data cleaning, integration, selection and transformation.

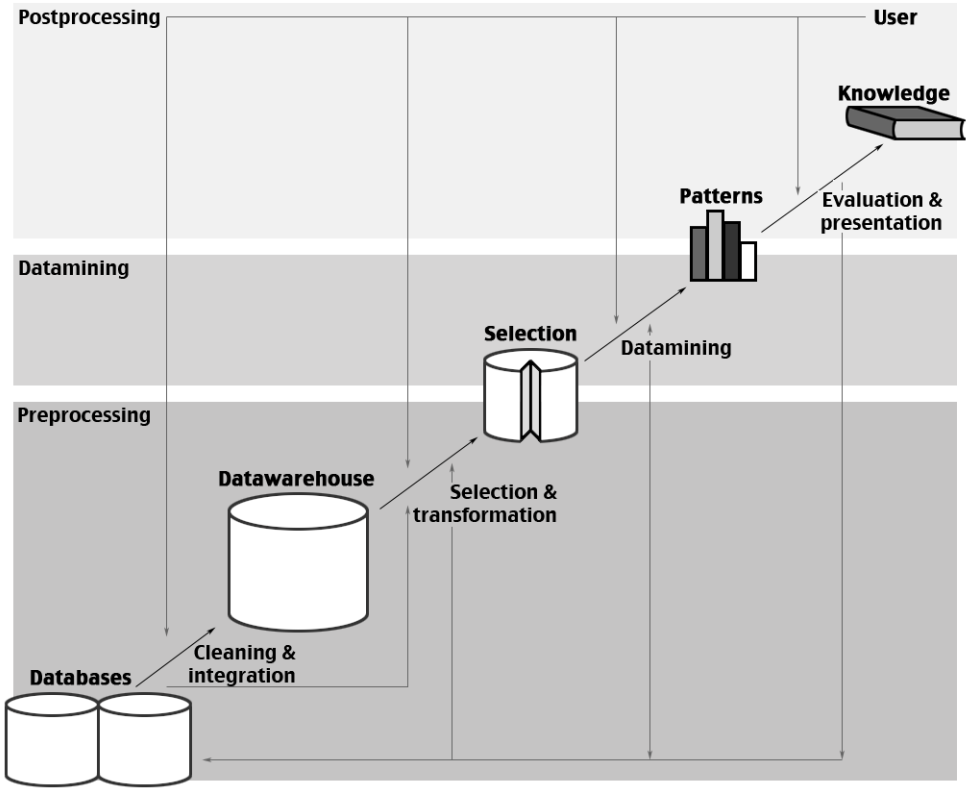

Figure 1.2: Overview of the phases of Knowledge Discovery in Databases.

Considering our tourism example, this phase could have the following scenario. The first phase is data cleaning, which entails the removal of noise from the dataset, data which can not be used, for example trajectories consisting of a single point, or incorrect data, maybe because of a technical malfunction during data acquisition. An example of incorrect data could be a tourist that moves through Paris with a speed of 400 km/h. As such a trajectory is unlikely to be correct, it would be removed from the dataset. After the data cleaning, several different datasets may be merged, as we may wish to use datasets from different sources e.g., if data is stored at multiple locations.

Now that we have a single dataset, we might only want to take into account trajectories that are situated in the Paris area, so the user is given the opportunity to select which trajectories he wishes to mine. Finally, our selected trajectories may not be in the preferred format, depending on the kind of analysis we wish to perform. This does not only entail formatting

the trajectories into a model specified to the type of mining, but also to take the specifications of the user into account. For example, timestamps give a very exact indication of time, but perhaps the user wishes to transform these timestamps so they only represent years, seasons, months or days of the week.

Data mining

The second phase is data mining, a series of algorithms allowing us to analyze the preprocessed data. There are four major categories in which these techniques can be divided:

- association analysis
- classification and prediction
- clustering
- anomaly detection

Association analysis Association rule mining finds interesting association or correlation relationships in a large set of data items [HK00]. By analyzing these large amounts of data, also referred to as transactions, we are able to formulate association rules, for example $A \rightarrow B(sup = x\%, con = y\%)$. Each rule consists of a antecedent and a consequent, respectively A and B in our example. The actual meaning of $A \rightarrow B$ is that the presence of B is implied by the presence of A. Every rule is associated with a support and confidence, respectively represented by x and y, measuring the interestingness of a given rule. The support of a rule expresses how many of the transaction contain the rule, giving an indication of its usefulness. For example, it is possible that 2% of the transactions in our set contain A and B. Confidence expresses how many of the transactions containing A, also contain B, giving an indication of its reliability. For example, a confidence of 60% means that a transaction containing A, has a 60% chance of also containing B. These two measures are important as we will often need to define a minimum support and a minimum confidence threshold for the rules we wish to find.

One of the best-known examples of association analysis, is its use on shopping baskets. Here the content of large amounts of baskets is analyzed in order to try to understand the buying behavior of customers and formulating this behavior into rules. Once the analysis has been completed, its results can be used for various of purposes, going from marketing campaigns aimed at specific markets or products to determining the layout of a shop.

A textbook example of a rule found in basket analysis is $Diapers \rightarrow Beer(sup = 0,5\%, con = 60\%)$ which tells us that 0,5% of the transactions contain both Diapers and Beer, and that in 60% of the transactions that contain Diapers also the item Beer can be found. This rule therefore describes a strong correlation between both Diapers and Beer.

Association analysis can be interesting from a trajectory point of view to find rules defining mobility behavior. Rules describing how tourists move in a city, what locations they visit or which hotels they prefer.

Classification and prediction These two forms of data analysis can respectively be used to extract models describing important data classes or to predict future data trends. The main difference between the two is that classification predicts categorical labels, whereas prediction focuses on continuous-valued functions [HK00].
The classification process consists of two phases. The first phase is a learning process, where, depending on the technique being used, a model is created, describing a predetermined set of classes. In order to construct such a model, a training set is used, consisting of a set of tuples whose categorical labels are known in advance. The output of this phase can either be a set of classification rules or a classification tree. Figure 1.3 gives an example of how a possible classification tree might look like. As classification uses a training set with predefined labels, this phase is also called supervised learning. The second phase tests the constructed model by applying a test set, another set of tuples of which the label is known, in order to determine its accuracy.
Prediction is similar to classification, but it uses regression to model the continuous-valued functions.
Examples of classification can be found in many different fields: approving bank loan applications and detecting spam e-mails while examples of prediction can be found in transport management and weather forecasts.

Classification could also be applied to mobility data. For example, consider the case where we would collect trajectory data not only of tourists, but also of different types of people, all moving in the city. By analyzing these trajectories, we could possibly create a classification model allowing us to classify different types of people, to determine whether they are students, tourists or people working or living in the city. Such a model would not only allow us to classify people based on their movements, but would also describe each type's specific behavior, e.g., his/her transportation needs.

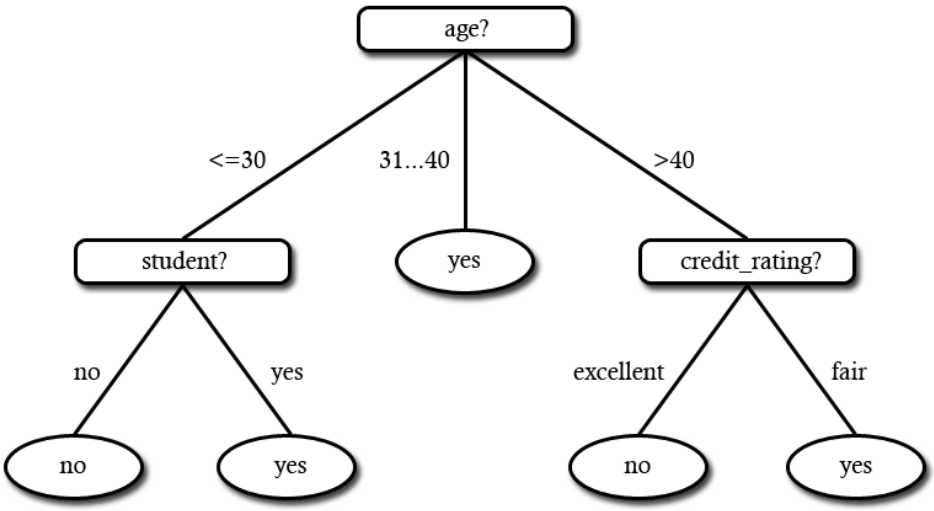

Figure 1.3: Example classification tree: determining whether a customer will buy a computer based on his age, credit rating and whether he is a student or not. [HK00]

Clustering Cluster analysis groups data objects based only on information found in the data that describes the objects and their relationships. The goal is that the objects within a group be similar (or related) to one another and different from (or unrelated to) the objects in other groups [TSK05]. Its principle is similar to that of classification, but clustering does not need predetermined classes, it determines the class labels on its own, hence the term unsupervised learning. It determines the similarity between the objects, and maximizes the similarity within a cluster, and minimizes the similarity between clusters.

An example of how clustering works, is given in Figure 1.4, taken from [TSK05]. This is an example of a possible input for clustering where items are projected on a two-dimension plane, where the X- and Y-axis both represent some feature of these items. Using these selected features, clusters can be formed based on the similarity principle mentioned previously. As the example demonstrates, it is possible for the user to determine the amount of clusters, depending on the actual clustering technique used.

Clustering can also be applied on trajectories, as Figure 1.5, taken from [GP08], demonstrates. It shows how in a collection of trajectories two clusters

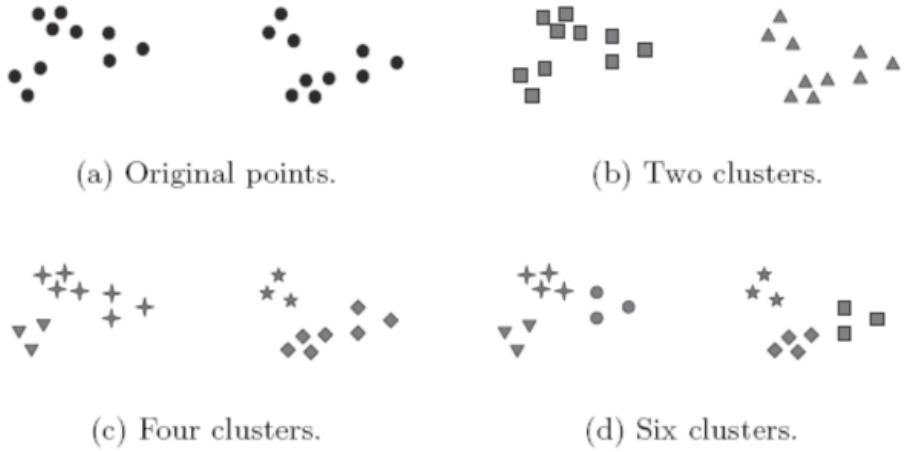

Figure 1.4: Example of input and result of clustering. [TSK05]

can be found based on their geometrical properties.

Anomaly detection Clustering does not only allow the grouping of objects according similarity, it also allows us to determine when an object belongs to no group. When an object is dissimilar from any other, this object might be of interest to us, particularly when the purpose of our mining is to find *outliers*. An example of the use of this technique is fraud detection with credit cards.

Postprocessing

The third and final phase is the postprocessing phase of the KDD process. This phase has two goals. First, we have to make sure that the results given to the user are of interest to him. Possibly some filtering is needed, because a part of the result might be common knowledge or even the number of results too large. Thus, this phase might include an analysis of the results to conclude if results may be omitted. Filtering results is very important, as demonstrated in [JÖ7], as sometimes a part of the mining results are common knowledge. However, previously cited work focuses on the filtering of results during the mining process.

The second goal is representation, converting the results of the mining process into a format the user wants and understands. Choosing the right

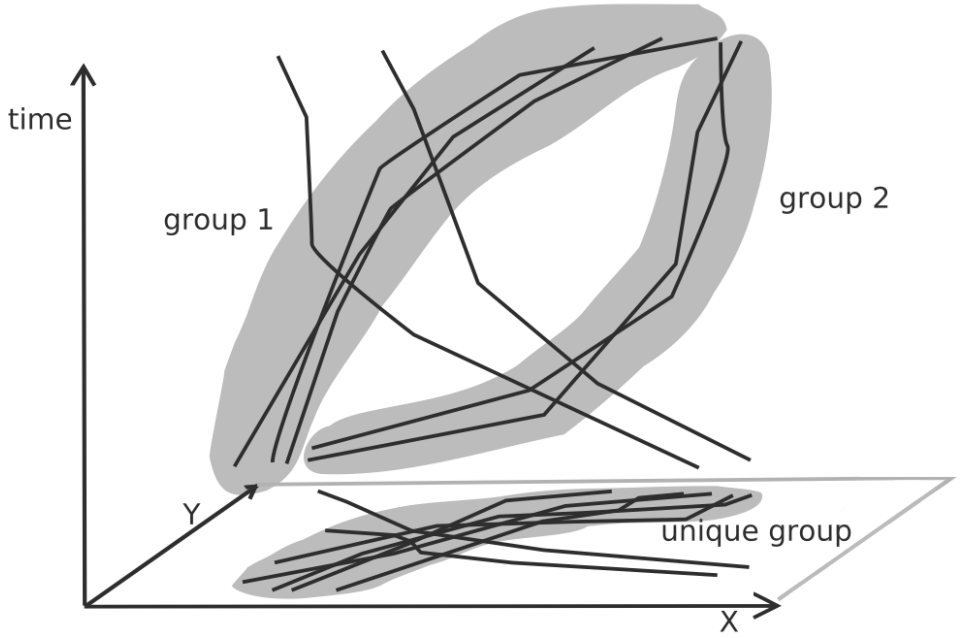

Figure 1.5: Example of trajectory clustering, clustering based on the geometric properties of trajectories. [HK00]

representation is of course of great importance as it is the output of our KDD process. One option is summarizing the results in a textual report. However, as visualized results are often more clear and easy to understand, this will often be the preferred format of representation, if visualization is possible. An example of such a representation is the classification tree in Figure 1.3, but could also be charts, diagrams, ... Visualization is also important to trajectories and derived patterns, as [AA06] demonstrates.

1.2 Problem statement, Motivation and Contribution

1.2.1 Problem

As mentioned, trajectory mining was previously focused on raw trajectories, where the semantics were added, based on background information, as a postprocessing phase. However, during this process, there was little focus on the semantics, losing the opportunity to involve the actual meaning of these trajectories in the mining. By introducing the notion of *semantically enriched trajectories*, we are given the opportunity to mine trajectories using

their semantics and generate results that emphasize their meaning.

1.2.2 Motivation

The possibilities from analyzing trajectories are endless. Consider a tourist application where trajectories of tourists visiting a city are recorded. By analyzing these in their raw form, we might be able to recognize large amounts of trajectories which are similar to each other. However, if we would observe these from a semantic point of view, we wouldn't just focus on its size, direction, location, ..., but we would focus on what a tourist actually encounters on his trajectory.

Consider a tourist roaming the streets of Brussels. While he explores the city and visits museums, shops, bars, restaurants, monuments and hotels, we record his/her movement. This movement is then translated, by the process mentioned before, into a list of interesting touristic locations. Consider the fact that we record the movement, not of a single tourist, but of a large group of tourists. Not only would this enable us to get an overview of the locations tourists find interesting, but also which combinations of locations are visited within the same day or which hotels attract which type of tourist. For example if an analysis would reveal that a considerable group of tourists visits the exact same museums on the same day, and these visits are in a particular order. We can take advantage of this fact by introducing a special ticket combining these museums, by providing transportation or by promoting this set of museums to other tourists.

1.2.3 Contribution

Our goal is to find patterns hidden in mobility data. This data, essentially composed of large amount of trajectories, is transformed into a semantically enriched format [dMR05, SPD+08] allowing us to emphasize its actual meaning. This format focuses on background information, allowing us to use this background information during the entire process of analysis, as opposed to using this information to interpret the found results after the analysis.

The contribution of this thesis lies on trying to use the specific characteristics of semantically enriched trajectories to define an association analysis technique that is able to find frequent trajectories. In order to succeed, we observe and define a data model capable of representing the semantics of semantically enriched trajectories, and adapt an algorithm known as Apriori,

introduced in Chapter 3, which is the basis of our own algorithm, referred to as Stops and Moves Apriori. Our data model is defined in such a way, as to allow the use of efficient techniques that will have a direct influence on the performance of our algorithm. Finally, we observe the results of our mining process, and analyze how these could be presented to the user.

1.3 Outline

This thesis is organized according to the following structure. First, in Chapter 2, the data that forms the base and input of our mining process, the mobility data, is described. Also in this chapter, techniques allowing trajectories to be transformed into a semantically enriched data model, namely the SMoT [ABK+07] and the CB-SMoT [PBKA08] algorithm, are introduced, together with a comparison between these two.

In Chapter 3, we introduce our own data model and give an overview of the process to adapt the existing Apriori method to allow it being applied to enriched trajectory data. The process summing up the needed adaptations will be a step-by-step approach. We start by introducing our Stops and Moves Apriori, first without wildcards, and expand this algorithm to include single- and multi-matching wildcards. Further observations are made of how the output of the Apriori algorithm, namely frequent sequences, can be transformed into rules.

Next, in Chapter 4 we describe the implementation of our Stops and Moves Apriori. Here the more technical details of the implementation of the technique will be summed up, together with an example of the technique applied on existing data.

Finally, Chapter 5 concludes this thesis, summarizing up the result, and proposes future works.

Chapter 2

Mobility data

2.1 Trajectories

2.1.1 Definitions

When an object moves in two- or three-dimensional space, the path of its movement describes a trajectory. In the real world, trajectories are a continuous function of time, where every instant in time t returns a tuple $\alpha(t) = (\alpha_1, \alpha_2, \ldots, \alpha_d)$ of d coordinates representing the position of an object at time t in a d-dimensional space. We can formalize this by a function $\alpha : \mathbf{R} \to \mathbf{R}^d$, where $\mathbf{R}$ is the set of real numbers. In the following definition [KO07] of a trajectory, we limit ourselves to movement in the two-dimensional plane $\mathbf{R}^2$. Using this definition, $\mathbf{R} \times \mathbf{R}^2$ expresses time-space space, where the first $\mathbf{R}$ represents time, and $\mathbf{R}^2$ space.

Definition 1. Let $I \subseteq \mathbf{R}$ be an interval. A *trajectory* T is the graph of a piecewise smooth (with respect to t) mapping

$$\alpha : I \subseteq \mathbf{R} \to \mathbf{R}^2 : t \mapsto \alpha(t) = (\alpha_x(t), \alpha_y(t)),$$

that is, $T = \{(t, \alpha_x(t), \alpha_y(t)) \in \mathbf{R} \times \mathbf{R}^2 \mid t \in I\}$. The set I is called the *time domain* of T. □

However, from an application perspective, this model is impractical. Recording and storing a trajectory as a continuous function could be very difficult since in real-life these are often obtained by using mobile location aware devices such as GSM and GPS equipment, where at regular intervals the coordinates are registered. This does not allow the trajectories to be recorded

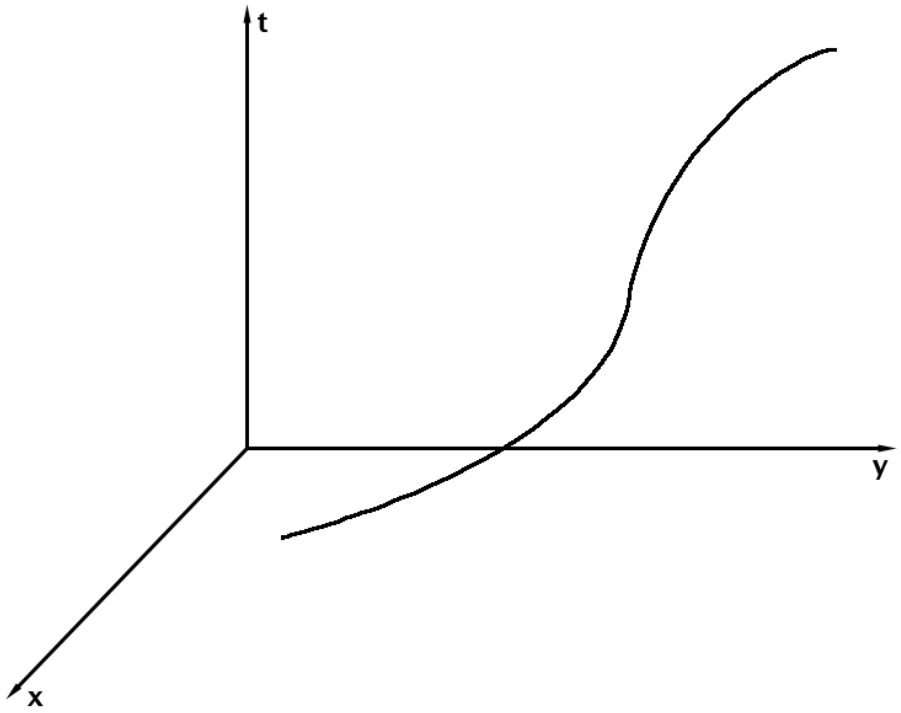

Figure 2.1: An example of a trajectory.

with the needed frequency. Therefore we use trajectory samples. Trajectory samples are a sequence of coordinates describing the trajectory. Each of these coordinates is then linked to a timestamp, thus creating an ordinal relation on the coordinates. More formally, trajectory samples can be defined as follows. Like the previous definition, this restricts itself to the real plane $\mathbf{R}^2$ to represent the coordinates of the moving object.

Definition 2. A *trajectory sample* is a list of time-space points $\langle (x_0, y_0, t_0), \ldots, (x_N, y_N, t_N) \rangle$, where $x_i, y_i, t_i \in \mathbf{R}^2$ for i = 0, ... N and $t_0 < t_1 < ... < t_N$. □

Definition 3. Let $S = \langle (t_0, x_0, y_0), \ldots, (t_N, x_N, y_N) \rangle$ be a sample of a trajectory and let $T = \{(t, \alpha_x(t), \alpha_y(t)) \in \mathbf{R} \times \mathbf{R}^2 \mid t \in I\}$ be a trajectory. We say that the trajectory T is *consistent* with the sample S, if $t_0, t_1, \ldots t_N \in I$ and $\alpha_x(t_i) = x_i$, $\alpha_y(t_i) = y_i$ for $i = 0, ...N$. □

Based on this trajectory sample, an approximation of the original trajectory can be reconstructed using the linear-interpolation model. Assuming a

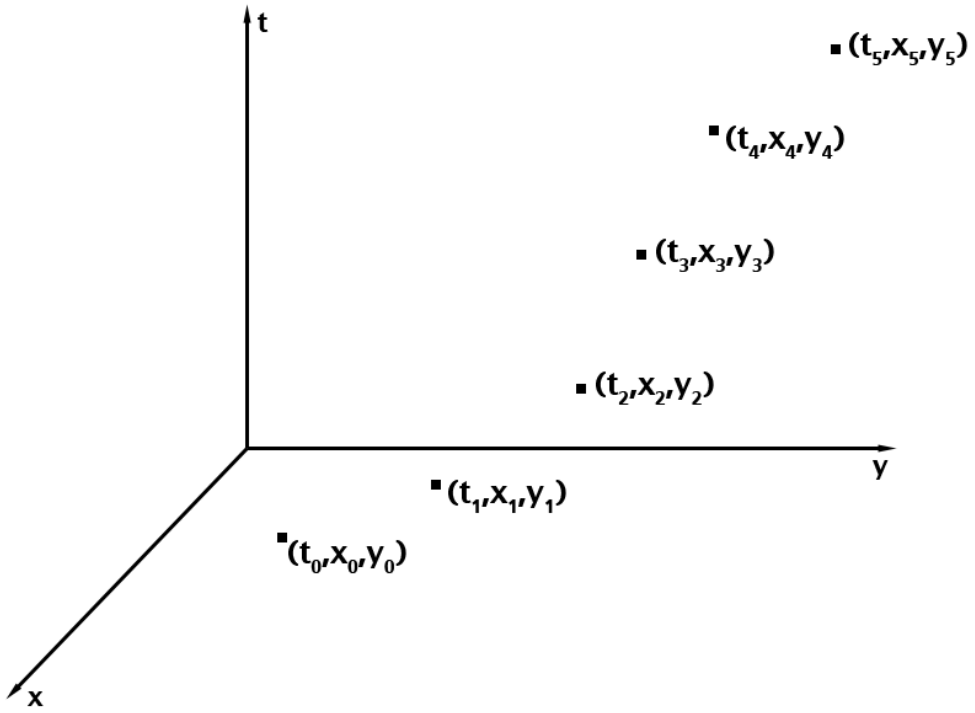

Figure 2.2: An example of a trajectory sample based on the trajectory depicted in Figure 2.1.

constant lowest speed between two consecutive sample points, this model allows us to construct a trajectory that is consistent with the trajectory sample. For a sample $S = \langle (x_0, y_0, t_0), \ldots, (x_N, y_N, t_N) \rangle$, the trajectory $LIT(S) :=$

$$\bigcup_{i=0}^{N-1} \{(t, \frac{(t_{i+1} - t)x_i + (t - t_i)x_{i+1}}{t_{i+1} - t_i}, \frac{(t_{i+1} - t)y_i + (t - t_i)y_{i+1}}{t_{i+1} - t_i}) \mid t_i \leq t \leq t_{i+1}\}$$

describes its the *linear-interpolation trajectory*. The functions that are used to determine the x- and y-coordinates are differentiable except maybe at the instances of time $t_0, t_1, \ldots, t_N$.

The trajectories and trajectory samples used in the mining process are stored in a database, or more precisely in a relation, which is a part of the database. More formally, this database and its relation can be described as follows. We assume the existence of an infinite set **Labels** $= \{a, b, \ldots, a_1, b_2, \ldots, a_2, b_2, \ldots\}$ of *trajectory labels*.

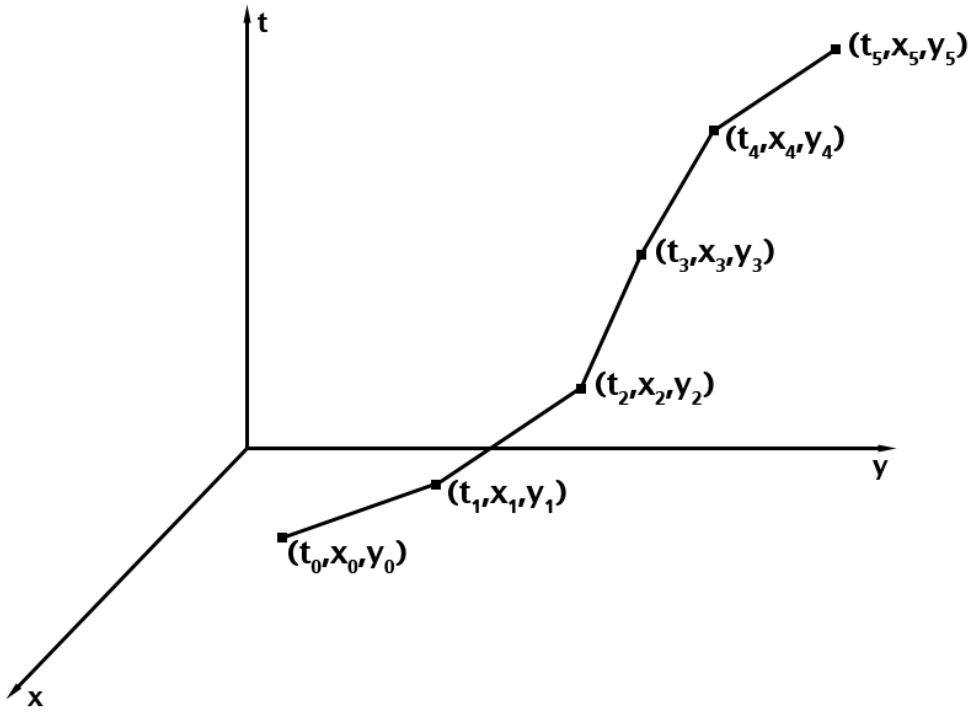

Figure 2.3: An example of a linear-interpolation trajectory based on the trajectory sample depicted in Figure 2.2.

Definition 4. A *trajectory relation* R is a finite set of tuples $(a_i, T_i), i = 1, \ldots, r$ where T_i is a trajectory and $a_i \in$ **Labels** uniquely identifies this trajectory and thus can only appear once. Similarly, a *trajectory sample relation* R is a finite set of tuples $(a_i, t_{i,j}, x_{i,j}, y_{i,j})$ with $i = 1, \ldots, r$ and $j = 0, \ldots, N_i$ such that $a_i \in$ **Labels** cannot appear twice in combination with the same t-value and that $\langle (t_{i,0}, x_{i,0}, y_{i,0}), (t_{i,1}, x_{i,1}, y_{i,1}), \ldots, (t_{i,N}, x_{i,N}, y_{i,N}) \rangle$ is a trajectory sample.
A trajectory (sample) database is a finite collection $\{R_1, R_2, \ldots, R_M\}$ of trajectory (sample) relations. □

In this section, we have defined the trajectories and trajectory samples based on coordinates. From a semantic point of view, this is a low-level representation, as it does not give information on the geographic or semantic characteristics of the trajectories. This nature can be retrieved using background information. We refer to this class of trajectories as *raw trajectories*. Besides these raw trajectories, we could also distinguish semantically enriched trajectories, which we will focus on in the next section.

2.2 Semantically enriching trajectory data

Most of the mining techniques that were developed for trajectories, use raw trajectories. But as we have already pointed out in Chapter 1, there are advantages gained from lifting a raw trajectory into a model that concentrates on its semantics. In this section, we will focus on this model, and also on techniques to transform raw trajectories into semantically enriched trajectories.

2.2.1 Enriched trajectories

The model of semantically enriched trajectories, proposed in [dMR05, SPD+08], attempts to combine raw trajectory data and background geographic information, e.g., city maps indicating touristic locations, geographical maps indicating rivers, districts or cities. Instead of defining a trajectory as a sequence of coordinates combined with a timestamp, they propose to transform the trajectory into a sequence of stops and moves. These stops and moves would in turn incorporate geographic background information making it unnecessary to reestablish the link between the raw trajectories and its background geographical information.

The principle is simple, the stops represent areas that are considered important to the trajectory, the moves represent the part of the trajectory between stops (and the start and end point) of the trajectory.

As an introduction to the definitions of stops and moves, we will first introduce the notions of an application and candidate stop:

Definition 5. A *candidate stop* C is a tuple (R_c, Δ_c), where R_c is a topologically closed polygon in $\mathbf{R}^2$ and Δ_c is a strict positive real number. The set R_c is called the *geometry* of the candidate stop and Δ_c is called its *minimum duration*. The candidate stops are dependent of specific applications. An *application* is a finite set $\{C_1 = (R_{c_1}, \Delta_{c_1}), C_2 = (R_{c_2}, \Delta_{c_2}), ..., C_N = (R_{c_N}, \Delta_{c_N})\}$ of candidate stops with non-overlapping geometries $R_{c_1}, R_{c_2}, ... R_{c_N}$. □

These candidate stops are application dependent and could be anything from roads, buildings or parks, as long as they can be represented by a polygon. With each of these polygons, representing objects, we associate a minimum duration enabling us to distinguish a stop from a pass-through. These minimum durations can differ from candidate to candidate and from

application to application. So we can conclude that determining the candidate stops is a semi-manually process, as the background information can be gathered automatically, but user input is still necessary.

Our running example of an application, is this of a city, where we focus on its touristic nature. The candidate stops, all represented by polygons can be information kiosks, hotels, restaurants, museums, parks, shops or squares. The moving objects are tourists that move around in this city. These trajectories can later be transformed to a ordered list of locations.

Having defined applications and candidate stops, we can now proceed by defining stops and moves:

Definition 6. A *stop* of trajectory sample T with respect to an application A is a tuple (R_k, t_j, t_{j+n}) such that a maximal subtrajectory S of T with $S = \{(x_i, y_i, t_i) | (x_i, y_i) \in R_k\} = \{(x_j,\ y_j,\ t_j),\ (x_{j+1},\ y_{j+1},\ t_{j+1}),\ ...,\ (x_{j+n},\ y_{j+n},\ t_{j+n})\}$, where R_k is the geometry of C_k and $|t_{j+n} - t_j| \geq \Delta_k$. □

When a person walks through our tourist city, his/her trajectory will intersect many of our candidate stops. When this person remains within the polygon representing the candidate stops for a period of consecutive time points longer than the minimum duration of this candidate stop, we add that stop to our sequence. We now know that our person remained in this stop, and did not just pass through it. It can be considered an important part of our moving objects trajectory.

Once all stops are determined, the list of moves can be composed.

Definition 7. A *move* of a trajectory sample T with respect to an application A is:

1. a maximal contiguous subtrajectory of T in between two temporally consecutive stops of T
2. a maximal contiguous subtrajectory of T in between the starting point of T and the first stop of T
3. a maximal contiguous subtrajectory of T in between the last stop of T and the last point of T
4. the trajectory T itself, if T has no stops. □

Because stops are derived from a series of samples intersecting a candidate stop, we can observe that the start and end point of the subtrajectory composed of these samples will also lie within a candidate stop. These points will respectively be used for the end/start points of the preceding/following moves.

2.2.2 Enriching trajectories: SMoT

The first algorithm enabling us to enrich trajectories is SMoT, Stops and Moves of Trajectories, which was proposed and elaborated on in [ABK+07]. This technique encapsulates the idea of stops and moves, but also shows us some of the disadvantages of the Stops and Moves of Trajectories model.

The input of this algorithm is the given trajectories that are the subject of our mining process and the application consisting of candidate stops. As we will check whether points intersect with candidate stops, we will define a buffer around each polygon. This will allow us to intersect with small polygons, lines and points more easily, and allow a small margin of error (due to measurement imprecision) concerning coordinates.

Starting from the first point of the trajectory, the algorithm checks if this point intersects a candidate stop and its surrounding buffer, if not, the next point is considered. But if the point does intersect a candidate stop, the time point of this point is recorded. The next points are also checked and if the time spent within the polygon of the candidate stop becomes greater than its minimum duration, this stop is recorded. Between each of the recorded stops, moves are inserted to connect consecutive stops.

To demonstrate how SMoT exactly works, we observe an example of a possible input in Figure 2.4. Consider a trajectory, consisting of sample points, and an application, a map containing polygons, each a candidate stop, which could, for example, represent a touristic object. Each of these objects is also associated with a minimum duration. Based on these two, the trajectory can be transformed in two lists, a list of stops and a list of moves. We can observe that the first sample point s_0 intersects with $Hotel_1$, as do all the sample points up to s_6. As the time between s_0 and s_6 is larger than the minimum duration of $Hotel_1$, these points become the first stop of our list. The next sample point, s_7 does not intersect with a candidate stop and is therefore ignored. Based on the pseudo-code, we can see that after the creation of a stop, we first check sample points up to the point

Listing 2.1: SMoT pseudo-code

```
1  Input: T //set of trajectories
2         A //application
3  Output: S //set of stops
4          M //set of moves
5  Method:
6  S = new Stops(); M = new Moves();
7  for each trajectory t ∈ T do
8    i = 0; previousStop = null;
9    while(i ≤ size(t)) do
10     if (∃(R_C, Δ_C) ∈ A |
11         geometry(t[i]) intersects R_C) //using spatial index
12       enterTime = time(t[i]);  i++;
13       while (intersects(t[i], R_C)) do
14         i++;
15       endwhile
16       i--; //Go one step back (went outside R_C)
17       leaveTime = time(t[i]);
18       if (leavetime-entertime >= Δ_C)
19         stop = (t, R_C, enterTime, leaveTime);
20         S.add(stop);
21         move = (t, previousStop, stop,
22           previousStop.leaveTime, enterTime)
23         M.add(move);
24         previousStop = stop;
25       endif
26     endif
27     i++;
28     j = 1;
29     while((i+j ≤ size(t)) and (t[i+j]-t[i]<min_ΔC(A))) do
30       j++;
31     endwhile
32     if(∄(R_C, Δ_C) ∈ A | geometry(t[i+j-1]) intersects R_C)
33       i = i+j;
34     endif
35   endwhile
36   if (t[i-1] not ∈ previousStop) //t do not end with a stop
37     move = (t, previousStop, null,
38       previousStop.leaveTime, time(t[i-1]))
39     M.add(move);
40   endif
41 endfor
```

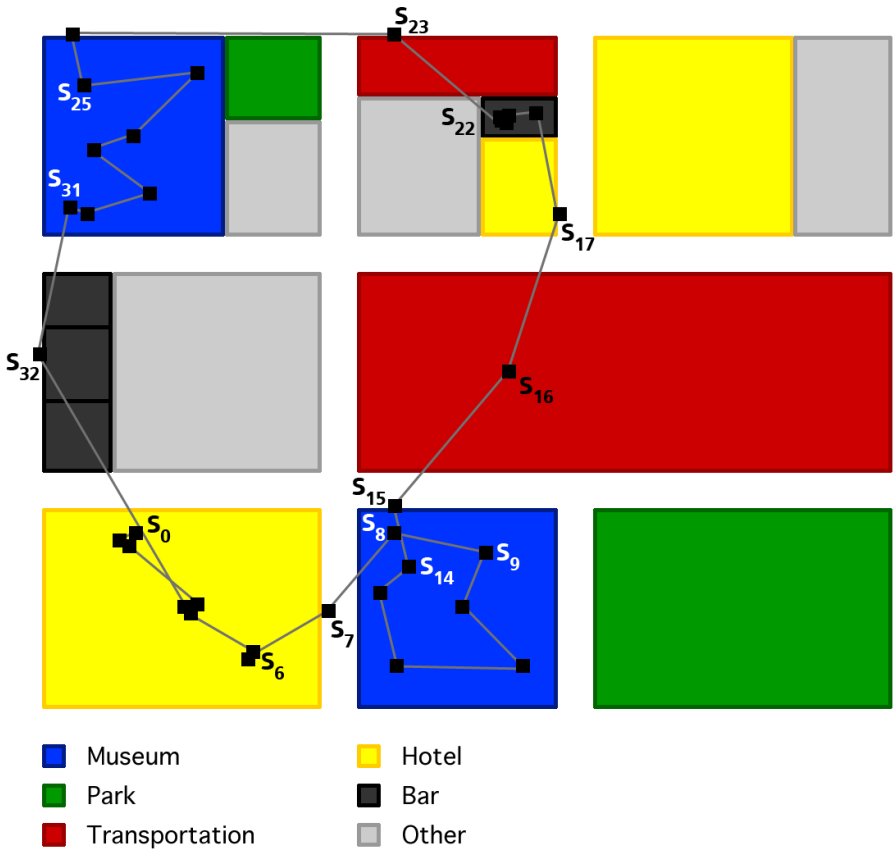

Figure 2.4: An example of input from application and sampled trajectory.

where the duration of the run is larger than the smallest minimum duration, if we still do not intersect a candidate stop, we can ignore the entire run of sample points, as they will not contain a possible stop. In our example we will thus evaluate several sample points, for example up to s_9. As s_9 intersects $Museum_1$, we go back up to s_7 ignore this sample point as it does not intersect with a candidate stop. The subtrajectory from s_8 up to s_{14} intersects a candidate stop, and consequently we also add a move to the list of moves, namely from s_6 up to s_8. Next, the subtrajectory from s_{15} up to s_{17} cannot de transformed into a stop, even if s_{16} intersects a candidate stop, because the minimum duration of $Transportation_1$ is not reached. The next stops are Bar_1, derived from s_{17} up to s_{22}, and $Museum_2$, derived from s_{25} up to s_{31}. The sample points in between two stops, from s_{14} to s_{17} and from

s_{22} to s_{25} are transformed into two moves. Finally, another move is added containing all remaining sample points, s_{25} up to s_{31}, are transformed into a move, as no run of samples reaches the minimum duration of a candidate stop. The two lists, the stops and moves, are outputted.

2.2.3 Enriching trajectories: CB-SMoT

Although the idea of SMoT is very simple and reasonably easy to accomplish, the drawback of this technique is the high dependency of users to specify the interesting geographic objects that are considered candidate stops. This dependency raises some issues. First, human interaction in error-prone, by reducing the need for interaction, errors can be avoided. Second, specifying all interesting candidate stops can be difficult, as with some applications, a user cannot always foresee how the patterns will evolve.

A solution can be found in the CB-SMoT technique, proposed in [PBKA08], which instead of completely relying on the input of users, tries to find the potential stops on its own, after which these are matched to candidate stops given by the user. In addition to finding potential stops that match candidate stops, it is also able to find potential stops where no candidate stop was defined, essentially discovering new candidate stops. The idea behind this technique relies on the understanding that if a geographic object is considered important to the trajectory of a moving object, this moving object slows down as it nears the geographic object. As it slows down, the points representing the trajectory become denser. By using a clustering technique, we are able to pinpoint areas where the moving object slows down, and identify these as stops.

As mentioned above, clustering techniques are used to find areas where the moving object slows down. To be more exact, DBSCAN [EKSX96], a density based algorithm, is used, as our clustering technique, modified to the needs of trajectory data. One of these modifications is the notion that only the distance of points within a single trajectory can be measured. This distance is not the Euclidean distance, but the total length of the subtrajectory between the two points. The measure of defining clusters is modified to a certain extend to take this into account.

The pseudo-code in Listing 2.2 shows how CB-SMoT is defined. To fully understand the algorithm, we need to explain some of the functions and notions used. The first function we would like to clarify, is $quantile(\mu(T), \sigma(T),$

Listing 2.2: CB-SMoT pseudo-code

```
1  \textbf{Input}: T //set of trajectories
2        A //application
3        a //area for the quantile function
4        minTime //minimum time for clustering
5  \textbf{Output}: S //set of stops
6         M //set of moves
7  \textbf{Method}:
8  S = new Stops(); M = new Moves();
9  for each trajectory T ∈ T do
10   set clusters as empty;
11   Eps = quantile(μ(T), σ(T), a);
12   for each unprocessed point p ∈ T do
13     neighbors = linear−neighborhood(p,Eps);
14     if p is a core point with respect to minTime, Eps
15       for each neighbor n ∈ neighbors do
16         add to neighbors every unprocessed point
17             ∈ linear_neighborhood(n, Eps);
18         add to clusters neighbors;
19         set points in neighbors as processed;
20       endfor
21     endif
22   endfor
23   for each cluster C ∈ clusters do
24     for each (R_C, Δ_C) ∈ A |
25           C intersects R_C for a duration time time ≥ Δ_C do
26       sub = subtrajectory(R_C, C);
27       stop = (T,R_C,enterTime(sub,leaveTime(sub)));
28       S.add(stop);
29     endfor
30     for each subtrajectory s ∈ C that is not stop
31       if leaveTime(s) − enterTime(s) ≥ minTime
32         stop = (T,Unknown,enterTime(sub,leaveTime(sub)));
33         S.add(stop);
34       endif
35     endfor
36     for each subtrajectory s ∈ C not converted into stop
37       s_1 = stop preceding s; s_2 = stop following s;
38       move = (t,s_1,s_2,leaveTime(s_1),enterTime(s_2));
39       M.add(move);
40     endfor
41   endfor
42 endfor
```

a). This is used to calculate the *Eps*-value, the value that represents the absolute distance used to calculate the neighborhood of a point. Without going into too much detail, as the exact calculating of *Eps* lies outside the scope of this thesis and we refer to the original paper [PBKA08], we can say that *Eps* is computed by using $\mu(T)$, which is the arithmetic mean of the distances between the points of T, $\sigma(T)$ the standard deviation of these distances and a, a value between 0 and 1, which should provide an approximation of the proportion of points that generate potential stops in relation to the total amount of points in the trajectory and is provided by the user.
The second function that needs some explanation is $linear\_neighborhood(n, Eps)$. Using the previously describe *Eps*-value, this function composes the collection containing all points near the point n.
Finally, we describe the notion of core point. This is defined to be a point $p = (x, y, t)$ of trajectory T of which the time span of its linear-neighborhood exceeds $minTime$. Or, if p_n and p_m are respectively the first and last point of the linear-neighborhood of p, then p is a core point if $|t_n - t_m| > minTime$.

Figure 2.5 demonstrates how the algorithm works. It works in two phases. First, it identifies areas that could be regarded as interesting by detecting whether the moving object slows down. These are indicated in the figure by the grey ellipses. Detecting slower movement can be achieved by the clustering mentioned before. Once all clusters have been identified, we match these clusters onto candidate stops selected by the user. For these clusters to become stops, they still have to comply with the minimum duration of the matching candidate stops. $Stop_1$, $Stop_2$ and $Stop_3$ are for example three stops that could be recognized in this way. However, there is one cluster that could not be matched onto a candidate stop. As this cluster and a potentially interesting stop cannot be ignored, it is labeled as an 'unknown stop' and output to the user whom is able to reject or accept this 'unknown stop' as a stop of the trajectory.

2.2.4 A comparison of SMoT and CB-SMoT

Applicability

SMoT As SMoT uses user-defined candidate stops, these stops become the weakness and strength of this technique. In a type of application where the candidate stops can be easily identified or where new candidate stops are considered irrelevant, this technique can be applied. For example, in an application where movement of tourists is analyzed, the candidate stops are the collection of sights, hotels, bars, ... Most of them can be extracted

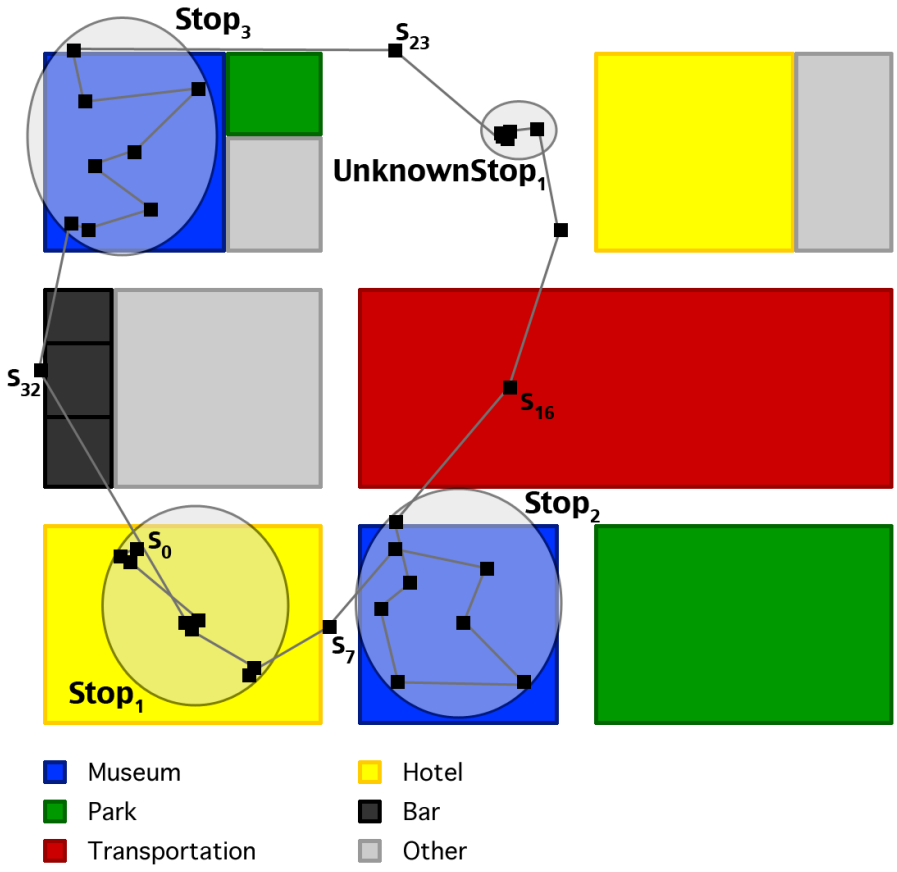

Figure 2.5: CB-SMoT uses an application and sampled trajectory as input to determine stops and moves.

from city plans and therefore easily recognized. Other possible candidate stops could merely be the result of traffic congestion, and possibly irrelevant to our application.

CB-SMoT However, in case of an application where the number of candidate stops is very large (e.g., junctions in a city), it is difficult or even impossible to pinpoint these exactly (e.g., resting points for migrating birds). It is also possible that the user considers every possible candidate stop as interesting (e.g., analysis of traffic congestion in a city). In both cases CB-SMoT can be more interesting than the SMoT technique because even if a certain candidate stops where not included in the application, CB-SMoT will

recognize them and point them out as 'Unknown stop'.

Quality

SMoT The entire algorithm is dependent on the quality of its input, on the definition of its candidate stops and the minimum durations. Since all these are user-defined, we can conclude that the quality is determined by the user. As users are more prone to errors, this might influence the quality.

CB-SMoT Candidate stops can be determined dynamically, by calculating when an object slows down. The fact that it focuses on speed makes it on one hand less error-prone than SMoT, as this is independent on user input. On the other hand, when speed is not the best way to determine stops, errors can be made as uninteresting stops can be inserted in the list, and interesting stops can be ignored.

For example, consider a touristic application where we track the movements of tourists in a city. If a tourist moves slowely or stands still, s/he might be waiting for transportation, as in this case the points are densely situated so this would be recognized as a stop. However, if he would walk in a park, a historical center or a museum at a fast pace, these points would be less dense, and might not be recognized as a stop.

Speed

SMoT A good measure to compare the speed between these two algorithm, is to determine their complexity. We use the pseudo-code provided in Listing 2.1 as a guideline to define the complexity of SMoT.
Consider N to be the number of trajectories and M to be the average number of sample points a trajectory is composed of. We start at Line 7, where a for-loop is defined, which is executed N times, as the code from Lines 8-40 is executed for each trajectory. Another loop is defined at Line 9, this will run M times, as this loop is used to evaluate each sample point in the trajectory. Lines 10-11 define an if-test determining whether a point intersects a candidate stop. Instead of evaluating this for each candidate, a spatial index is used. Because of this index, this instruction can be evaluated in constant time. The loop defined from Lines 13-15 uses the same variable used by the loop defined at Line 9, so the amount of loops here result in a lower amount of loops in the loop at Line 9. Consider K to be the average amount of sample points intersecting a candidate stop. This would mean that the amount of loops at Line 9 is reduced by a factor K and that the

loop at Line 13 is performed K times. Thus far we have $N \cdot \frac{M}{K} \cdot K$. Another loop is defined from Lines 29-31. As this is an optimization, we can to neglect its influence, as it has roughly the same effect as the loop at Line 13. We can conclude that the complexity of SMoT is $O(N \cdot \frac{M}{K} \cdot K)$ or reduced, $O(N \cdot M)$.

CB-SMoT To determine the complexity of CB-SMoT, We define variables for which we use the same literals as before to make it possible to compare both complexities. Consider the variables N, the number of trajectories and M, the average amount of sample points in a trajectory. As a basis to determine the complexity, we use the pseudo-code in Listing 2.2. We use a different method than previous complexity calculation, as we build up from the lower levels up to the higher levels, to determine the overall complexity. Consider the Lines 15-20, this has complexity $O(M \cdot (M+M))$, as the amount of neighbors, used by the for-loop at Line 15, will be of the order of the amount of items of the trajectory, which defines the upper limit of the number of neighbors, and both Lines 16 and 19 evaluate every point in the set of neighbors. Lines 12-22 have a complexity of $M \cdot (M + M \cdot (M + M))$, as Line 13 will evaluate all points in t to determine their distance to p. Together with Lines 15-20, they are performed for every point $p \in t$, which is M times.
Lines 24-29 have a complexity of $M \cdot M$, as for each point in cluster C has to be evaluated whether it intersects a candidate stop, taking an order of M steps, as M defines the upper limit of the number of points in C. We presume a spatial index is defined on the candidate stops to allow retrieval in constant time. If a point intersects a candidate stop, the subtrajectory of this intersection is generated, taking another M steps, as the maximum number of points intersecting a candidate stop is M, hence the complexity $M \cdot M$.
Lines 30-35 have a complexity of M as every subtrajectory is evaluated, and we consider the amount of subtrajectories to be an order of M, as M defines its upper limit.
The complexity of Lines 36-40 is M, as we also evaluate every subtrajectory. For the Lines 23-41 this totals to a complexity of $O(M \cdot ((M \cdot M) + M + M))$ as the Lines 24-40 are performed M times, as we consider clusters to be of order M.
As the Lines 10-41 are performed N times, we can conclude the complexity to be $O(N \cdot ((M \cdot (M + M \cdot (M + M))) + (M \cdot ((M \cdot M) + M + M))))$ or reduced, $O(N \cdot M^3)$. If we would compare this with the complexity of SMoT, which is $O(N \cdot M)$, we can conclude that SMoT needs fewer computations than CB-SMoT.

Conclusion

In this thesis, we focus on semantically enriched trajectories. In essence, it is of no importance how these are generated, as it is mainly the type of data that concerns us. However, in the implementation we prefer to use SMoT, as we do not need the features provided by CB-SMoT. So, any future references to enrichment, will refer to SMoT.

Chapter 3

Mining sequential data

In Chapter 2, we have introduced our sequential data, namely a trajectory stored as trajectory samples, which is transformed into a semantically enriched format, so every trajectory eventually is represented as a list of stops and a list of moves. This model is interesting, as it emphasizes the semantics of the trajectories.

As we wish to use this data as the input of our mining process, our semantically enriched trajectories need to be transformed into a usable data model. The output list of stops is formatted into a sequence, which essentially is an ordered list of itemsets, in this case composed of stops. The advantage of considering trajectories as sequences, is that by doing so it allows us to introducing previous research, as this type of data has been the subject of previous studies, where sequences represented DNA sequences, purchase history, event based data concerning security systems or web page history. The order defining these sequences is often of a temporal nature, as in trajectories, but a spatial ordering is also possible, as in DNA sequences.

Thus, trajectories are transformed into sequences, which can be used for our mining process. However, as we have mentioned in Chapter 1, most of the developed mining techniques are traditionally applied on itemsets, e.g., shopping baskets, where a ordinal relation is of no importance, or non-existent. Therefore the existing techniques need to be adapted, something many studies have focused on in the past. In this chapter we use this previous research, together with the principles behind trajectories to define mining techniques allowing us to find frequent sequences and consequently mobility patterns.

Before we focus on our own contribution, mining mobility data, we will first introduce important concepts concerning association analysis and in particular the Apriori algorithm.

3.1 Association analysis

3.1.1 Introduction

Association analysis, as explained in Section 1.1.2, is a group of algorithms used to analyze large amounts of transactions stored in databases with the purpose of finding frequent itemsets or association rules [HK00, TSK05].

As we wish to explain the notion of association analysis in detail, we need to introduce a few concepts. When $I = \{i_1, i_2, ..., i_d\}$ is the set of all items found in the transactions and $T = \{t_1, t_2, ..., t_n\}$ is the set of all transactions, then every transaction can be represented as a non-empty subset of I, called an itemset. One itemset I_k can contain another itemset I_l, meaning that every item present in I_l, is also present in I_k. Finally, every itemset X is associated with a support count σ with respect to a collection transactions, indicating the number of transactions containing this itemset, or more formally:

$$\sigma(X) = |\{t_i | X \subseteq t_i, t_i \in T\}|$$

Support count σ is used to recognize frequent itemsets, which are itemsets that have a support count that exceeds the minimum support count. Once the frequent itemsets are found, they form the basis to formulate rules, the eventual goal of the analysis.

Table 3.1: Example of set of transactions represented as itemsets.

T_1	$\{Salt, Bread, Pepper\}$
T_2	$\{Bread, Pepper\}$
T_3	$\{Salt, Pepper, Lemons, Milk\}$
T_4	$\{Pepper, Lemons, Milk\}$
T_5	$\{Salt, Pepper, Milk\}$

A rule is always of the form $A \rightarrow B(sup = x, con = y)$, where A and B are disjoint itemsets. Such a rule states that if a transaction contains all items present in itemset A, this transaction will also include all items in itemset B. For example:

$$\{Salt, Pepper\} \rightarrow \{Milk\}$$

states that if a transaction contains the items 'Salt' and 'Pepper', it will also contain the item 'Milk'. Every rule has a support *sup* and confidence *con*, respectively expressing the interestingness and reliability of a rule. For a rule to be of interest, it preferably applies to a large portion of our analyzed itemsets. To express this interestingness, support measures the percentage of the transactions where both A and B are present, or more formal:

$$support(A \rightarrow B) = \frac{\sigma(A \cup B)}{N}$$

where N is the number of transactions used for analysis. On the other hand, for a rule to be reliable, the implication expressed by our rule should be valid for as many transactions as possible. Confidence expresses this reliability by measuring the percentage of transactions containing the itemset A that also contains B, or more formal:

$$confidence(A \rightarrow B) = \frac{\sigma(A \cup B)}{\sigma(A)}.$$

Support and confidence are two important tools for the user in the search for rules describing patterns in the collection transactions, since association analysis uses a minimum support and confidence, formulated by the user, to collect all rules that are of interest to him. The resulting set of rules will only include rules that fall within this set of minima of support and confidence.

As an example of the introduced notions and concepts, consider the itemsets representing transactions T depicted in Table 3.1. To give examples of the notions mentioned in the previous paragraph, we can conclude that $I = \{Salt, Bread, Pepper, Lemons, Milk\}$ is the set of items. The support count of the itemset $\{Lemons, Milk\}$ equals 2, as it is the subset of two transactions. A support count results in a support of 40%, as this is calculated by dividing the support count, which equals 2, by the number of transactions, which equals 5. In case the minimum support is set to 40% or less, this itemset would be considered frequent.

Consider the rule $\{Salt, Pepper\} \rightarrow \{Milk\}$. It has a support of 40%, as two transactions contain 'Salt', 'Pepper' and 'Milk' and a confidence of 66%, as 2 transactions contain 'Salt', 'Pepper' and 'Milk', while 3 contain 'Salt' and 'Pepper'.

3.1.2 Apriori

Association analysis concentrates on finding frequent itemsets, which are used to generate association rules. One naive possibility to find these frequent itemsets, is to generate all possible candidates and determine for each whether it is frequent. By comparing the candidate to each transaction to check whether all items contained in the candidate are present in the transaction, we can establish the support count of the candidate. If the support count is larger than what the user defined to be the minimum support, the candidate is considered frequent. However, the drawback of generating all possible candidates is that up to $2^k - 1$ itemsets, with k being the number of distinct items present in the transactions, need to be generated, requiring a large amount of computations and comparisons. However, the Apriori algorithm [AIS93] tries to avoid generating these candidates by applying the Apriori Principle.

Apriori Principle *If an itemset is frequent, then all of its subsets must also be frequent.* □

The idea behind this principle enables us to reduce the amount of candidates and thus also the amount of comparisons. It states that if a candidate itemset $\{A, B, C\}$ is frequent, $\{A, B\}$, $\{B, C\}$ and $\{A, C\}$ should be frequent too. If $\{A, B\}$ and $\{B, C\}$ have a support of 10% and $\{A, C\}$ has a support of 5%, it impossible for $\{A, B, C\}$ to have a support higher than 5%.

This principle is an example of an anti-monotone property, a group of properties that state that if a set does not pass a test, all of its supersets will also fail this test. It is this anti-monotonicity that gives the apriori algorithm its strength.

To demonstrate how the Apriori algorithm uses this property, a step-by-step demonstration will be given using the given pseudo-code presented in Listing 3.1 as a guideline. First observe the input of the algorithm, a set of transactions T, where each transaction is represented as an itemset, and *minsup* the minimum support that is expected of the outputted frequent itemsets. The first step (Line 9) is generating all frequent itemsets of size one, meaning all frequent itemsets that consist out of a single item. To determine these itemsets, an itemset is generated for every possible item that is then checked whether it is frequent or not by calculating its support count. If this support count is greater than the minimum support, it is added

to the set F_1, otherwise it is discarded. F_1 will form the basis of our next step where we try to determine the frequent itemsets of size 2.

First, the function $apriori\_gen(F_{k-1})$ composes the set of candidate itemsets C_2 using the set F_1 (Line 12). Candidates are composed my combining every itemset in F_1 with an itemset of the same collection. Then, for each transaction t in T is determined which of the candidates in C_2 are subsets of t by using the function $subset(C_k, t)$ (Line 14). For each of the candidates that are a subset of t, the support count is calculated. After all transactions have been checked, and the support of every candidate has been determined, it is decided which of the candidates have the set minimum support. These frequent 2-itemsets are added to the set of outputted frequent itemsets (Line 19).

In the next phase we use the frequent 2-itemsets to determine those of size three (Line 10). This process goes on until no frequent itemset can be added to the outputted set of frequent itemsets. The output of our algorithm is the union of all sets of frequent itemsets determined over the course of its execution.

3.2 Related Work

3.2.1 Sequential data

Having defined the idea behind Apriori, it would be interesting if we could apply this to semantically enriched trajectories, while respecting and using their ordinal relation. As mentioned in the introduction, one step in the direction of reaching this goal is to model our mobility data into sequences. At this point we need to clearly define this model.

Traditionally, when sequences are observed in literature, as described in [AS95], they are defined as follows:

Definition 8. An *itemset* is a non-empty set of items. A *sequence* is an ordered list of itemsets. We denote an itemset i by $(i_1 i_2 \dots i_m)$ where i_j is an item. We denote a sequence s by $\langle s_1 s_2 \dots s_n \rangle$, where s_j is an itemset. □

An example of a sequence representing video rental history could be ⟨(Gladiator, The Shining) (Good Bye Lenin) (Stardust, Lucky Number Slevin)⟩. This example represents a video rental sequence, and can be interpreted as

Listing 3.1: The pseudo-code of the original Apriori presented in [TSK05]

```
1  Input:  T //set of transactions
2          minsup //minimal support for frequent itemsets
3
4  Output: R //set of frequent itemsets
5
6  Method:
7  k = 1;
8  N =number of transactions in T;
9  F_k = {i | i ∈ I ∧ σ({i})/N ≥ minsup};
10 repeat
11    k = k + 1;
12    C_k = apriori_gen(F_{k-1});
13    for each transaction t ∈ T do
14       C_t = subset(C_k, t);
15       for each candidate itemset c ∈ C_t do
16          σ(c) = σ(c) + 1;
17       endfor
18    endfor
19    F_k = {c | c ∈ C_k ∧ σ(c)/N ≥ minsup};
20 until F_k = ∅
21 R = ⋃ F_k;
```

the rental history of a person who first rented *Gladiator* and *The Shining* together, after which s/he rented *Good Bye Lenin*, and finally *Stardust* and *Lucky Number Slevin*, again at the same time.

Previously we presented the notion of a subset, but a similar concept also exists in sequential data, namely subsequences. Traditionally a sequence is defined as follows:

Definition 9. $\langle a_1 a_2 \dots a_n \rangle$, is a *subsequence* of sequence $\langle b_1 b_2 \dots b_m \rangle$, with $a_1, \dots, a_n, b_1, \dots, b_m$ being itemsets, if there exist integers $i_1 < i_2 < \dots < i_n$ such that $a_1 \subseteq b_{i_1}$, $a_2 \subseteq b_{i_2}$, $\dots$, $a_n \subseteq b_{i_n}$. □

3.2.2 AprioriAll

Applying an Apriori-inspired algorithm on sequential data has already been investigated, resulting in the AprioriAll [AS95] and the GSP [SA96] algorithms. AprioriAll uses the same structure provided in the Apriori algorithm, as the pseudo-code in Listing 3.2 demonstrates. But as it concentrates on sequences, it differs on two particular points from the original algorithm, namely candidate generation and support count measuring.

Before any analysis can be performed, a transformation of the input sequences is needed. Consider a sequence, a list of itemsets possibly representing transactions. The transformation will replace every itemset by the set of frequent itemsets it contains. If by doing so transactions become empty, these transactions are removed. If a sequence becomes empty, this is also removed from the input. Every itemset is also represented by a literal, a unique identifier representing every possible composition of an itemset. For example, consider the sequence $\langle \{A, B\}\{B, C\}\}$, assuming all items in this sequence are frequent, this would be transformed into, $\langle \{1, 2, 3\}\{2, 4, 5\}\rangle$, with 1 representing $\{A\}$, 2 $\{B\}$, 3 $\{A, B\}$, 4 $\{C\}$ and 5 $\{B, C\}$. However, in following examples, we will not use this transformed format, as this makes some concepts more difficult to understand.

The first phase we should concentrate on, is candidate generation, a phase performed by the function $apriori - gen(F_{k-1})$. In this phase the candidates of size k are generated using the frequent sequences of size $k - 1$. Here, the size of a sequence is interpreted as the amount of items making up the sequence, regardless of the amount of itemsets containing the items. For example, $\langle \{A\}\{B\}\rangle$ and $\langle \{A\}\{C\}\rangle$ are both sequences of size 3. Combining these two in a sequence of size 4 would result in the sequences $\langle \{A\}\{B\}\{C\}\rangle$

Listing 3.2: Pseudo-code AprioriAll

```
1  Input:  T //set of trajectories
2            minsup //minimal support for frequent itemsets
3
4  Output: R //set of frequent sequences
5
6  Method:
7  k = 1;
8  N =number of sequences in T;
9  F_k = {i | i ∈ I ∧ σ({i})/N ≥ minsup};
10 repeat
11    k = k + 1;
12    C_k = apriori − gen(F_{k−1});
13    for each data sequence t ∈ T do
14       C_t = subsequence(C_k, t);
15       for each candidate k−subsequences c ∈ C_t do
16          σ(c) = σ(c) + 1;
17       endfor
18    endfor
19    F_k = {c | c ∈ C_k ∧ σ(c)/N ≥ minsup};
20 until F_k = ∅
21 R = ⋃ F_k;
```

and $\langle\{A\}\{C\}\{B\}\rangle$. Candidates are generated by matching the first $k-2$ items, if these match, a candidate in generated by adding the last itemset of the second sequence, to the first sequence.

Once all candidates are generated, some have to be pruned, as they do not comply with the Apriori Principle. To check whether a candidate of size k complies with the Apriori Principle, we remove each item once and check if the sequence consisting out of the remaining items belongs to the frequent sequences of size $k-1$. Consider our generated candidate $\langle\{A\}\{B\}\{A\}\{B\}\{C\}\rangle$, if we want it to comply with the Apriori Principle, all subsequences of size 4 should be frequent, being $\langle\{B\}\{A\}\{B\}\{C\}\rangle$, $\langle\{A\}\{A\}\{B\}\{C\}\rangle$, $\langle\{A\}\{B\}\{B\}\{C\}\rangle$, $\langle\{A\}\{B\}\{A\}\{C\}\rangle$ and $\langle\{A\}\{B\}\{A\}\{B\}\rangle$. If one of these would prove not to be frequent, the candidate is discarded.

The second important phase is the support count measuring. Here, the function $subsequence(C_k, t)$ is used to determine the candidates that are contained in a trajectory, after which the support count of the found candidates is increased. To find all candidate subsequences contained in a specific sequence, AprioriAll uses the method proposed in [AS95], which employs a hash-tree containing all candidates.

The hash-tree used in the subsequence matching, stores all candidate subsequences. Using this tree allows us to minimize the amount of matching needed to determine the support count of our candidates. The tree is built up from leaf nodes, containing a table of sequences, and interior nodes, containing a hash-table referring to other interior nodes or leaf nodes.

To add a candidate to the hash-tree, we start at the root, at depth 1, and go down each interior node until we reach a leaf node. To determine at depth d of the tree, which branch should be followed, we apply a hash function to the dth item of the sequence. As an interior node contains a hash-table, the result of the hash function will point at a bucket containing the next node. If we would reach a leaf node, the candidate should simply be added, unless the node has reached it maximum capacity. In this case we replace the leaf node by a new interior node, to which the new candidate is added together with all sequences previously contained in the old leaf node.

The constructed tree can then be used to determine all candidates contained in a sequence. At leaf nodes, containing a table of sequences, we

evaluate for each candidate in the table whether it is contained in the sequence. The resulting candidates will be added to the output. At interior nodes, reached by hashing an item i, we hash every item in the itemsets following the itemset containing i and apply this process recursively on the nodes contained in the buckets. At the root of the tree we hash on every item in the sequence, and access the nodes in the buckets accordingly.

3.2.3 GSP

Following the AprioriAll, another algorithm was introduced, called GSP, Generalized Sequential Patterns [SA96]. Overall, this algorithm uses the same techniques as AprioriAll, as it focuses on improving and expanding the AprioriAll principles to fulfill the expanding needs of sequential data. The most important features introduced in GSP are described in the following section.

Sliding windows The first feature introduced by the GSP algorithm is the sliding window constraint. This constraint was originally introduced to allow the mining algorithm to map an itemset that is part of a sequence on several other sequences regarded as a whole as long as the time frame of these multiple sequences fall within the limits of the stated time window. Or more formally as described in [SA96], a data sequence $d = \langle d_1, ..., d_m \rangle$ contains a sequence $s = \langle s_1, ..., s_n \rangle$ it there exist integers $l_1 \leq u_1 < l_2 \leq u_2 < ... < l_n \leq u_n$ such that

1. s_i is contained in $\cup_{k=l_i}^{u_i} d_k, 1 \leq i \leq n$, and
2. transaction-time(d_{u_i}) - transaction-time(d_{l_i}) $\leq$ window-size, $1 \leq i \leq n$

where transaction-time(d), d being an itemset or a transaction, denotes the time at which this itemset was recorded.

Time constraints The second two constraints involve the elapsed time between two consecutive stops, namely *maxspan* and *minspan*. The definition as given in [SA96], in combination with the definition of a sliding window: a data sequence $d = \langle d_1, ..., d_m \rangle$ contains a sequence $s = \langle s_1, ..., s_n \rangle$ it there exist integers $l_1 \leq u_1 < l_2 \leq u_2 < ... < l_n \leq u_n$ such that

1. s_i is contained in $\cup_{k=l_i}^{u_i} d_k, 1 \leq i \leq n$, and
2. transaction-time(d_{u_i}) - transaction-time(d_{l_i}) $\leq window - size$, $1 \leq i \leq n$

3. transaction-time(d_{l_i}) - transaction-time($d_{u_{i-1}}$) > $minspan$, $2 \leq i \leq n$

4. transaction-time(d_{u_i}) - transaction-time($d_{l_{i-1}}$) $\leq$ $maxspan$, $2 \leq i \leq n$

The first two conditions are those of the sliding window. They are mentioned here as they are related to the conditions of time constraints.

Taxonomies In this thesis, we have always regarded the items as being events unrelated to other events. However, there are applications where the items are related or comparable to each other of different levels. These relations can be expressed in a taxonomy, a is-a hierarchy featuring several levels expressing how items are related to each other. Taxonomies can be interesting as it is a multiple level approach to finding patterns.

A sequence s contains the item $i \in I$, if i is in s or i is an ancestor of some item in s.

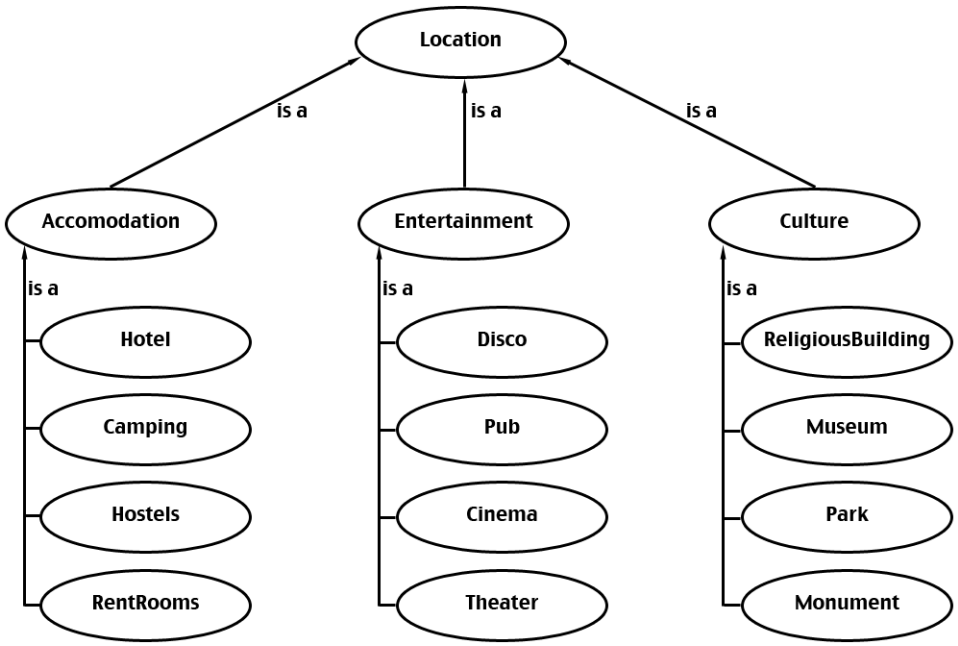

Figure 3.1: A possible hierarchy on touristic places of interest.

Mining on different levels can be achieved quite easily. By replacing an item by the item and all its upper levels, the algorithm will automatically reduce the depth of rules if this does not reach the necessary support. For example consider the sequence $\langle\{Hotel_1\}\{Bar_1\}\{Museum_1\}\rangle$, if we would like to mine on different levels of the taxonomy, we would need to transform this sequence to $\langle\{Location, Accommodation, Hotel, Hotel_1\}$ $\{Location, Entertainment, Bar, Bar_1\}$ $\{Location, Culture, Museum, Museum_1\}\rangle$. As the mining process proceeds, it is possible that lower levels are dropped during the candidate generation and support count measuring. It could be possible that the previous sequence does not get enough support, but that the sequence $\langle\{Location, Accommodation, Hotel\}$ $\{Location, Entertainment, Bar, Bar_1)$ $\{Location, Culture, Museum, Museum_1\}\rangle$ does get enough support simply because $Hotel_1$ was dropped from the first itemset.

Having studied AprioriAll and GSP, we can conclude that although we are able to find frequent sequences, these do not reflect the nature of our trajectories. Therefore we will define our own data model that we will use as a basis for our *Stops and Moves Apriori*.

3.3 Stops and Moves Apriori

3.3.1 Sequential data

If we would focus on our type of data, semantically enriched trajectories, the traditional model can be considered unnecessary complex, as the ordinal relation is defined on the stops, which are our items. If we would like to describe our data with this notion of sequences, we could consider them to be a list of singleton itemsets, where each itemset contains a stop. However, to simplify the representation, we could regard a sequence as an ordered list of items.

Because the traditional model is inadequate for representing our semantically enriched trajectories, we will define our own model. The inspiration for this model comes from the paper *Mobility Patterns* [dMR05], which proposes a language of regular expressions to describe patterns in semantically enriched trajectories. These patterns are intended for query-purposes, as they define a filter that can be applied to a collection of trajectories to retrieve a set of matching trajectories. As the concept of Mobility Patterns is designed to encapsulate the notion and semantics of trajectories, it becomes a very good basis for our own model.

The idea we would like to introduce, is a semantic model that embodies the notion of strict sequentiality, but also the notion of wildcards, as we wish to include some flexibility in our frequent sequences. Our notation is also derived from Mobility Patterns, as we use '.' to represent sequentiality in trajectories. The wildcards are represented by \$. We will now define our sequence data model.

Definition 10. Consider $S = \{s_1, s_2, \ldots, s_d\}$ to be the set of all stops found in the enriched trajectories. A sequence is a multiset of elements from S, where an ordinal relation is defined on its elements. □

An example of such a sequence could be $Museum_1.Monument_1.Hotel_1$.

We use the term multiset instead of set, as a set will only allow one membership per element, where a multiset does not have this restriction. Consider the possibility that a trajectory passes the same stop twice or more, as it returns to the same location after visiting other stops. An example of such a sequence could be

$$Hotel_1.Museum_1.Monument_1.Hotel_1$$

where $Hotel_1$ has multiple occurrences in the sequence. As with traditional sequences, our model also defines the notion of subsequences.

Definition 11. A sequence s_i of size m is a *subsequence* of sequence s_j, if $s_i[0] = s_j[n]$, $s_i[1] = s_j[n+1]$, ..., $s_i[m] = s_j[n+m]$. □

For example, both $Museum_1.Monument_1.Hotel_1$ and $Hotel_1.Museum_1$ are subsequences of $Hotel_1.Museum_1.Monument_1.Hotel_1$, while $Hotel_1.Hotel_1$ is not.

The goal of applying association analysis on sequences is to find the *frequent sequences* hidden in the trajectories. Once these are found, they can be translated into rules, describing this newfound knowledge.

Definition 12. A *sequential rule* is of the form $A \rightarrow B$ ($sup = x$, $con = y$), where A and B are two sequences, implies that if a sequence s contains the subsequence A, s also contains the subsequence B, which follows A. Every rules is associated with a support *sup* and confidence *con*. □

This is a very limited form of representing patterns, as the ordinal relation available in sequences gives us new opportunities. We will elaborate on sequential rules later in this chapter.

Listing 3.3: Pseudo-code Stops and Moves Apriori.

```
1  Input:  T //set of trajectories
2            minsup //minimal support for frequent itemsets
3
4  Output: R //set of frequent sequences
5
6  Method:
7  k = 1;
8  N =number of sequences in T;
9  F_k = {i | i ∈ I ∧ σ({i})/N ≥ minsup};
10 repeat
11    k = k + 1;
12    C_k = apriori − gen(F_{k−1});
13    for each data sequence t ∈ T do
14       C_t = subsequence(C_k, t);
15       for each candidate k−subsequences c ∈ C_t do
16          σ(c) = σ(c) + 1;
17       endfor
18    endfor
19    F_k = {c | c ∈ C_k ∧ σ(c)/N ≥ minsup};
20 until F_k = ∅
21 R = ⋃ F_k;
```

3.3.2 Description

In the next section we define our own Stops and Moves Apriori based on the principles of AprioriAll and combining these with our sequence data model. Initially we will not include the use of wildcards, as we will first focus on the differences with AprioriAll. The structure of our Apriori, as shown by the pseudo-code in Listing 3.3, will be exactly the same to AprioriAll. However, the $apriori\_gen(F_{k-1})$ and $subsequence(C_k, t)$, both essential functions, will be defined differently.

The data model we defined has an advantage, namely the fact that it resembles that of a string. To be more exact, our definition of a subsequence is similar to that of a substring. This creates possibilities of introducing string-based algorithms.

Candidate generation

During the candidate generation phase, which is performed with the help of the function $apriori\_gen(F_{k-1})$, two frequent subsequences of size $k-1$ are matched to form a subsequence of size k. First subsequences as represented as strings on which a string-matching algorithm is performed to determine if both subsequences are equal. The matching is performed linearly, by comparing $k-2$ characters.

Formally the candidate generation can be described as follows: let $A = A_1. \ldots .A_n$ and $B = B_1. \ldots .B_n$ be two subsequences of size n. If $A_2 = B_1$, ..., $A_n = B_{n-1}$ a new sequence of size $n+1$ is generated $A_1.B_1.B_2. \ldots .B_n$ and added to the collection of candidates.

For example, consider the subsequences $A.B.C$ and $B.C.D$. Using the string-matching algorithm, we check whether the second item of the first subsequence equals the first item of the second subsequence and whether the third item of the first subsequence equals the second item of the second subsequence. Since this is the case in our example, $A.B.C$ and $B.C.D$ would result in the creation of the candidate subsequence $A.B.C.D$.

A first concern is whether the Apriori Principle is respected. The Principle applies to all our generated candidates, which is a direct result of our definition of a subsequence and the principle of our candidate generation. Every candidate of size k, with $k > 1$, has only two subsequences of size $k-1$, namely those used from the collection of frequent sequences to generate the candidate. As we know that these sequences are frequent, the Apriori Principle applies to all generated candidates. Therefore there is no need for pruning, which was the case in AprioriAll and also GSP, as a subsequence was defined less strict.

The Apriori Principle holds, but was there an increase in efficiency because of the adaptations? Well, because pruning is no longer necessary, this step is eliminated resulting in a decrease in needed computations. The first step - and only in case of the Stops and Moves Apriori - is practically identical as for each pair of frequent sequences of size $n-1$ a n-sized candidate is generated in linear time. So the improvement in complexity is that of the pruning phase.

To measure a potential increase in efficiency, the complexity of our candidate generation should be compared to that of the AprioriAll. AprioriAll generates candidates of size k by comparing the first $k-2$ itemsets of the frequent sequences of size $k-1$. If these match, a k-sized candidate is generated by taking the sequence of size $k-2$ that occurs in both frequent sequences, to which both $(k-1)$th items are added. Consider N to be the number of frequent sequences of size $k-1$, then the complexity of generating the candidates can be described as $N \cdot N \cdot (k-2)$, as the collection of frequent sequences is read twice, and $k-2$ comparisons are needed. This complexity corresponds with that of Stops and Moves Apriori, as the candidates are generated in similar fashion.

However, the fact we no longer need to prune candidates containing infrequent subsequences does result in a reduction of needed computations. To determine the complexity of the pruning phase, we use the variables N to be the number of frequent sequences of size $k-1$, and C to be the number of candidates of size k. The complexity can then be described as $O(C \cdot k \cdot N \cdot (k-1))$, which can be reduced to $O(M \cdot N \cdot k^2)$ as for each candidate we check if the sequence generated by removing one of the k itemsets, is a frequent sequence of size $k-1$.

We can thus conclude that generating the candidates has a similar complexity in Stops and Moves Apriori and AprioriAll, but that pruning of infrequent candidates, which is only necessary in AprioriAll, requires $O(M \cdot N \cdot k^2)$ computations.

Support count measuring

During this phase, we determine for each sequence which candidates it contains as to measure the support count of each candidate. We use the function $subsequence(C_k, t)$, where C_k is the collection of candidates and t is the sequence representing a trajectory. As our focus is mainly on speed, we should concentrate on two important elements. First, the amount of matching should be limited, and second, the matching should be performed as efficient as possible.

To limit the amount of matching to be performed, we can use the same technique used by the AprioriAll, namely a hash-tree containing all candidates, allowing us the retrieve a small portion of the candidates that are then matched to the sequence.

The hash-tree has two types of nodes, leaf nodes, containing a table of candidate subsequences, and interior nodes, containing a hash-table where every bucket leads to a node at lower level. The root is defined to be at depth 1. To add a candidate subsequence to the tree, we start at the root and apply a hash function to its first item. If the node the bucket leads to is a leaf node that has not reached maximum capacity, the subsequence is added to the table, if it has reached maximum capacity, this leaf node is replaced by a interior node and the new subsequence, together with the subsequences from the old leaf node are added to the child nodes of the new interior node using the hash function. If however the node is a interior node, we apply the hash function to the next item, which we repeat recursively until we reach a leaf node. Generally, we can conclude that in a node at depth d we hash the dth item to reach the node at depth $d + 1$ until we reach a leaf node.

The next step is to use the tree to determine which subsequences are contained in a sequence. There is a different approach depending on the type of node we have reached. If the node is a leaf node, we match every subsequence in the table to the sequence, and add all matching subsequences to the result. If the node is an interior node that we reached by hashing the ith item, we apply the hash function on every item following the ith item, and move down every selected branch. By recursively applying the hash function, we might eventually reach a leaf node. At the root we apply the hash function on every item in the sequence, and follow each selected branch.

We have limited the amount of matching needed to determine the subsequences contained in a sequence, but as mentioned, there are still a few comparisons to be performed. During the candidate generation, we had to match two frequent sequences of size $i - 1$ to generate a candidate of size i. However, here we had fixed positions for comparisons, something we no longer have. That is why we need a different approach.

We already stated that the analogy of our sequences to strings might allow us to use string-based techniques. From this point of view, there was one interesting algorithm used for substring matching, namely the Boyer-Moore algorithm, which could prove a fast way to match whether a candidate subsequence is contained in a sequence.

Boyer-Moore

The Boyer-Moore string search algorithm was first proposed in [BM77]. This algorithm was developed to speed up the search of a substring or pat-

tern in a string by avoiding checking every character of the string. Instead of matching the first character of the pattern with the first character of the string, it starts by matching the last character of the pattern with the corresponding character in the string. As Boyer-Moore is intended to be used on strings instead of sequences, a translation has to occur prior to support count measuring, as the candidates and the sequences need to be formatted into strings.

To demonstrate how the Boyer-Moore string search works, three observations can be made. They form the basis of the algorithm and encapsulate the idea behind it.

However, before the observations can be described, a few of the variables used need to be introduced together with their purpose.

- *pattern_length*: the length of the pattern that is searched for in a string.
- $pattern(j)$: the j-th character in the pattern
- $delta_1$ and $delta_2$: two tables used to determine the cursor jumps during the matching process. How these are calculated will be explained later.

Observation 1 If the character in the string that is being focused on, later referred to as c, does not appear in the pattern, the first characters, of which the amount corresponds to the length of the pattern, referred to as *pattern_length*, do not need to be checked. It is impossible that the pattern will occur within these characters. As a consequence, we can move our cursor *pattern_length* positions, and focus on the next *pattern_length* characters of the string. For example:

```
Pattern:    E.C.A.G.E.C
Sequence:   C.A.A.B.C.D.G.F.E.C.E.G.A.C.E.C.A.G.E.C
```

As `D` does not occur in the pattern, there is no need to check the characters that precede `D`, since it is impossible that pattern can match. Therefore we can concentrate on the next six characters.

```
Pattern:                E.C.A.G.E.C
Sequence:   C.A.A.B.C.D.G.F.E.C.E.G.A.C.E.C.A.G.E.C
```

Observation 2 If the character in the string does appear in the pattern, with its position being *delta* (counting from the right side), we can move our cursor *delta* position, as to align the corresponding characters of both the string and the pattern. For example:

```
Pattern:                E.C.A.G.E.C
Sequence:   C.A.A.B.C.D.G.F.E.C.E.G.A.C.E.C.A.G.E.C
```

As `G` does occur in the pattern, we cannot move our cursor *pattern_length* positions. We have to match the positions of the `C` in the pattern and the string by moving the pattern two positions.

```
Pattern:                      E.C.A.G.E.C
Sequence:   C.A.A.B.C.D.G.F.E.C.E.G.A.C.E.C.A.G.E.C
```

Observation 3a If we have already matched a part of the pattern to the string and encounter a mismatch, we can move our cursor $m + k$ characters to the right, where m equals the number of characters of the string that were already matched and k is either 1, if the mismatched does not occur in the pattern or the mismatched character's rightmost occurrence in the pattern is in the part of the patterns that has already been matched, or if it is in the unmatched part of the pattern, k equals the rightmost position of the mismatched character minus m, the amount of characters that were already checked. By doing so, we align these two equal characters.

```
Pattern:                      E.C.A.G.E.C
Sequence:   C.A.A.B.C.D.G.F.E.C.E.G.A.C.E.C.A.G.E.C
```

`A` does not match the `E` from the pattern, so we can move our cursor and pattern one position as we have already matched 1 character, plus 1 extra position, as the rightmost occurrence of `E` is to the left of `A`. So we move our pattern two positions.

```
Pattern:                          E.C.A.G.E.C
Sequence:   C.A.A.B.C.D.G.F.E.C.E.G.A.C.E.C.A.G.E.C
```

Observation 3b More general, if some characters of the pattern were matched onto the string, and a mismatch occurs, it is possible that this part of our pattern, this subpattern, recurs in the pattern, referred to as a 'plausible recurrence'. Therefore by moving our pattern as to align the

rightmost recurrence of the subpattern in the pattern, left of our original subpattern, with the part of the string that was matched, some comparisons can be avoided. After shifting the pattern, the cursor also needs shifting to the right.

```
Pattern:                          E.C.A.G.E.C
Sequence:   C.A.A.B.C.D.G.F.E.C.E.G.A.C.E.C.A.G.E.C
```

We are able to match `E.C` when a mismatch occurs, as `G` and `C` don't match. By shifting the pattern 4 positions we align `E.C` with the second occurrence of this subpattern.

```
Pattern:                                  E.C.A.G.E.C
Sequence:   C.A.A.B.C.D.G.F.E.C.E.G.A.C.E.C.A.G.E.C
```

Algorithm The notation $pattern(j)$ will refer to the j^{th} character of the pattern. Two tables are needed for the algorithm, $delta_1$ and $delta_2$. The size of $delta_1$ equals the number of characters in the alphabet. As observation 1 and observation 2 indicated, if a character does not occur in the pattern or it occurs in the pattern but there is a mismatch, we are able to shift the focus a number of characters to the right. That is why we need $delta_1$. For every character it contains an integer, if the character does not occur in the pattern, this integer equals the length of the pattern. However, if it does occur, this integer equals the length of the pattern minus j, j being the maximum possible shift so the characters in the pattern and string are equal. A single item in $delta_1$ is referred to as $delta_1(char)$.

The table $delta_1$ can be constructed as follows: create a table that contains an entry for every possible character. Initialize each entry to $pattern\_length$. As we process every character c of the pattern, we replace the entry with $pattern\_length$ - position of c, replacing the entry where needed.

$delta_2$ also contains integers, namely for every character in the pattern, so $delta_2(j)$ refers to the pattern's character at position j. If it was able to match a certain part of the pattern to the string, but there was a mismatch, the pattern and also the focus need to be shifted. When a part of the pattern recurs within the pattern, and that part has already been matched, as demonstrated in observation 3a and 3b, the recurring part of the pattern and the corresponding part of the string can be positioned opposite of each other to ensure they already match. This shifting is calculated in advance

and inserted in $delta_2$. Each integer is the sum of the shift of our pattern and the shift of our focus.

These two tables are used during the process of the algorithm to calculate the maximal shift of our pattern and focus, and to minimize the needed time for our pattern search. The Tables 3.2 and 3.3 give example of $delta_1$ and $delta_2$ calculated for the previously explained example.

Table 3.2: $delta_1$ as calculated for the example.

A	3
B	6
C	0
D	6
E	1
F	6
G	2

Table 3.3: $delta_2$ as calculated for the example.

A	0
C	4
E	0
G	4

The algorithm starts by preprocessing *pattern* and inserting the corresponding data in $delta_1$ and $delta_2$. These tables are later used during pattern matching, as it allows it to calculate the shifts of the pattern and focus. After having calculated the two tables, the pattern and string are given as input. The pseudo-code demonstrates the progress of the pattern search process.

The algorithm either returns -1 or a positive integer. The -1 indicates that the pattern was not found in the string, while the positive integer indicates the pattern was found and represents the leftmost location of the pattern in the string. Therefore, if the output is different from -1, the support count of the pattern can be increased. Naturally, the output could also be a boolean, as we don't need the leftmost location of the pattern, only confirmation that the pattern occurs in the sequence.

Listing 3.4: Boyer-Moore string search algorithm.

```
1  Input: string //string to be searched for matching pattern
2             pattern //pattern to be searched in string
3
4  Output: output //integer
5        //if output = -1, no match was found
6        //if output > 0, match was successful
7
8  Method:
9  stringlength = length of string;
10 patternlength = length of pattern;
11 output = -1;
12 i = patternlength;
13 stop = false;
14 while i ≤ stringlength and stop = false
15   j = patternlength;
16   continue = true;
17   while continue and not stop
18     if j = 0
19       stop = true;
20       output = j + 1;
21     else
22       if string(i) = pattern(i)
23         j = j - 1;
24         i = i - 1;
25       else
26         i = i + max(delta1(string(i)), delta2(j));
27         continue = false;
28       endif
29     endif
30   endwhile
31 endwhile
32 return output;
```

After having introduced Boyer-Moore as a matching algorithm to use in our support count measuring, does this have an influence on the amount of computations needed to perform our mining? To answer that question, we can compare the complexity of the support count measuring of AprioriAll and Stops and Moves Apriori.

One important part of the computation is building a hash-tree and retrieving all candidates. However, as we use the same technique in both AprioriAll and Stops and Moves Apriori, we choose not to elaborate on this, as the complexity would be the same in both algorithms.

However, once the collection of k-sized candidates has been determined, we have to match these with our sequences to determine their support count. AprioriAll uses a matching algorithm with takes $O(k \cdot M)$ computations, as each of the k items in the candidate needs a maximum of M comparisons, with M being the average length of a sequence, to find before a match can be made. As Stops and Moves Apriori uses the Boyer-Moore algorithm to match, matching these only takes $O(\frac{M}{k})$ computations, which is an improvement. If we would consider C to be the total number of candidates, than the complexity of the matching phase in Stops and Moves Apriori becomes $O(N \cdot C \cdot \frac{M}{k})$ compared to the $O(N \cdot C \cdot k \cdot M)$ of AprioriAll, where C, the total number of candidates of size k gives an upper limit to the amount of candidates retrieved from the hash-tree.

After having compared AprioriAll and Stops and Moves Apriori, we can conclude that the most important advantages of Stops and Moves Apriori is that pruning is no longer needed, and the use of Boyer-Moore, both reducing the amount of needed computations.

However, do these improvements come at a price? The most significant drawback is the limited amount of frequent sequences found as opposed to AprioriAll, which is a direct consequence of our definition of a sequence. By defining a subsequence so strictly, we leave a large portion of potential patterns out. However, in the following section we try to solve this by introducing wildcards as a way to a more flexible approach to subsequences and thus also frequent sequences.

3.3.3 Wildcards

We choose a two-step approach to introducing and defining our notion of wildcards. We start with single-matching wildcards and expand this definition to multi-matching wildcards in an attempt to make a more transparent transition.

Single-matching wildcards

Stops and Moves Apriori is already able to find frequent sequences, but as mentioned, sometimes these results are not satisfactory. It could be interesting to widen the class of outputted patterns, and lift some of the restrictions we are faced with at this point. Some of these restrictions are a direct result of our definition of sequences, primarily of the way we defined a subsequence.

For an example of these drawbacks and restrictions, observe the three trajectories shown in Figure 3.2. Translated in textual form, these could become

1. $Hotel_1.Museum_1.Bar_1.Museum_2$
2. $Hotel_2.Museum_1.Bar_2.Museum_2$
3. $Hotel_3.Museum_1.Bar_1.Museum_2$

If a minimum support of 70% is set, the collection of found frequent sequences would be limited to $\{\langle Museum_1\rangle, \langle Museum_2\rangle\}$.

But as we can observe from the graphical representation, there is one very significant pattern that was not found, namely the fact that when an object visits $Museum_1$ it will also visit $Museum_2$. We will now try to find a way to find this class of frequent sequences.

What measures can be made to allow us to mine for this type of pattern? One option originates from the ideas, proposed in the paper [GRS99], to *increase the user-controlled focus in the pattern mining process* by using regular expressions as to limit uninteresting result, and the principles behind Mobility Patterns [dMR05], which use a form of regular expressions adapted to trajectories as a querying tool. The regular expressions are a tool provided to the user as to constraint and direct the nature and the amount of results generated by the mining or querying process.

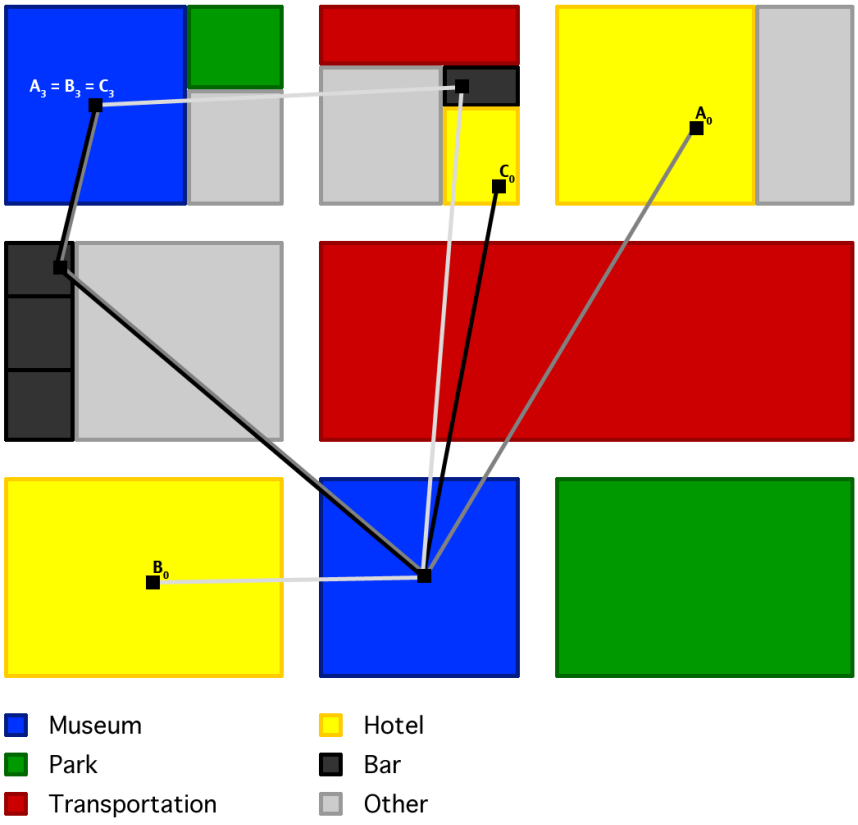

Figure 3.2: Visual representation of example of patterns ignored without the use of single-match wildcards.

However, our intention is not to constraint the results, but to allow more flexibility while mining for patterns. The use of regular expressions did inspire us to use wildcards while generating candidate sequences. By using these, we gain the advantages provided by regular expressions, as it is a simple and comprehensible way of expressing the flexibility we want in our candidate sequences.

Definition 13. Consider $S = \{s_1, s_2, \ldots, s_d\}$ to be the set of all stops found in the enriched trajectories and \$ to be a *single-matching wildcard*, which can represent any stop $s \in S$. A sequence using single-matching wildcards is a sequence composed of elements of $S \cup \{\$\}$. □

This wildcard makes it possible to find patterns that could be interesting to the user, for example of the form

$$A.\$.C.D,$$

implying a frequent sequence that goes from A through any possible stop, after which it goes through C and finally through D.

This expansion can be achieved by two simple measures. First, when generating the collection of frequent item sequences of size 1, the wildcard \$ is added. In further steps of candidate generation this will be treated like a regular item. Secondly during the evaluation whether the candidates are frequent or not, \$ is treated as a wildcard that could represent any item.

One issue that arises from introducing this wildcard is whether the Apriori Principle is still respected. In a sense, this is still the case, if the notion behind the wildcards is taken into account. Consider the candidate sequence $A.\$.C$. If we would like the Apriori Principle to be respected for this sequence, then following sequences need to be frequent: A, \$, C, $A.\$$ and $\$.C$. We know that $A.\$$ and $\$.C$ are frequent, otherwise the candidate would not have been generated, and this also applies to A, \$, C and $A.\$$. But it could be possible that the sequence $B.C$, which could be regarded a subsequence of $A.\$.C$, is not frequent, as thus we could conclude that the Apriori Principle does not hold. However, when determining whether the subsequences of a candidate are frequent, the wildcards in these subsequences should be regarded as such. The support count of this subsequence should be the sum of the support counts of the subsequences where the wildcard has been replaced by all possible items. As a consequence, we can conclude that the Apriori Principle still holds.

To demonstrate this interpretation, consider the collection of sequences S depicted in Table 3.4. The frequent sequences of size 2, with a support of 50% would be $F_2 = \{A.B, B.C, C.D, D.E, A.\$, B.\$, C.\$, D.\$, \$.B, \$.C, \$.D, \$.E, \$.\$\}$. One of the generated candidates would be $A.B.\$$, which is generated by combining $A.B$, and $B.\$$. By respecting the nature of the wildcard, we can conclude that they both are frequent with a support count of respectively 2 and 3.

The introduction of wildcards in candidate subsequences makes Boyer-Moore difficult to use for pattern matching, since the two tables used to minimize the number of comparisons cannot be generated properly when

Table 3.4: Example of set of sequences.

S_1	$\{A.B.C.D\}$
S_2	$\{B.C.D.E\}$
S_3	$\{A.B.D.E\}$

using our wildcard. Instead we take our inspiration from resolving regular expressions, as the introduction of wildcards originates from the notion of Mobility Patterns, which is a form or regular expressions.

Multi-matching wildcards

Going a step further, we can introduce multi-matching wildcards. The single-matching wildcards described in the previous section can represent any item, but it is restricted to one single item. However, this restriction leaves out an interesting class of patterns. For example, observe the sequences $A.B.C.F$, $A.B.C.D.F$ and $A.B.C.D.E.F$. There is clearly a pattern visible amongst the sequences namely all three sequences start with $A.B.C$ and end with F. As for a certain application this type of pattern might be considered interesting, these patterns should be recognized and outputted. Modifying our Stops and Moves Apriori to be able to recognize frequent sequences containing multi-matching wildcards is the next step in our process.

Definition 14. Consider $S = \{s_1, s_2, \ldots, s_d\}$ to be the set of all stops found in the enriched trajectories and $\$^+$ to be a *multi-matching wildcard*, which can represent one or more stops $s \in S$. A sequence using multi-matching wildcards is a sequence composed of elements of $S \cup \{\$^+\}$. □

The adaptations needed, are mainly situated in the support count phase, as candidate generation is identical to that proposed with single-matching wildcards. During the evaluation of the support count, the algorithm checks whether a candidate frequent sequence can be mapped on a data sequence. It is this mapping that should be adapted in a way to allow for the multi-matching wildcards.

If we would like to match a candidate regular sequence onto a sequence, there is an approach that allows us to recursively resolve whether these two match. Consider a matching-function $find(candidate, sequence)$, which matches the candidate onto the sequence by comparing the first item of the candidate with the first item of the sequence, if they match, we compare the

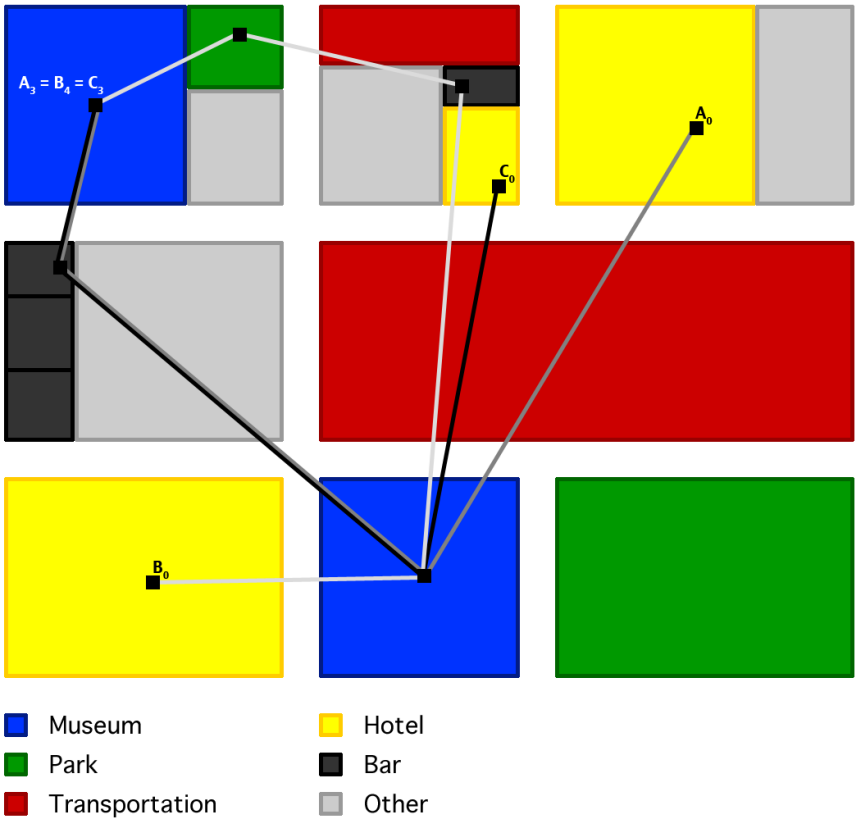

Figure 3.3: Visual representation of example of patterns ignored without the use of multi-match wildcards.

second item of the candidate with the second item of the sequence, and so on, until all items have been matched. If we would encounter a mismatch, we move the cursor of the sequence 1 position, so we would compare the first item of the candidate with the second item of the sequence. When we encounter a wildcard in the candidate, we recursively call the matching-function, where *candidate* is the remainder of the candidate after the wildcard, and *sequence* is the remainder of the sequence after the item that was mapped onto the wildcard.

For example, consider the candidate frequent sequence $B.\$^{+}.E.\$^{+}.G$ and the sequence $A.B.C.D.E.F.G.H$. Matching this candidate onto the sequence

would start by comparing B with A. As this fails, the cursor of the sequence is moved, resulting in a successful match between B of the pattern and B of the sequence. Next we encounter $\$^+$ in the candidate, resulting in a recursive call, where $E.\$^+.G$ is matched onto $D.E.F.G.H$. As E and does not match onto D, E and E are matched. Again a $\$^+$ is encountered, so G is matched onto $G.H$. As this candidate is a subsequence of $G.H$, it can be concluded that $B.\$^+.E.\$^+.G$ matches onto $A.B.C.D.E.F.G.H$.

Again, we can ask the question whether the Apriori Principle is respected during the candidate generation, but as the candidate generation is based on exact the same principles as with single-matching wildcards.

Will the results of Stops and Moves Apriori with the use of multi-matching wildcards differ from those of AprioriAll? As mentioned before, AprioriAll already enables us to mine semantically enriched trajectories, where we would describe each stop as a singleton itemset. There is however a difference in the type of frequent sequences that can be described. Consider the example of a sequence $A.B.C.D.E$ and $A.B.E$, if we would transform these to the traditional model they would become $\langle\{A\}\{B\}\{C\}\{D\}\{E\}\rangle$ and $\langle\{A\}\{B\}\{E\}\rangle$. Focusing on the two traditional sequences, the frequent sequence $\langle\{A\}\{B\}\{E\}\rangle$ is found. However, a frequent sequence using our own definition of sequences, with the same semantics, cannot be found. For example $A.B.\$^+.E$, can be mapped on the first sequence, but not the second, and $A.\$^+.E$ can be mapped on both the sequences, but does not express the presence of B in the pattern. We can thus conclude that both algorithms will not present the same results.

However, this problem can be avoided by introducing the wildcard $\$^*$, which can represent 'zero, one or more' items. But also in this case some difference occurs, as the traditional model is not able to reflect the fact that one item directly follows another, which is the case in our model. Again, consider previous examples $A.B.C.D.E$ and $A.B.E$ and their transformed versions $\langle\{A\}\{B\}\{C\}\{D\}\{E\}\rangle$ and $\langle\{A\}\{B\}\{E\}\rangle$. There does not exist any traditional frequent sequence that expresses $A.B.\$^*.E$, which states, if a sequence contains A directly followed by B, it will also contain E. On the other hand, will every result from the AprioriAll be included in the result of Stops and Moves Apriori? The answer is yes, as every result outputted by AprioriAll can be transformed into an equivalent frequent sequence of our own definition, by inserting a $\$^*$ between every item. For example the frequent sequence $\langle\{A\}\{B\}\{E\}\rangle$ can be transformed into $A.\$^*.B.\$^*.E$ and

will also have the same support count, as they both are contained in all sequences that contain an A, followed by a B, followed by an E.

Again, if we want to find out if there was some increase of efficiency, and thus a reduction of computations, we have to compare the complexity of AprioriAll and Stops and moves Apriori with multi-matching wildcards. We start with candidate generation. As this is similar to that of the 'No wildcards' version, we can conclude this is also similar to AprioriAll. Pruning, however, is still not needed, as the Apriori Principle is respected. And finally we have the support count measuring, which has a different matching algorithm to that of the 'No wildcards' version. The matching algorithm has the same complexity than that of AprioriAll, which is $O(k \cdot M)$, with k being the size of the candidate, and M being the average length of a sequence.

3.4 Generating rules from frequent sequences

Once all frequent itemsets have been found, these itemsets need to be transformed into rules, as these are the end product of our association analysis process and a tool to formulate the newly found knowledge.

3.4.1 Traditional rule formulation

In traditional association analysis, that is to say in case of itemsets instead of sequences, rules are formed in a two-step process. First, for every frequent itemset I we generate all non-empty subsets S of I. Secondly we generate rules by formulating for every S of I the rule $S \rightarrow (I - S)$. If the rule evaluates within the stated minimum confidence, this rule is outputted. To measure the confidence of a rule based on the support count of a frequent itemset we use the following formula:

$$confidence(A \rightarrow B) = \frac{\sigma(A \cup B)}{\sigma(A)}$$

In this formula $\sigma(A \cup B)$ signifies the support count of the itemset that contains both the items of A and B and $\sigma(A)$ the support count of the itemset equal to A. Of course the minimum support of the rules does not need to be evaluated, as these rules are based on frequent itemsets, which already satisfy the minimum support. The collection of outputted rules are the strong rules, which satisfy both our minimum support and confidence.

3.4.2 Sequential rule formulation

In order to be able to formulate rules regarding frequent sequences, some changes are needed that take the specific characteristics of sequences into account. As there are three types of possible frequent sequences, frequent sequences without wildcards, frequent sequences with single-matching wildcards and frequent sequences with multi-matching wildcards, and each of these will be evaluated for needed alterations.

Without wildcards

This type of frequent sequences has the strictest mapping. Given the fact that wildcards do no occur within the frequent sequences, the formulating of rules can be achieved in a relatively simple manner.

For each frequent sequence s of size t we generate $t-1$ possible subsequences $s_1, \ldots, s_{t-1}$ consisting out of the first n consecutive items with n ranging from 1 to $t-1$. For each of these subsequences s_i we generate a rule stating that $s_i \rightarrow (s - s_i)$ and check whether the confidence of this rule falls within the minimum confidence. Measuring the confidence of a rule is similar to how this is measured with frequent itemsets, thus based on the support count.

For example, consider the frequent sequence

$$Hotel_1.Museum_1.Bar_1.Museum_2.$$

Translating this frequent sequence into a set of rules can be achieved by generating the subsequences of this frequent sequence, generating the candidate rules derived from these subsequences and checking whether these candidate rules comply with the minimum confidence. Table 3.5 demonstrates the process of generating subsequences and rules.

Table 3.5: Deriving rules from frequent sequences.

Subsequences	**Rules**
$Hotel_1$	$Hotel_1 \rightarrow Museum_1.Bar_1.Museum_2$
$Hotel_1.Museum_1$	$Hotel_1.Museum_1 \rightarrow Bar_1.Museum_2$
$Hotel_1.Museum_1.Bar_1$	$Hotel_1.Museum_1.Bar_1 \rightarrow Museum_2$

The semantics of this type of rules can be described as following. Consider a rule and sequence, if the antecedent of this rules is present is the sequence, there is a possibility the consequent of the rule will follow the antecedent. The chance of this also being the case is described in the confidence of this rule.

Single-level rules with single-matching wildcards

The next type of frequent sequences we wish to observe, are the frequent sequences that contain single-matching wildcards, for example

$$Hotel_1.Museum_1.\$.Museum_2$$

. These are less strict as the frequent sequences without wildcards, but because of the fact that a frequent sequence is mapped on a subsequence of the same size, there are no extra alterations needed to those proposed in the technique for formulating rules on frequent sequences without wildcards.

Table 3.6: Deriving rules from frequent sequences with single-matching wildcards.

Subsequences	**Rules**
$Hotel_1$	$Hotel_1 \rightarrow Museum_1.\$.Museum_2$
$Hotel_1.Museum_1$	$Hotel_1.Museum_1 \rightarrow \$.Museum_2$
$Hotel_1.Museum_1.\$$	$Hotel_1.Museum_1.\$ \rightarrow Museum_2$

Consider the given example $Hotel_1.Museum_1.\$.Museum_2$, Table 3.6 describes how rules can be derived from this frequent sequence.

Single-level rules with multi-matching wildcards

The third type of frequent sequences, are the sequences that contain multi-matching wildcards. For example $A.B.E.\$^+.G$ and $A.C.\$^+.E.G.\$^+.H$ both contain multi-matching wildcards. Compared to the single-matching wildcards in the previous class of frequent sequences, these wildcards can match one or more items, making them far more flexible than previous patterns, as a frequent sequence of a certain length does no longer be matched on a subsequence of the same length. This flexibility makes them the most interesting of the three and therefore we will focus on its potential.

If we would observe the frequent sequence $A.C.\$^+.E.G.\$^+.H$, how would we derive rules as to describe the knowledge expressed by this frequent sequence. If we would apply the previously proposed technique for generating rules based on frequent sequences, we would conclude in two rules, namely $A.C.\$^+ \rightarrow \$^+.E.G.\$^+.H$ and $A.C.\$^+.E.G.\$^+ \rightarrow \$^+.H$. This last rule states that if a trajectory passes the stops A and C consecutively, and afterwards also passes E and G consecutively, it will eventually pass H.

As our emphasis in this thesis lies on semantics, we might be able to improve the readability of the rules. Especially the use of the wildcard symbol $\$^+$ can be avoided. We wish to propose a series of predicates and a conjunctive operator that would enable us to capture the simple semantics of a rule. The predicate we propose is passes(). This predicate should only contain one or more consecutive items, thus no wildcards. We also propose the use of a conjunctive operator $\vec{\wedge}$ implying a ordinal relation between two propositions. This operator lets us respect the ordinal nature of sequential patterns.

The example used before $A.C.\$^+.E.G.\$^+.H$ delivered two rules, namely $A.C.\$^+ \rightarrow \$^+.E.G.\$^+.H$ and $A.C.\$^+.E.G.\$^+ \rightarrow \$^+.H$. If we would use our proposed predicates and operator, these rules would respectively be translated to

$$passes(A.C) \rightarrow passes(E.G) \vec{\wedge} passes(H) \text{ and}$$

$$passes(A.C) \vec{\wedge} passes(E.G) \rightarrow passes(H).$$

We have defined our Stops and Moves Apriori, we have described three versions, one without wildcards, one with single-matching wildcards and finally one with multi-matching wildcards. Next, we want to implement these three algorithms as to demonstrate how they work and test these on actual data.

Chapter 4

Implementation

The following chapter contains more technical details on the implementation of Stops and Moves Apriori. It demonstrates the concepts introduced in this thesis, and how they can be implemented. First we describe the input, namely the trajectories, and how these are stored in the database. Next we observe the enrichment process, after which we describe the actual mining process and the eventual output.

4.1 Input Description

4.1.1 Trajectories

The input used to evaluate and demonstrate our implementation, is a large collection of trajectories, which are stored in the database as trajectory samples. The table contains a record for each sample, and is defined by the following structure:

- trajectory identifier: indicates which trajectory the sample belongs to.
- a sample identifier: indicates the order of the samples within the trajectories.
- X-coordinate: X-coordinate describes the exact location in a two-dimensional space.
- Y-coordinate: Y-coordinate describes the exact location in a two-dimensional space.
- timestamp: the exact time at which the sample was recorded.

and other less important variables. These trajectories are transformed into semantically enriched trajectories. Figure 4.1 gives a visual representation of our collection of trajectories.

4.1.2 Application

Besides trajectories, we also need an application, which forms the basis of determining the actual stops. We decided it would be best to divide the area over which the trajectories were defined into different districts, as Figure 4.1 shows. These are artificially generated, and not based on any real-world background information. Each of these districts represents a candidate stop to the trajectories that intersect these districts. Every candidate is also associated to a minimum duration, which was set to one minute. We chose a very low minimum, as to maximize the lengths of the semantically enriched trajectories.

This application is also stored in the database, where every candidate stop is associated with an identifier, a name and a geometry.

4.2 Semantic Enrichment Process

The enrichment process transforms trajectory samples into a list of stops by using the background information to interpret the actual semantics of the trajectory. The samples and the application are retrieved from the database.

To perform this transformation, an existing implementation of the SMoT algorithm was used. As described in Chapter 2, the algorithm generates a list of stops for each trajectory. For each stop a record is added to the table 'Stops' in the database, resulting in a semantically enriched trajectory composed of several records. The structure of this database is:

- trajectory id: indicates to which trajectory the stop belongs
- candidate stop id: refers to the id used in the application database as a unique identifier of the candidate stop
- candidate stop name: the name of the candidate stop retrieved from the application database
- candidate stop type: the type of the candidate stop retrieved from the application database

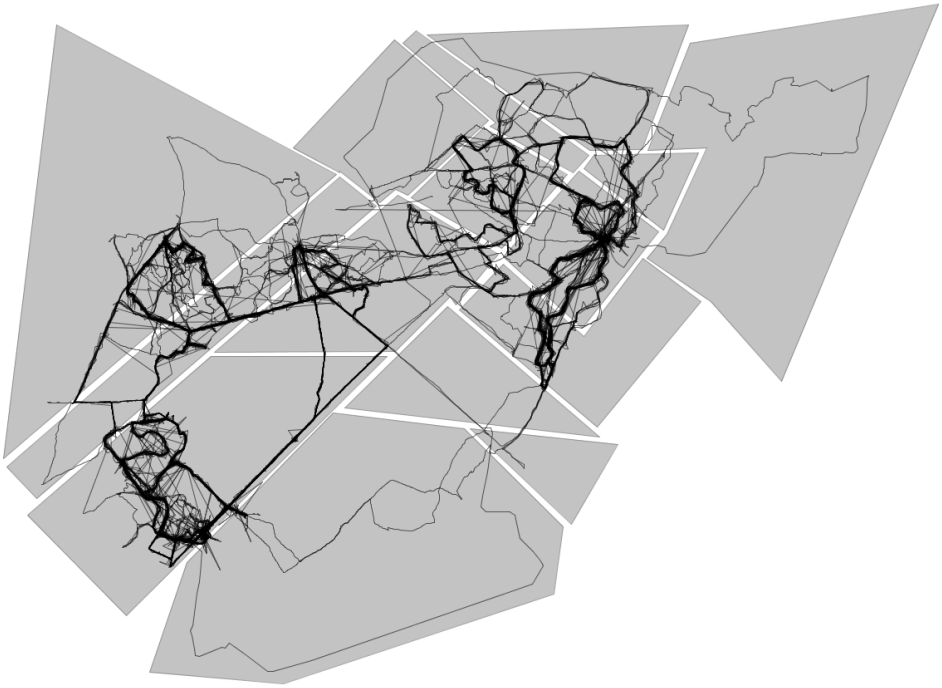

Figure 4.1: A visual representation of the input trajectories and candidate stops as visualized by OpenJUMP.

- time of entering: a timestamp that indicates when the moving object entered the stop
- time of leaving: a timestamp that indicates when the moving object left the stop

4.3 Mining Semantically Enriched Trajectories

The description of our Stops and Moves Apriori defines three possible types of frequent sequences that could be interesting. We implement the three versions: no wildcards, single-matching wildcards and multi-matching wildcards. For each implementation we describe a different class, which all use the general framework we created.

At the end of this chapter we will make a comparison between the three implementations on execution time, number of frequent sequences found and memory usage. A crucial part of this implementation is the sequence class, as it is use the model trajectories and frequent sequences. Before we start discussing the algorithms, we introduce our Sequence class.

4.3.1 Sequence

Our *Sequence* class uses the principle of a double-linked list to compose a sequence of *Stop* objects. An important function is *getStopIds()*, which returns an array of integers containing the identifiers of the stops contained in the sequence. This array is used during the matching of two sequences.

The *Stop* class is a typical data class containing all important variables concerning stops and the appropriate functions to manage these variables. It has one special variable, as it is also used to represent wildcards in candidate frequent sequences, which is a Boolean indicating whether it is a wildcard or not.

4.3.2 DatabaseManager

As our stops are stored in the database, there is the need for a class that coordinates the communication between the application and the database. The *DatabaseManager* class is able to set up this connection and also to retrieve the stops from the database which are then formatted into sequences.

4.3.3 No Wildcards

The first class to be implemented, is the Stops and Moves Apriori without the use of wildcards, described in Chapter 3. To mine a collection of trajectories, these first have to be reconstructed using the stops in the database. After reconstruction the function *execute(double minsup, Vector ⟨Sequence⟩ seq)* is called where *minsup* is the minimum support and *seq* is the collection of sequences, which were inserted in a Vector. We use the pseudo-code described in 3.3 as a blueprint for the structure of this function. We implement the functions mentioned in the pseudo-code: *generateFrequentSingleItemSequences(double minsup, Vector seq)*, *generateCandidates(Vector v, int k)* and *identifySubsequences(Vector candidates, Sequence s)*.

generateFrequentSingleItemSequences(double *minsup*, **Vector** *seq*). One of the first things to do is to determine the frequent sequences of size 1, as these form the basis for following candidate generation. To determine this collection, we have to calculate the support of each stop, and compare this with the minimum support *minsup*.
We introduce a hash-table that contains an integer value for each distinct stop, where this integer value equals the support count of the stop. This support count is calculated by reading each sequence of *seq*. Every time we encounter a stop that was not encountered in this sequences, we increase the integer value of this stop. After all sequences have been processed, we can calculate the support of each stop based on the hash-table and generate a sequence of size 1 for every frequent value. The collection of these sequences is then outputted.

generateCandidates(Vector v, **int** k). This function is responsible for the candidate generation of k+1-sized candidates using the (k)-sized frequent sequences. The latter is referred to as v in the function declaration, and k refers to the size of the frequent sequences. This function has two parts, one which generates candidates of size 2 when $k = 1$, and one for all candidate larger than 2.
This difference is made as the first group does not need matching, while the second needs to match $k - 1$ items.
The first collection is composed by taking every sequence of size 1 and concatenating two sequences. The collection is then outputted.
The second collection, with $k > 1$, needs some matching. If the $k - 1$ items match, the last item of one sequence, is added to the other sequence, as to generate sequences of size $k + 1$. Finaly, the collection of all generated sequences is then outputted.

identifySubsequences(Vector *candidates*, **Sequence** s). The implementation of this function includes the Boyer-Moore algorithm presented in Chapter 3. The output is the collection of candidates from the Vector *candidates* that are present in s. These are the candidates of which the support count has to be increased.
First we build a hash-tree containing all candidates by using the principle mentioned before. The hash-function uses the modulo function to distribute the candidates evenly. We described our hashing function to be:

$$stop - id \ \% \left\lfloor \frac{number\ of\ distinct\ candidate\ stops}{2} \right\rfloor$$

Using this function we were able to build a hash-tree containing candidates, which then can be used to find the candidates using the sequences.
To allow us to determine whether a sequence is contained in another sequence, we used Boyer-Moore. The implementation of the algorithm follows the specification described in the pseudo-code in Listing 3.4. The only changes to be made, is that instead of processing strings, which are in essence an array of characters, the algorithm had to process arrays of integers, which is of course exactly the same. To transform a sequence into an array of integer we use the previously mentioned function *getStopIds()*.

As mentioned, our implementation uses the same structure as described in the pseudo-code, so the algorithm will end in the same way. After the support of every candidate has been calculated, and none of the candidates reach the minimum support, the collection of found frequent sequences is outputted.

4.3.4 Single-matching wildcards

This class does not only use the same framework as the 'No wildcards' class, it extends it. We redefine the previously mentioned functions generateFrequentSingleItemSequences(double *minsup*, Vector *seq*) and identifySubsequences(Vector *candidates*, Sequence *s*).

generateFrequentSingleItemSequences(double *minsup***, Vector** *seq***).** This function calls the function generateFrequentSingleItemSequences(double *minsup*, Vector *seq*) of the 'No wildcard'-class, and adds a new stop to the collection, namely a stop containing a single-matching wildcard.

identifySubsequences(Vector *candidates***, Sequence** *s***).** The purpose of this function is to determine the collection of candidates contained in a sequence. Again, we build a hash-tree to which all candidates are added and we use the same hash-function as previously described. However, in this case the hash-function treats the wildcards in a different way as we provide a special branch for the wildcards. The reason why this branch is special, is because it is always selected during the retrieval of the candidates. So, if we are at a node, for example the root node, we always go down the branch of the wildcards, regardless of hash-value.

After all candidates are selected, we can proceed with matching the candidates onto sequences. The matching is performed by reading the sequence

and checking whether the item read, matches the first item of the candidate. If these match, the second is matched, and so on until all items have been matched. If at some point they do not match, the algorithm goes back to the first item and tries to match it with the next item of the sequence. During this matching, the wildcard in naturally mapped on any item.

4.3.5 Multi-matching wildcards

To implement Stops and Moves Apriori using multi-matching wildcards, we extend the *single-matching wildcards* class and redefine some of its functions, namely generateFrequentSingleItemSequences(double *minsup*, Vector *seq*) and identifySubsequences(Vector *candidates*, Sequence *s*).

generateFrequentSingleItemSequences(double *minsup***, Vector** *seq***).** This function calls the function generateFrequentSingleItemSequences(double *minsup*, Vector *seq*) of the 'No wildcard'-class, and adds a new stop to the collection, namely a stop containing a multi-matching wildcard.

identifySubsequences(Vector *candidates***, Sequence** *s***)** if we would compare this function with the corresponding function of the 'Single-matching wildcards'-class, we would find two differences: the hash-tree and the matching technique.

As we did with single-matching wildcards, we need to treat the wildcard in a special way. Again, we provide a special branch for the wildcards which is always selected during the retrieval. However, the node is also special as we do not only hash the identifier of the next stop, but of all following stops. Every branch which is selected, needs to be considered.

We also need to define another matching technique for which we opted to use recursion. When matching a candidate onto a sequence, we start with the first item of the candidate and match it with the first item of the sequence, if they match we compare the second of both, and so on. If a mismatch occurs, we move our cursor one position and try to match the first item of the candidate with the second item of the sequence. In case we encounter a multi-matching wildcard in our sequence, we recursively call the same function and try to match the remainder of the candidate, after the wildcard, on the remainder of the sequence, after the last matched onto the wildcard. If this recursive call succeeds, the candidate can be mapped onto the sequence.

4.4 Output Description

The output of the mining process is composed of a list of all frequent sequences found in the database, as shown in Figure 4.2. Depending on the type of application ran, this list might include frequent sequences containing single- or multi-matching wildcards.

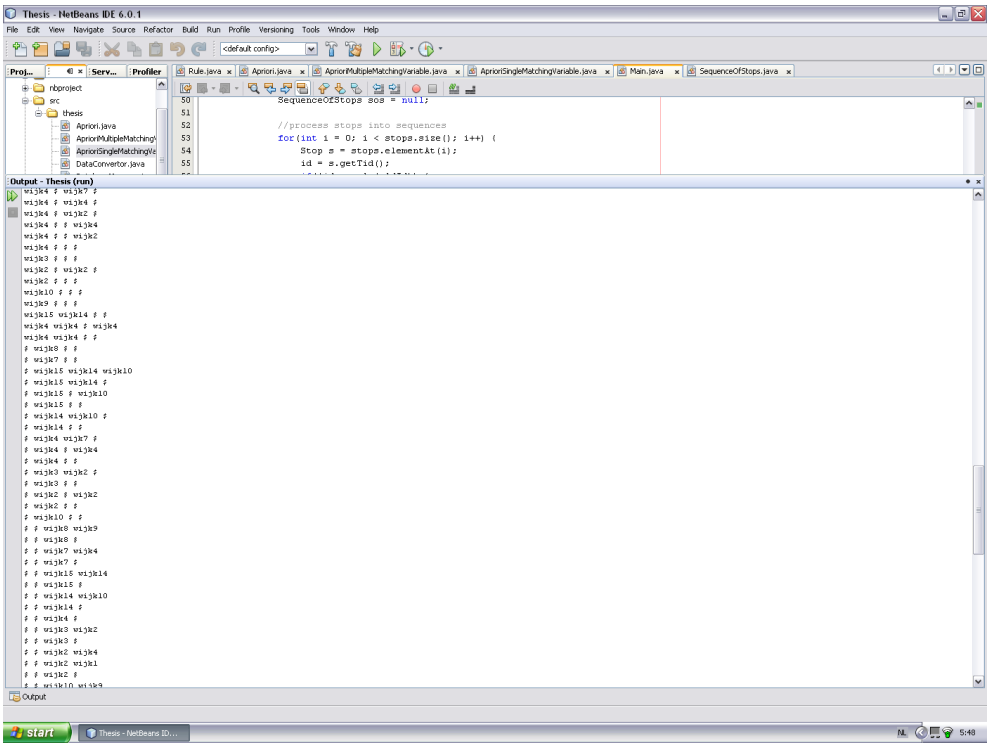

Figure 4.2: Screenshot showing the results of Stops and Moves Apriori using single-matching wildcards.

4.5 Comparison

Following charts give a comparison of the described algorithms on execution time, number of frequent sequences and memory usage. The first chart, Figure 4.3, demonstrates that a decrease of the minimum support leads to an increase in the number of results and also that the use of multi-matching wildcards delivers more results as opposed to single-matching wildcards, and single-matching wildcards as opposed to no wildcards.

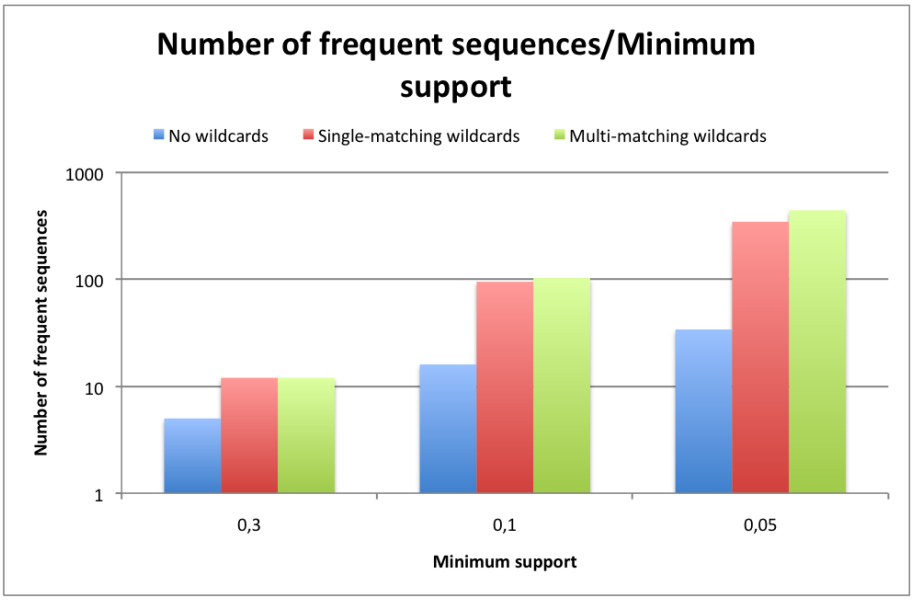

Figure 4.3: A comparison between the different implementations: number frequent sequences vs. minimum support.

The second comparison is that of memory usage. As we can see, the difference between the minimum supports is not very large, so the influence is quite limited. There is a difference between the different implementations, as the use of multi-matching wildcards will demand the most memory, and using no wildcards demands the least.

Finally we also compared the execution time of the different implementations at different minimum supports. Here there is a clear influence of the minimum support on the execution time. This is actually a very logical result, as a lower minimum support will allow more frequent sequences to be discovered.

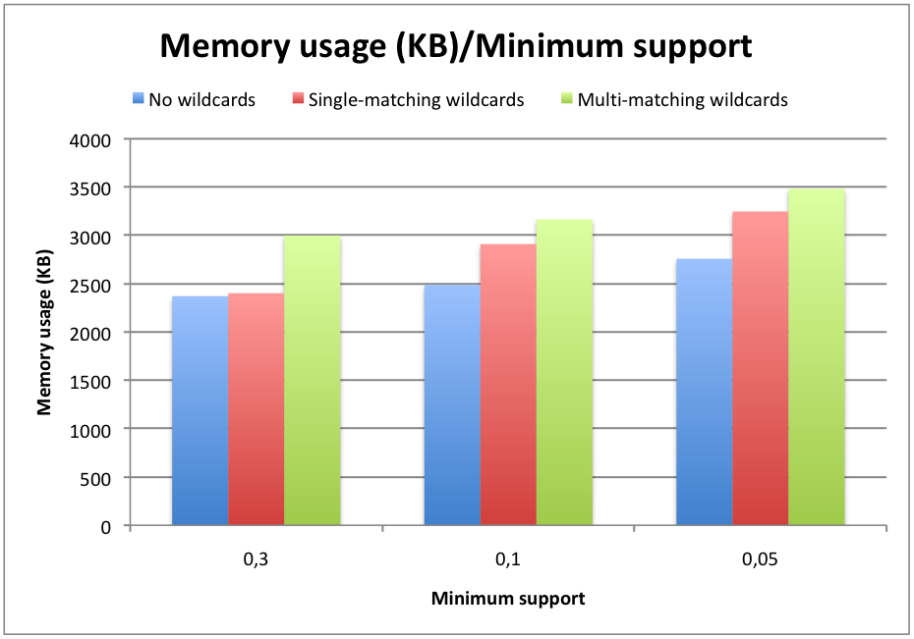

Figure 4.4: A comparison between the different implementations: memory usage vs. minimum support.

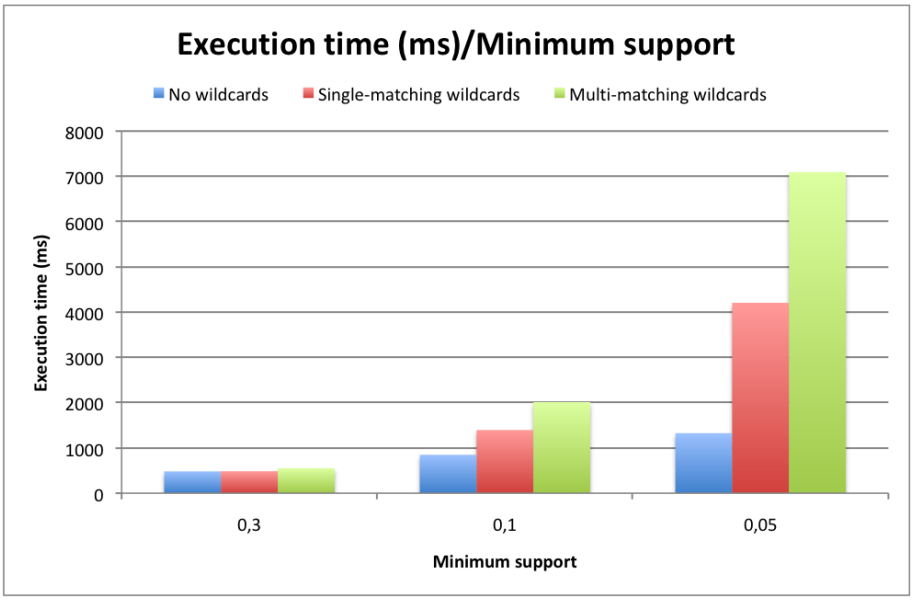

Figure 4.5: A comparison between the different implementations: execution time vs. minimum support.

Chapter 5

Conclusions

Here we conclude our study of mobility patterns. Our initial goal was to find an association analysis tool enabling us to mine semantically enriched trajectories. First we observed how raw trajectories could be transformed into semantically enriched trajectories which focus on their actual meaning by interpreting them using background information. A trajectory is transformed into lists of stops and moves describing the locations of interest this trajectory visits.

The list of stops is then formatted into a sequence, which is the data type allowing us to represent data containing ordered items. As we studied existing algorithms, we found the AprioriAll algorithm, an algorithm that could be applied on sequences. However, this does not focus on the specific characteristics of our semantically enriched trajectories.

As we want to focus on the nature of semantically enriched trajectories, we need to describe our own data model that incorporates these characteristics. As an inspiration we could use the notion of Mobility Patterns, as this is language of regular expressions purposely described to be a querying tool on trajectories. The resulting data model uses a stricter notion of subsequences and allowed the use of wildcards to define candidate frequent sequences.

We described three versions of Stops and Moves Apriori, namely one without the use of wildcards, one using single-matching wildcards and one using multi-matching wildcards. The version that does not use wildcards has the advantage that is could apply Boyer-Moore, a string search algorithm, to match a subsequence onto a sequence, resulting in an decrease of the amount of needed computations, and thus increasing its efficiency. This was made possible by defining our sequences similar to strings.

The next version enables the use of single-matching wildcards, denoted by \$, in frequent sequences as to allow some flexibility. This single-matching wildcard, denoted by $\$^+$ can represent any other item, resulting is less strict frequent sequences. One step further, we introduce multi-matching wildcards, which are used in frequent sequences to represent 'one or more' items. We conclude that this algorithm has similar complexity to AprioriAll, but if we would wish to find the same frequent sequences, we have to introduce $\$^*$, a multi-matching wildcard that can represent 'zero, one or more items'.

We can conclude that our definition of Stops and Moves Apriori captures the nature of our semantically enriched trajectories more than AprioriAll. The definition of our data model and the inclusion of single- and multi-matching wildcards create a stronger semantic context in which results can be placed.

5.1 Future work

We have presented the GSP algorithm, which introduces some new features, sliding window, time constraints and taxonomies, to sequence mining. However, those are not reflected into our algorithm, but they are of interest, especially the use of taxonomies.

In this thesis, we have only focused on Apriori-type algorithms as a basis for our sequential mining. However, it might be interesting to research whether other existing algorithm could be adapted to the use on semantically enriched trajectories.

Bibliography

[AA06] Natalia Andrienko and Gennady Andrienko. *Exploratory Analysis of Spatial and Temporal Data: A Systematic Approach*. Springer-Verlag, Heidelberg, Germany, 2006.

[ABK+07] Luis Otavio Alvares, Vania Bogorny, Bart Kuijpers, Jose Antonio Fernandes de Macedo, Bart Moelans, and Alejandro Vaisman. A model for enriching trajectories with semantic geographical information. In *GIS '07: Proceedings of the 15th annual ACM international symposium on Advances in geographic information systems*, pages 1–8, New York, NY, USA, 2007. ACM.

[AIS93] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. *SIGMOD Record*, 22(2):207–216, 1993.

[AS95] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In Philip S. Yu and Arbee L. P. Chen, editors, *ICDE*, pages 3–14. IEEE Computer Society, 1995.

[BM77] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[dMR05] Cédric du Mouza and Philippe Rigaux. Mobility patterns. *GeoInformatica*, 9(4):297–319, 2005.

[EKSX96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining (KDD-96)*, pages 226–231. AAAI Press, 1996.

[GP08] Fosca Giannotti and Dino Pedreschi. *Mobility, Data Mining and Privacy: Geographic Knowledge Discovery*. Springer-Verlag, 2008.

[GRS99] Minos N. Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Spirit: Sequential pattern mining with regular expression constraints. In *The VLDB Journal*, pages 223–234, 1999.

[HK00] Jiawei Han and Micheline Kamber. *Data mining: concepts and techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[JÖ7] Anke Jäger. Mining of frequent sets using pruning, based on background knowledge. Master's thesis, UHasselt, 2007.

[KMdW06] Bart Kuijpers, Bart Moelans, and Nico Van de Weghe. Qualitative polyline similarity testing with applications to query-by-sketch, indexing and classification. In *GIS '06: Proceedings of the 14th annual ACM international symposium on Advances in geographic information systems*, pages 11–18, New York, NY, USA, 2006. ACM.

[KO07] Bart Kuijpers and Walied Othman. Trajectory databases: Data models, uncertainty and complete query languages. In Thomas Schwentick and Dan Suciu, editors, *International Conference on Database Theory*, volume 4353 of *Lecture Notes in Computer Science*, pages 224–238. Springer, 2007.

[PBKA08] Andrey Tietbohl Palma, Vania Bogorny, Bart Kuijpers, and Luis Otavio Alvares. A clustering-based approach for discovering interesting places in trajectories. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 863–868, New York, NY, USA, 2008. ACM.

[SA96] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generalizations and performance improvements. In Peter M. G. Apers, Mokrane Bouzeghoub, and Georges Gardarin, editors, *EDBT*, volume 1057 of *Lecture Notes in Computer Science*, pages 3–17. Springer-Verlag, 1996.

[SPD+08] Stefano Spaccapietra, Christine Parent, Maria Luisa Damiani, Jose Antonio de Macedo, Fabio Porto, and Christelle Vangenot. A conceptual view on trajectories. *Data Knowl. Eng.*, 65(1):126–146, 2008.

[TSK05] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley, 2005.