OntoC4S framework: constraint based framework for sequential pattern mining

António Pedro Rodrigues dos Santos Martins Pereira

Dissertation to obtain the Master of Science Degree in Information Systems and Computer Engineering

Supervisor: Profª. Cláudia Martins Antunes

Examination Committee

Chairperson: Prof. João Paulo Marques da Silva
Supervisor: Profª. Cláudia Martins Antunes
Member of the Committee: Prof. Diogo Manuel Ribeiro Ferreira

October 2014
Acknowledgments

This work is partially supported by FCT Fundação para a Ciência e a Tecnologia, under research project D2PM (PTDC/EIA-EIA/110074/2009).

First of all, I would like to thank my supervisor, Professor Cláudia Antunes, for all her guidance, motivation and dedication. Especially when proceeding with the work seemed very much impossible, her key words and knowledge were fundamental and precious.

To my close colleagues, for all the support they gave me through the development of this work. As true friends, they were always available to help and to guide me, and their efforts were certainly significant.

To each and every one that helped the development of this work, even if it was just a small piece of advice, my sincere acknowledgment.

At last, but definitely not least, a very special thanks to my mother, sister, father and girlfriend, without whom I can honestly say I could not have done this project and dissertation.

My deepest thanks to them all.
Abstract

Nowadays, data is everywhere and everyone can access it. This data is characterized by its diversification and heterogeneity. So, as consequence, the challenge is to be able of getting useful information of all that data. The interdisciplinary field of data mining appears as a computational process of extracting information from data and transform it into an understandable structure to be applied. Pattern mining is a subfield inside data mining which tries to extract relevant knowledge in the form of patterns from datasets. The final goal of these patterns is to be useful and to help the decision-making process. Constraints are an identified way of getting more interesting patterns and focusing on the pattern mining algorithms to the expectations of the users but until now their importance is very limited by the expression power of the tools used to define them. Ontologies were identified as a new possibility of representing knowledge in a more interesting way. In this work, we developed an ontology based framework for the sequential pattern mining process, the OntoC4S framework, to introduce domain knowledge specified by the user’s input. This framework is able to represent sequential and concurrent events, in order to define constraints over sequential data. The goal of this work is to enable the representation of more expressive constraints when compared with other constraints’ representation and to obtain a broader way of embody user knowledge while optimizing the current algorithms. The results show that the framework achieves the proposed goals while keeping the performance of unconstraint algorithms. This work is done within the D2PM project where a framework for guiding the pattern mining process is the goal to be accomplished.

Keywords

Data Mining, Pattern Mining, Constraints, Ontology, OntoC4S framework, SeqD2PrefixGrowth
Resumo

Atualmente, a informação é omnipresente e todas as pessoas lhe podem aceder. É caracterizada pela sua diversidade e heterogeneidade. Como consequência, existe o desafio de obter informação útil de toda esta informação. A área interdisciplinar da data mining surge como um processo computacional de extração de informação útil de toda esta informação e a sua transformação numa estrutura adequada para a tomada de decisões. O pattern mining é um subtópico para a obtenção de conhecimento na forma de padrões. As restrições foram identificadas como uma ferramenta para a orientação dos algoritmos de pattern mining dentro das expectativas dos utilizadores mas até hoje a sua expressividade tem sido limitada pelas ferramentas utilizadas na sua definição. As ontologias foram identificadas como um caminho para a representação de conhecimento de domínio. Neste trabalho, desenvolveu-se uma framework baseada numa ontologia para o processo de sequential pattern mining, a framework OntoC4S, na definição de conhecimento de domínio através das especificações do utilizador. É capaz de representar eventos sequenciais e paralelos, de modo a possibilitar a definição de restrições sobre dados sequenciais. O objetivo é permitir a representação de restrições mais expressivas quando comparada com outras representações e alargar a possibilidade de integração de conhecimento ao mesmo tempo que otimiza os algoritmos atuais. Os resultados mostram que a framework atinge os objetivos enquanto mantém a performance dos algoritmos sem restrições. Este trabalho está inserido no projeto D2PM onde uma framework para a orientação dos algoritmos de pattern mining é o objetivo a atingir.

Palavras-Chave

Data Mining, Pattern Mining, Restrições, Ontologia, OntoC4S framework, SeqD2PrefixGrowth
# Contents

1 Introduction 1  
   1.1 Problem ................................................................. 1  
   1.2 Objectives ............................................................. 1  
   1.3 Document Structure .................................................. 2  

2 Related Work 3  
   2.1 Pattern Mining ....................................................... 3  
   2.2 Transactional Pattern Mining ....................................... 3  
   2.3 Sequential Pattern Mining .......................... 4  
      2.3.1 GSP ................................................................. 6  
      2.3.2 PrefixSpan ....................................................... 8  
      2.3.3 GenPrefixSpan .................................................. 9  
      2.3.4 SPaRSe ........................................................... 10  
   2.4 Constrained Sequential Pattern Mining ...................... 11  
      2.4.1 SPIRIT (Sequential Patterns mining with Regular expressions constraints) ..... 12  
      2.4.2 PrefixGrowth .................................................... 15  
   2.5 Limitations of using Regular Languages ....................... 16  
      2.5.1 GenPrefixGrowth ............................................... 20  
   2.6 Open Issues ........................................................... 21  

3 Solution 23  
   3.1 Onto4AR (Ontologies for Association Rules) .................. 23  
   3.2 D2PM Framework (Domain Driven Pattern Mining) .............. 23  
   3.3 OntoC4S framework .................................................. 24  
   3.4 Constraint sequence representation ................................ 25  
   3.5 Alphabet items taxonomy representation ......................... 27  

4 OntoC4S: Constraints Ontology for SPM 30  
   4.1 PSL (Process Specification Language) ....................... 30  
   4.2 OntoC4S ............................................................... 36  
   4.3 Taxonomy integration ................................................ 38  
   4.4 Practical Examples .................................................. 40  

5 SeqD2PrefixGrowth: Constraints Algorithm for SPM 45  
   5.1 OntoC4S ontology read and mining algorithm execution .......... 45  
      5.1.1 Algorithm’s first phase: ontology processing ............... 46  
      5.1.2 Algorithm’s second phase: sequence pattern mining process execution .......... 53  

6 Evaluation 58  
   6.1 SeqD2PrefixGrowth performance .................................. 59  
   6.2 Performance comparison with unconstraint algorithms .......... 62  
   6.3 Patterns extraction from unconstraint vs constraint .......... 64  

7 Conclusions and Future Work 68  

Bibliography 69
Appendix

Almaden introduction and experimental setup
## List of Figures

2.1 *GSP* pseudocode (without taxonomies and only with max_gap=δ). ............................................ 7  
2.2 *PrefixSpan* pseudocode. ................................................................................................................ 8  
2.3 *GenPrefixSpan* pseudocode. ........................................................................................................... 10  
2.4 *SPaRSe* pseudocode. ......................................................................................................................... 11  
2.5 *SPIRIT* pseudocode. ......................................................................................................................... 13  
2.6 Automaton $A_R$ corresponding to RE $R$. ......................................................................................... 14  
2.7 *PrefixGrowth* pseudocode. ............................................................................................................... 15  
2.8 Example of a DFA with a simple precedence to change the state. .................................................... 16  
2.9 Example of a DFA representing a richer constraint. ......................................................................... 17  
2.10 Pushdown automaton with itemsets. ................................................................................................. 18  
2.11 Extended pushdown automaton equivalent to the PDA in the figure 2.10. ...................................... 19  
2.12 *GenPrefixGrowth* pseudocode. .................................................................................................... 20  
3.1 *OntoC4S* framework overview. .................................................................................................... 24  
3.2 Constraint sequence *XML DOM* node tree. ................................................................................... 25  
3.3 Items taxonomy representation *XML DOM* node tree. ................................................................. 28  
3.4 Example taxonomy representation tree. ............................................................................................ 28  
4.1 The theories in the *Outer Core* of *PSL*. ....................................................................................... 32  
4.2 UML representing a sequential nursery protocol. ............................................................................ 33  
4.3 UML representing a concurrent nursery protocol. ......................................................................... 35  
4.4 First approach to the ontology with a simple relation. ..................................................................... 36  
4.5 Reification incorporated in the relations of the ontology. ............................................................... 37  
4.6 *OntoC4S* architecture. ................................................................................................................ 38  
4.7 Taxonomy relations in the *OntoC4S* ontology. ............................................................................. 39  
4.8 *OntoC4S* architecture with taxonomy integration. ..................................................................... 40  
4.9 Sequence of a nursery protocol and the gaps between the activities. ............................................. 41  
4.10 Example of the Figure 4.9 through the ontology *ontoC4S*. ............................................................ 42  
4.11 Sequence of a nursery protocol with concurrency and the gaps between the activities. ............. 43  
4.12 Example of the Figure 4.11 through the ontology *ontoC4S*. ............................................................ 44  
5.1 Taxonomy *Java* classes in *UML* modelling language notation. ................................................. 46  
5.2 Order of the taxonomy nodes read when instantiating the java classes. ..................................... 47  
5.3 Rule’s representation of the *exists* constraint. .............................................................................. 49  
5.4 Rule’s representation of the *precedence* constraint. .................................................................... 50  
5.5 Ontology instantiation for the complete example. ......................................................................... 52  
5.6 *SeqD2PrefixGrowth* algorithm mining process overview. .......................................................... 56  
6.1 Execution time with different gap values. ......................................................................................... 59  
6.2 Execution time by main algorithm phases. ....................................................................................... 60  
6.3 Execution time by algorithm mining phases. ................................................................................... 61  
6.4 Execution time proportion by algorithm mining phases. ................................................................ 61  
6.5 Execution time comparision with unconstraint algorithms. .......................................................... 63  
6.6 Memory consuming comparision with unconstraint algorithms. .................................................. 63  
6.7 Unconstraint vs Constraint when support = 10%. .......................................................................... 65  
6.8 Unconstraint vs Constraint when support = 30%. .......................................................................... 66  
6.9 Unconstraint vs Constraint when support = 50%. .......................................................................... 66
List of Tables

2.1 Sequences Database. ................................................................. 5
2.2 GSP candidate generation phase example. .................................. 7
2.3 Count of items in the PrefixSpan algorithm. ............................. 9
2.4 1-projected database in the PrefixSpan algorithm example. .......... 9
3.1 Table of the primitive constraints. ............................................ 27
3.2 XML representation of constraints in the constraint schema. ......... 27
4.1 Example of a plane departing through the key concepts of the PSL Core. ........ 31
5.1 Rules correspondent to the complete example. ......................... 52
5.2 Validation of a sequence over the result rules of the example. ....... 53
7.1 Almaden dataset generator arguments ..................................... A1
1 Introduction

1.1 Problem

Nowadays, data is an abundant and an easily accessible good. In the recent past, the creation, consumption and share of information has increased in an incredible way, accompanied by the technological evolution and the easiest access to that technology for everyone. The number of interactions between possible data sources (as mobile phones and computers) have exponentially increased and the generated data is from now on more heterogeneous and diverse. With this huge amount of new data, lots of challenges and opportunities arise to extract non trivial information that could be useful to the decision making in all levels. With the goal of extracting non-trivial information from data sets, the field of data mining was developed, by combining advanced techniques and methodologies from the fields of machine learning, statistics and databases. The information discovered could become in the form of hidden patterns, unexpected patterns or models governing some classification task.

One of the most interesting and complex challenges in the discovery of new information in the datasets is the focusing and guiding of the algorithms to the extraction of relevant knowledge. The focus of the algorithms to return results, following the user expectations and the domain that is involved, is an important feature to extract actionable knowledge, i.e. useful information that can be (as far as possible) directly converted into decision-making actions [Cao et al., 2007]. The goal of this domain oriented approach will lead to a better accepted and advantageously useful use in businesses and applications, because the resulting new information will prompt the users to take concrete actions to their advantage in the real world. In particular, the state of the art techniques for pattern mining actually discover a huge amount of patterns with no valid domain orientation and useless purpose. Besides, the actual algorithms consume a very large amount of resources (time and search space) and are very complex when dealing with the incorporation of ways to domain orienting the discovery.

This document introduces the OntoC4S framework, a framework created in order to fulfill these issues related to sequential pattern mining, i.e. allow the incorporation of knowledge in the mining process, increase it’s flexibility and management (through the use of a ontology created specifically to the process), and to ease the introduction of that constraints by a user.

This work was done within the D2PM project [Antunes, 2011], where a framework for guiding the pattern mining process is being built. The OntoC4S framework is based in previously work done in this project, as the case of the SeqD2PrefixGrowth algorithm that is based in the GenPrefixGrowth and it’s core process is done over the GenPrefixSpan algorithm.

1.2 Objectives

In particular, the main goal of this works stands for the definition of a family of constraints for sequential pattern mining to be adopted in the D2PM framework. These constraints have to be defined by the user and mapped to an ontology before being used in the algorithm, and so, the first goal to accomplish is to define an ontology that designate sequences and sequential processes. The concretization of this goal was reached in the OntoC4S ontology and user’s input definition.

An important objective of the work is to allow the easy definition of the constraints and the taxonomy
by the user. In the previous approaches that fact was not intuitive and the flexibility of the constraints were not sufficient. This objective was accomplished here when the user is capable of introducing the constraints easily and has flexibility to enrich those constraints. This is an innovation achieved by this work.

Then the objective is to make possible to use that ontology in a sequential pattern mining algorithm, reusing an existent one or building a new algorithm. That was possible when the $\text{SeqD2PrefixGrowth}$ was materialized. This algorithm was created with this purpose of using constraints and so it corresponds with the initial objectives presented before.

All the elements that constitute this process starting from the constraints and taxonomy definition, mapping of these user defined inputs in the ontology and mining process ending in the interested result set was called $\text{OntoC4S}$ framework.

### 1.3 Document Structure

The work described in this document is organized as follows: chapter 2 introduces the Data mining (2.1) and Pattern mining fields (2.1, 2.2, 2.3, 2.4) describing the problem, the main algorithms besides the challenge and the open issues of the field (2.5, 2.6); chapter 3 presents the framework where the solution overview is described, the result framework (3.3) and the schema of the user's defined input (3.4 and 3.5); chapter 4 will present the $\text{OntoC4S}$ ontology created with the taxonomy integration (4.3) and some examples (4.4); the sequential pattern mining algorithm is explained in chapter 5 and the evaluation of the complete framework in chapter 6. At the end of the document, in the chapter 7, the principal topics are highlighted and summarization of our work is done.
2 Related Work

In this chapter we will present the pattern mining topic and the most important subtopics regarding the work. In chapter 2.1, the pattern mining will be stated, as it represents the main topic in this context. Chapter 2.2 and 2.3 are sub-topics of the field of pattern mining, but our focus will be on the Sequential Pattern Mining. Chapter 2.4 describes an application of constraints to the sequential pattern mining and the limitations of the use of regular languages as constraints in the mining process is presented in chapter 2.5. Finally, the use of some constraints may become a problem, as they may not have the power to express the user's expectations. Because of that, it's in chapter 2.6 that we described the open issues on this context.

2.1 Pattern Mining

Data Mining (or Knowledge Discovery in Databases, KDD) is the process of extracting interesting, implicit, non-trivial, previously unknown and potentially useful, information [Frawley et al., 1992] or patterns from large repositories of data (ex. relational databases, semi-structured data, data warehouses, etc). Data Mining can also be defined as an efficiently discovering of interesting rules from a set of data [Srikant and Agrawal, 1996].

Pattern mining is an important topic of research in data mining and with a broad market interest. Its purpose is extracting relevant patterns in large data sets. The patterns include itemsets, subsequences or other substructures as graphs, trees, or lattices, all of them following a specific set of constraints, such as support, which stands that each pattern must have a frequency no less than a specific threshold. Three types of algorithm approaches were developed accordingly to the type of patterns that wanted to be obtained: transactional, sequential, and structured. Beside those approaches, three types of strategies are used as methodology for those algorithms: candidate generation, pattern-growth, and vertical-based methods: some examples are Apriori [Agrawal et al., 1993; Agrawal and Srikant, 1994], FP-Growth [Han et al., 2000b], and SPADE [Zaki, 2001] respectively, for mining transactional patterns.

Sequential pattern mining is an important problem with broad applications, including the analysis of customers purchase behavior, web access patterns, scientific experiments, disease treatment, natural disasters and DNA analysis. The difference between sequential analysis and other approaches is that the first enables the repetition of items whereas the others don't, and the order of the items are relevant, thus enabling its interest in many areas.

2.2 Transactional Pattern Mining

An itemset represents a non-empty set of items appearing together in a transactional dataset that are related in a certain way, such as products purchased by a customer in a market basket visit. By the analysis of these data in transactional datasets we can extract patterns, through the use of data mining algorithms, which could be represented in the form of association rules, as proposed originally in [Agrawal et al., 1993]. As an example of such association rules, by the analysis of a customer's purchase dataset, we can discover a rule that says that people who bought a bed also bought a pillow (or more than one pillow, but in these algorithms the cardinality of items in the same set is considered as one) with 98% of support.
Let \( L = \{i_1, i_2, \ldots, i_n\} \) be a set of literals representing items. The issue of discovering frequent itemsets can be formally stated as follows: let \( D \) be a set of transactions or dataset, where each transaction \( T \) is a set of items (or itemset) such that \( T \subseteq L \) and has a \( TID \) (Transaction ID), which is a unique identifier among all transactions (could be accompanied by a timestamp or a customer id, in the case of purchase datasets). \( X \) is a set of some items in \( L \), or a \( k\)-itemset (in case of containing \( k \) items). A transaction is defined as a tuple \( T = (TID, X) \). A transaction is said to contain \( X \) if \( X \subseteq T \). An association rule is an implication in the form \( X \Rightarrow Y \), with \( X \subseteq L \) and \( Y \subseteq L \), and \( X \cap Y = \emptyset \) (\( X \) and \( Y \) are disjoint).

Three measures are typically used to determine if an association rule is of interest: \textit{support}, \textit{confidence}, and \textit{lift}. The support of an itemset \( X \) (\( \text{sup}(X) \)) is defined as the proportion of transactions in the dataset, which contain the itemset \( X \). Formally, a rule \( X \cap Y \) has support \( \xi \) in the transaction set \( D \) if \( \xi \) of the transactions in \( D \) contain \( X \cup Y \).

\[
\text{Support}(X) = \frac{\text{Number of support transactions}}{\text{Total number of transactions}}
\]

\textit{Confidence} of an association rule is denoted as the proportion of transactions containing the item(s) on the left side of the rule that also contain the item(s) on its right side. Formally, the rule \( X \Rightarrow Y \) holds in the transaction set \( D \) with confidence \( c \) if \( c \) of the transactions in \( D \) that contains \( X \) also contain \( Y \).

\[
\text{Confidence}(X \Rightarrow Y) = \frac{\text{Support}(X \cup Y)}{\text{Support}(X)}
\]

Finally, the \textit{lift} of a rule is the ratio of the observed support to that expected if \( X \) and \( Y \) were independent. If \( \text{lift}(X, Y) < 1 \), then occurrence \( X \) is negatively correlated with occurrence \( Y \), if \( \text{lift}(X, Y) = 1 \) means that the occurrence of both is independent. Larger lift means "more useful" rules.

\[
\text{Lift}(X \Rightarrow Y) = \frac{\text{Support}(X \cup Y)}{\text{Support}(X) \times \text{Support}(Y)} = \frac{\text{Confidence}(Y \Rightarrow X)}{\text{Support}(X)}
\]

### 2.3 Sequential Pattern Mining

A new relevant trend in data mining problem is sequential pattern mining. The goal of sequential pattern mining is to find the relations between items of sequential events (if there exist any specific order of the occurrences of the items) or itemsets. A sequential pattern mining algorithm mines the sequence database looking for repeating patterns (known as frequent sequences) that can be useful to be used later by end users or managers to find association rules between the different items or events in their collected data for purposes such as marketing campaigns, business reorganization, prediction and planning.

Given a database \( D \) of customer transactions, the problem of \textit{mining sequential patterns} is to find the
maximal sequences among all the sequences that have a certain user’s specified minimum support. Each maximal sequence represents a sequential pattern. The discovery of sequential patterns [Agrawal and Srikant, 1995], motivates the problem of finding all sequential patterns following a specific set of constraints, as the support constraint that forces the sequence to be present in the dataset a minimum number of times (user-defined threshold). While related, the issue of finding which items are related together is associated with intra-transaction patterns, while the issue of finding sequential patterns are concerned with inter-transaction patterns. Real applications of sequential pattern mining could be purchase transactions behavior, DNA sequences, scientific experiments, disease treatment, and Web usage analysis.

A sequence is defined as an ordered list of non-empty set of items (called itemsets). An item can occur in an itemset only once, while in a sequence it can occur multiple times. We denote a sequence s by <s1s2s3...sn>, where each sj (1 <= j <= n) is an itemset. The itemset a is denoted as (a1a2a3...an), where each aj (1 <= j <= n) is an item. A sequence <a1a2a3...an> is said to be contained in another sequence <b1b2b3...bm> if there are integers i1 < i2 < i3 < ... < in such that a1 ⊆ b1, a2 ⊆ b2, a3 ⊆ b3, ..., an ⊆ bm. For a better understanding, the sequence ((1 2) (1 3) 4 2) is said to be contained in sequence ((1 2) (2 3) 4) since (1 2) ⊆ (1 2), (1 3) ⊆ (1 2 3), (4) ⊆ (4), and (2) ⊆ (1 2). However, the sequence ((4 5)) is not contained in the same sequence. Inside a sequence, if an itemset is constituted for one item its representation could be without brackets, as in the previous sequence with ((4) (2)) representing different itemsets and ((4 2)) representing that the two items are in the same itemset. A frequent sequence is maximal if it is not a subsequence of any other frequent sequence. The length of a sequence is denoted as the number of elements (items) that are present in the sequence. A sequence called k-sequence has k elements. Using this representation, the problem of mining sequential patterns is that of finding the set of all frequent sequences S in the given sequence database D of items L at the given threshold support ξ. In a real example, each sequence can be viewed as a set of itemsets representing each itemset a transaction made by a customer. In this case the itemsets are ordered by increasing transaction-time. A sequence satisfying the minimum support constraint is called a large sequence. As an example, consider table 2.1, representing 4 sequences. With the support value of 50% we can say that the subsequence ((1 2) 3) is a pattern because it is maximal and the value of support constraint is above 50%. We cannot say the same for the sequence ((1 3) 3) because its support value is below the defined threshold.

![Table 2.1: Sequences Database.](image-url)

Sequential frequent pattern algorithms mainly differ by the way in which candidate sequences are generated and stored, how support is counted and how candidate sequences are tested for frequency. There are three main categories of sequential pattern mining algorithms, namely, apriori-based, pattern-growth and vertical-based. The extraction of patterns consists in several iterations of a level wise algorithm where each iteration extracts longer patterns than the previous and it stops when no more candidates can be generated. The apriori-based algorithms are known by depending largely on the apriori property, which states that "All nonempty subsets of a frequent itemset must also be frequent", and the use of
the Apriori-generate join procedure to generate candidate sequences. Their strategy is characterized by implementing breadth-first search, generate-and-test and multiple scans of the database. Some of the most known apriori-based algorithms are AprioriAll [Agrawal and Srikant, 1995], and GSP [Srikant and Agrawal, 1996]. These algorithms have some issues such as slow performance and the use of a large search space.

Pattern-growth category algorithms use depth-first transversal, suffix/prefix growth, and memory-only feature that do not spawn an explosive number of candidate sequences. In this category, algorithms avoid the generation and test of candidates, making use of different approaches to reduce the search space at each step. Besides that the major cost of pattern-growth algorithms is the recursively projected databases. FreeSpan [Han et al., 2000a] and PrefixSpan [Han et al., 2001] are some of the pattern-growth algorithms. Vertical-based algorithms combine several features that are characteristics of the other sequential mining categories. The ability of these algorithms to use a wide range of efficient features gives them an edge over other algorithms. SPADE [Zaki, 2001] is one example of vertical-based algorithms which combines features from apriori-based and pattern-growth algorithms, first finding frequent 1-sequences in an apriori-like way and then follows a depth-first search as in PrefixSpan. What is expected from a sequential pattern mining algorithm performance is its acceptable performance by running fast in CPU execution time and low memory use, especially when mining with low minimum support values, and scalability.

### 2.3.1 GSP

The GSP algorithm [Srikant and Agrawal, 1996] follows the apriori-based approach, where candidates are generated and then are tested. It starts with the discovery of frequent 1-sequences (k=1) and, with those discovered sequences, generates the set of potential frequent (k+1)-sequences, called candidates. Generalizing, to generate the k-sequences (k-candidates) it uses the frequent (k-1)-sequences discovered in the previous step, as present in the figure 2.1 in candidateGeneration pseudo code. This strategy allows the significant reduction of search space by, significantly, decrease the number of sequences to consider at each step. The creation of k-candidates is done by the combination of any two frequent (k-1)-sequences, such that the proper maximal prefix of one is equal to the proper maximal suffix of the other.

```
GSP (DB, min_sup, δ)
L₁ ← {frequent 1-sequences}
for (k=2; L_k-1 ≠ ∅; k++) do
    C_k ← candidateGeneration(L_k-1, k)
    C_k ← candidatePruning(C_k, L_k-1, DB, min_sup, δ)
    L_k ← supportPruning(C_k, DB, min_sup, δ)
return L_k

candidateGeneration(L_k-1, k)
    for each aє L_k-1 do
        for each bє L_k-1 do
            C_k ← C_k ∪ join(a, b)

candidatePruning(C_k, L_k-1, DB, min_sup, δ)
    for each C_k do
        if not isPossibleFrequent(s, L_k-1, k, δ)
            C_k ← C_k \ {s}

isPossibleFrequent(s, L_k-1, k, δ)
    return ¬∃l_1, ..., l_k such that l_1 ∩ L_k-1

supportPruning(C_k, DB, min_sup, δ)
    for each cє DB do
        for each cє C_k do
            if C_k ∩ s then
                c.support ++
            L_k ← L_k \ {cє C_k: c.support ≥ min_sup}

Figure 2.1: GSP pseudocode (without taxonomies and only with max_gap=δ).
```

To illustrate the generation of k-sequences with the junction of two k-1-sequences consider the fol-
The following table:

<table>
<thead>
<tr>
<th>Frequent 3-sequences</th>
<th>Candidate 4-Sequences after join</th>
<th>Candidate 4-Sequences after pruning</th>
</tr>
</thead>
<tbody>
<tr>
<td>((1 2) (3))</td>
<td>((1 2) (3 4))</td>
<td>((1 2) (3 4))</td>
</tr>
<tr>
<td>((1 2) (3))</td>
<td>((1 2) 3 5)</td>
<td></td>
</tr>
<tr>
<td>(1 (3 4))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>((1 3) 5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2 (3 4))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2 3 5)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 2.2:** GSP candidate generation phase example.

In this table it is present the frequent 3-sequences. The join phase of these sequences result in two 4-sequences candidates. The pruning phase removes one of two candidates due to the fact that not all the subsequences of the sequence ((1 2) 3 5) is present in the 3-sequences, for example the sequence ((1 2) 5). The sequence ((1 2) (3 4)) has all the 3-sequences in the frequent 3-sequence list and because of this it is a frequent 4-sequence.

### 2.3.2 PrefixSpan

*PrefixSpan* [Han et al., 2001] is an algorithm that follows the pattern-growth strategy. The algorithm can be understand as having 3 steps: the first step where the algorithm discovers the frequent 1-sequences, i.e. finds all frequent items in the dataset; secondly, the search space is divided in the number of the frequent items (prefixes) found in the step before; and the last step where the main loop takes place, in which projected databases are created and mined recursively. Concretely, at each k-iteration of the algorithm’s main loop, it discovers frequent k-sequences recursively based on the projection of the frequent (k-1)-sequences (prefixes), and on this way not generating and, posteriorly, test any nonexistent candidate sequence in a projected database. As the algorithms runs the projected databases also shrink. The premise behind this technique is that the growing of frequent prefixes can discover frequent subsequences, as can be seen in the figure 2.2, contrary to the technique of apriori-based approach which project sequence databases by considering all the possible occurrences of frequent subsequences, i.e. the algorithm doesn’t need to generate candidate sequences. As a consequence, the search space is much lesser than the apriori-based algorithms.

### 2.3.3 PrefixSpan pseudocode

```plaintext
PrefixSpan (DB, σ)
return run(<>[], 0, σ, DB)
run (α, k, σ, DB)
for each σ ∈ DB do
    get rid of all the frequent 1-sequences
    get the sequence of the prefix
    α’ ← db
    L ← L ∪ α’
    L ← L ∪ run(α’ k + 1, σ, create Proj DB(α’, DB))
return L
create Proj DB (α, DB)
for each α ∈ DB do
    if α ≤ s
        n ← first Occurrence After(α, 0)
        β ← n.suffix (α, n)
        α-proj DB ← α-proj DB ∪ {β}
return α-proj DB
```

**Figure 2.2:** *PrefixSpan* pseudocode.

In order to illustrate the construction of the projected databases, consider the sequences in the table 2.1. The count of the items is showed in the table 2.3.

If we consider a minimum support above 50%, the item 1 is one of the frequent 1-sequences. The projected database of this item are all the present subsequences that are suffixes of it (the first occurrence
<table>
<thead>
<tr>
<th>Item</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 2.3: Count of items in the PrefixSpan algorithm.

of the item in each sequence). As a result the project database of the prefix 1 is:

<table>
<thead>
<tr>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2 3)</td>
</tr>
<tr>
<td>((2 3) 3)</td>
</tr>
<tr>
<td>((2 4) 3 4)</td>
</tr>
</tbody>
</table>

Table 2.4: 1-projected database in the PrefixSpan algorithm example.

The major cost of PrefixSpan is the construction of projected databases. As the algorithm builds a projected database for each frequent pattern if there is a large number of frequent patterns there is the need of the same number of projected databases. To manage this cost it was developed the bi-level projection in a way of reducing the number and the size of the projected databases and the pseudo-projection to hold the projected databases in the main memory.

Comparatively, the PrefixSpan algorithm has a better performance in runtime and space used than the apriori-based and vertical-based algorithms, mainly in the case of low support values. Besides this fact, the algorithm has good scalability properties when there is an increase in the number of sequences analyzed.

2.3.3 GenPrefixSpan

The GenPrefixSpan [Antunes and Oliveira, 2003] is a generalization of PrefixSpan, for enabling the use of gap. Concretely, this generalization is achieved based on the redefinition of the method used to construct projected databases. Instead of looking only for the first occurrence of $\alpha$, every occurrence of $\alpha$ is considered.

The number of times that each pattern appears may no longer be the same, due the inclusion of all suffixes after the occurrence of an element, thus increasing the database size. In order to be able to properly compute the support for each pattern, an id is associated with each original sequence, ensuring that each original sequence counts at most once for each pattern support in the support counting procedure.

Given that this new method requires that the $\alpha$-projected database considers all the occurrences of $\alpha$, the database considered at each step may be considerably larger than the previous one. Consequently, this means that the search space is no longer smaller at each step, as in the PrefixSpan algorithm.
2.3.4 SPaRSe

SPaRSe stands for Sequential Pattern Mining with Restricted Search and is an algorithm that makes use of restricted databases in an adapted GSP. It follows the strategy of the apriori-based algorithms, i.e. it searches for patterns with growing length at each step after the discovery of frequent elements.

The algorithm ends when no more potential frequent patterns exist to search for. The goal behind restricted databases is to maintain a list of sequences that support a candidate for each of the elements in the set of candidates. The idea is to maintain a list of sequences that support a candidate, and to verify the existence of majority only in the subset of sequences that support both generating candidates.

The way SPaRSe deals with the search space is very similar to PrefixSpan, as it associates each frequent discovered pattern with the set of sequences where it is presented. The support database is this set and it is a way to count the support of a new candidate only in the intersection of the support databases of its parent pattern (k-1-sequences). The difference between this algorithm and GSP is the fact that SPaRSe generates and tests each candidate separately.

The pseudocode of the SPaRSe algorithm is presented in the figure 2.4.
2.4 Constrained Sequential Pattern Mining

The discovery of sequential patterns may become a complex task when we are dealing with a large dataset or when the user wants to embody some knowledge in the process. Constraints appear as the most viable way of solving those issues by focusing “the algorithm on regions of the trade-off curves (or space) known (or believed) to be most promising” [Bayardo, 2002]. By using constraints, the user assumes the responsibility of choosing which of those aspects are most important for the current task. Constraints represent filters on the data that capture application semantics and allow the users to somehow control the search process and focus the algorithms on what is really interesting. Essentially, constraints are used to reduce the search space, decreasing the execution time, the memory needed, and to filter the patterns/rules that are irrelevant, incorporating user domain knowledge and expectations in the mining process. The most used constraint in sequential pattern mining is the user-specified minimum support [Agrawal and Srikant, 1995]. The support of a subsequence is the fraction of sequences in the dataset where the subsequence is present at. Support could also be not considered as a constraint because it is the basis measure for the pattern mining process. The algorithms that just make use of the support return a very large number of sequential patterns, most of them completely uninteresting for the user. This lack of user-controlled focus suffers from two major drawbacks, as stated in [Garofalakis et al., 1999]:

1. Disproportionate computational cost of selective users. Different users may be not interested in the same patterns and ignoring user focus can force selective users with the same support threshold value to get the same results and to consume the same computational costs despite different selective criteria.

2. Overwhelming volume of potentially useless sequences and rules. Users with different interests on data are forced to filter a huge number of returned patterns due to the lack of tools focused on user control.

To resolve these issues many efforts have been made to push user's domain knowledge and expectation inside pattern mining algorithms. To attain that goal, constraints could be used to guide the pattern discovery [Bayardo, 2005]. The use of constraints allows the user to have a tool that enables it for controlling...
the discovery process and enables the user to continue to assume the responsibility of choosing which aspects are important for the current analysis. Some authors consider the constraints to be present only to filter the result set [Hipp and Günther, 2002] while others emphasize the importance of pushing the constraints deep in the mining process [Bayardo, 2005]. The use of constraints may, at most, resume the process of pattern discovery to a hypothesis task when the constraints have a too high level of restriction, impairing the discovery of unknown information. In this situation, the process only follows the constraints applied, given a very little contribution to increase the knowledge for addressing the problem in analysis. Formally, an itemset $S$ is said to satisfy a constraint $C$, if and only if, $C(S) = true$ [Pei et al., 2001].

Not all the constraints have the same relation over data. Interestingness measures are constraints that impose quantitative conditions over the analyzed data in the pattern or rule. These constraints rank the discovered patterns or rules, by quantifying their usefulness and utility, discarding the patterns with an evaluation less than the user-specified threshold $\theta$. By using these constraints, it is possible to both improve the performance of the algorithms, by pruning the discarded patterns, and reduce the number of discovered returned patterns. However there is a major issue with this type of constraints: the difficulty of choosing the value for the threshold $\theta$. The ability of associating a value to the parameter threshold is, often, a blindness process due to the insensitivity of the constraint over the questioned data. The preciseness of the chosen value can have a dramatic impact on the quality and quantity of the results. On the opposite side are the content constraints. Content constraints correspond to filters over the content of the discovered patterns, instead of its relevance. While interestingness measures are quantitative metrics, content constraints are predicates over the value of the items that would be present in the final returned set of patterns [Antunes, 2008]. These constraints try to capture application semantics and introduce it into the mining process.

### 2.4.1 SPIRIT (Sequential Patterns mining with Regular expressions constraints)

The first algorithm proposing the use of Regular Expressions for the discovering of sequential patterns was SPIRIT [Garofalakis et al., 1999]. SPIRIT algorithm follows in general the structure of Apriori strategy. This approach enables the user to insert flexible constraint specification to represent user-controlled focus to be incorporated into the pattern mining process through the use of an automaton represented by a regular expression. The use of formal language tools as a way of defining constraints in sequential pattern mining can reduce the space memory explored by the categorization of constraints type and the returning of those sequences which are frequent and satisfies the given constraint besides the increase of performance taken [Joshi et al., 2012].

The SPIRIT algorithm (whose pseudocode is presented in the figure below) receives an input parameter $C$ to represents a generic user-specified constraint on the mining process, and returns the set of frequent sequences which satisfy the constraint over the database given. The constraint $R$ is specified as a RE over the alphabet of sequence elements using the established set of RE operators, such as disjunction ($|$) and Kleene closure ($^*$) [Lewis and Papadimitriou, 1997]. Thus, a RE constraint $R$ specifies a language of strings over the element alphabet or, equivalently, a regular family of sequential patterns that is of interest to the user. Given any RE $R$ we can always build a deterministic finite automaton $A_R$ such that accepts the language generated by $R$. A sequence $s$ is accepted by $A_R$ if the path of sequence transitions for the elements in $s$ starting from the start state ends in a final accepted state.
Four schemes were proposed by SPIRIT to represent the degree in which the RE constraints are enforced to prune the search space of patterns during the process. Each scheme represents a relaxation for the user-specified RE constraint where each one push a stronger relaxation than its predecessor. The progressive increase of support of the constraint implies a subset relationship between the frequent sequences determined for each pass $k$.

$$F^\text{SPIRIT}(R)_k \subseteq F^\text{SPIRIT}(V)_k \subseteq F^\text{SPIRIT}(L)_k \subseteq F^\text{SPIRIT}(N)_k$$

The first SPIRIT algorithm ($\text{SPIRIT}(N)$, "N" for Naïve) employs the weakest relaxation, where only candidate sequences containing elements that do not appear on $R$ are pruned, i.e. requires that all elements in $C_k$ appear in the RE $R$. This relaxation guarantees the anti-monotonicity property of the constraint, turning the candidate generation and pruning as in GSP. A constraint $C$ is said to be anti-monotone if and only if it satisfies the apriori property, which means that $\forall S' : S' \subseteq S \land S$ satisfies $C \Rightarrow S'$ satisfies $C$).

The SPIRIT($L$) ("L" for Legal) scheme empowers the constraint power of the previous algorithm by requiring that every candidate sequence to be legal with respect to some state $A_R$. A sequence $s$ is said to be legal with respect to state $b$ of automaton $A_R$ if every state transition in $A_R$ is defined when following the sequence of transitions for the elements of $s$ from $b$. As an example, if we consider the RE constraint $R = 1 2^* 3 (4 5 | 3 | 5 5)$ we can say that the sequence $< 1 3 4 >$ is legal with respect to state $a$ and sequence $< 3 5 >$ is legal with respect to state $b$ whereas sequence $< 1 4 5 >$ is not a legal sequence with respect to state $a$. 

---

Figure 2.5: SPIRIT pseudocode.
The third algorithm, named $SPIRIT(V)$ ("V" for Valid), goes one step further by applying a filter to remove candidate sequences that are not valid with respect to any state of $A_R$. A sequence $s$ is said to be valid with respect to state $b$ of automaton $A_R$ if $s$ is legal with respect to $b$ and the final state of the transition path from $b$ on input $s$ is an accept state of $A_R$. Considering the finite automaton above representing figure 2.6, we can say that the sequence <2 2 3 5 5> is valid with respect to state $b$ because the path following the sequence ends in an end state of $A_R$, but is not a valid sequence, due to the fact that it is not valid with respect to the start state $a$ of $A_R$, whereas the sequence <1 3 3> is a valid sequence (since $a$ <1 3 3> $e$ and $a$ is the start state and $e$ is an accept state).

The last algorithm, termed $SPIRIT(R)$ ("R" for Regular) represents the stronger relaxation of all the above. This approach pushes $R$ as-is inside the mining process by counting support only for valid candidate sequences, i.e., sequences accepted by $A_R$.

2.4.2 PrefixGrowth

The $PrefixGrowth$ algorithm [Pei et al., 2002] was developed motivated by the goals of resolving the lack of cover of many practical constraints and the miss of a systematic method to push various constraints in the process categorizing the previous algorithms.

The algorithm follows the strategy adopted by the PrefixSpan [Han et al., 2001] and incorporate prefix-monotone constraints in the mining process. As $PrefixGrowth$ is built on a prefix-monotone property, all the monotonic and anti-monotonic constraints (with regular expressions constraints as example) can be pushed deep in the sequential pattern mining algorithm.

Figure 2.6: Automaton $A_R$ corresponding to RE $R$.

Figure 2.7: $PrefixGrowth$ pseudocode.
The algorithm above presents the pseudocode for \textit{PrefixGrowth}. As can be seen, this algorithm takes \textit{PrefixSpan} as the basic sequential pattern mining algorithm and integrates a constraint $C$ in the mining process, i.e. only extends sequences that are prefixes of accepted sequences (sequences that satisfy the constraint). Besides the use of constraints in the mining process, another difference of \textit{PrefixGrowth} algorithm to \textit{PrefixSpan} is that the \textit{discoverFList} procedure also removes the sequences that do not contain any accepted subsequence from the \textit{dataset}. Like \textit{SPIRIT}, \textit{PrefixGrowth} is able to deal with non-monotonic constraints, but in an easier way as it only demands that relaxed constraints are prefix-monotone. A constraint is said prefix-monotone if it is either prefix anti-monotonic or prefix-monotonic.

\textit{PrefixGrowth} is more efficient than \textit{GSP} and \textit{SPADE} (where there is no constraint besides support measure), specially in the case where the support threshold is low. When compared with \textit{SPIRIT} algorithm, \textit{PrefixGrowth} algorithm is so efficient as \textit{SPIRIT} (concretely \textit{SPIRIT}(V)) when dealing with simple regular expressions but when the complexity increases the performance of \textit{SPIRIT} decreases more dramatically than \textit{PrefixGrowth}. The algorithm shows still scalability properties when pushing constraints with low support threshold values and has linear scalability when database increases [Pei et al., 2007].

### 2.5 Limitations of using Regular Languages

The use of regular languages to restrict the returned result set in the mining process over sequences was the first approach to embody knowledge and, consequently, to obtain more focused and interesting patterns. Indeed, it is only viable in a situation where the knowledge to be contained in the constraint is relatively simple. As an example, consider a regular language generated by a deterministic finite-state automaton $M$ being a tuple $<Q, \Sigma, \delta, q_0, F>$, where $Q$ is the set of states, $\Sigma$ the actions possible, $\delta$ are the actions needed to the state change, $q_0$ the initial state, and $F$ the final state(s). One of the simplest DFA that could be made is an automaton where there is a simple precedence to change the state, as can be stated in the figure 2.8. In this manner, the only accepted sequence by this DFA is $<(a) (b) (c)>$.

![Figure 2.8: Example of a DFA with a simple precedence to change the state.](image)

If we want to increase the complexity of the DFA, and consequently, define a richer constraint, we easily get a larger and more confusing schema. In the figure 2.9, we can see a DFA with the same actions but with more elaborated patterns allowed.
In this case, the patterns allowed (i.e. valid and accepted sequences over the DFA) are sequences with zero to four times the item ‘a’ followed by the same number of the item ‘c’ that ends with a ‘b’. This case illustrates one relevant limitation when using DFAs which is that the cardinality of an item compared with others must be explicit, i.e. it is impossible to define that the numbers of ‘a’ must be the same as ‘b’ which ever they are but, instead, the cardinality of them must be defined. This limitation forces the DFA to increase the complexity easily as stated.

Summing, the DFA is not a valid tool when there is interest in embodying knowledge and when used the DFA gets complex to define and understand. This increasing of complexity could have more consequences as the decrease of efficiency of the algorithm running.

So, the use of regular languages to constraint the mining process have important limitations and so are unable to represent some of the most interesting patterns. Since context-free languages are more general than regular ones and are able to deal with part of those patterns, it is natural that we try to use them to guide the mining process. Although the use of context-free languages bring some challenges. The first one is related with the use of context-free languages with itemsets. Since context-free languages deal with symbols instead of itemsets, the use of any formal language to constrain the mining process is not directly possible. A naïve way to deal with this problem is to consider each itemset corresponding to a different symbol in a new alphabet that corresponds to the powerset of items in the original alphabet (excluding the empty set). As an example, if we consider three different items x, y, and z, a new alphabet could be made with as many symbols needed to represent all possible combinations of these three items. If the symbols considered in the new alphabet are a, b, c, d, e, f, and g, with the symbols a, b, and c corresponding to x, y, and z, respectively, and the combinations (x, y), (x, z), (y, z), and (x, y, z) being represented, respectively, by d, e, and g. The second problem is related to the representation of context-free languages, deterministic finite automata are not rich enough to be able to represent them (deterministic finite automata are only able to generate regular languages) and a new formalization is needed. The use of (non)deterministic pushdown automata are able to generate a context-free language. Besides these problems, using context-free languages introduces two challenges: how to effectively manipulate the pushdown stack, and the best way to deal with the non-deterministic property of the languages. To illustrate some difficulties, lets consider the pushdown automaton represented below in the figure 2.10.
The pushdown automaton represented is able to generate sequences that have the same number of baskets \((a, b)\) on the sides of \(c\) (e.g. \((a, b)c(a, b)\) or \((a, b)(a, b)c(a, b)(a, b)\)). Using this automaton as a constraint in a discovery pattern algorithm as \textit{PrefixGrowth}, if \(a, b,\) and \(c\) were frequent, when the algorithm tries to discover the frequent items after \(a\), which operation should be made over the stack? Pushing \(X\) will enable \(c\) after \(a\) but applying the same action when \(b\) is found will enable the acceptance as "potentially accepted" of sequences like \(aaa, aaaaa\) (and so on), since \(S\) would remain on the top of the stack.

As a consequence, being able to use context-free languages in the sequential mining process, the syntax and semantics of pushdown automata have to be extended. That extension of the notion of PDA enables the creation of ePDA (extended pushdown automaton) [Antunes, 2005]. This formalization differs from PDA in the transition function, which instead of manipulating items and stack elements, deals with itemsets and strings of stack elements.

**Definition.** An extended pushdown automaton (ePDA) is a tuple \(E = (Q, \Sigma, \Gamma, \delta, q_0, w_0, F)\), with \(Q, \Sigma, \Gamma, q_0\) and \(F\) defined as for pushdown automata; \(w_0 = Z_0\$\) with \(Z_0\) the start symbol as in pushdown automata, and \(\delta\) defined as a mapping function from \(Q \times P(\Sigma)\cup\{\}$\times \Phi\) to finite subsets of \(Q \times \Phi^*\), with \(P(\Sigma)\) representing the powerset \(\Sigma\) and \(\Phi\) defined as follows

\[
\Phi = \{ w \in \Gamma \bigcup \{\$\} : w = x \$ \land x \in \Gamma \}
\]

This extended pushdown automaton is able to generate the same sequences as the PDA before. Using this ePDA as a constraint in the sequence mining process, using \textit{PrefixGrowth} and discovering the same items as the case before, pushing \(XY\) will enable \(c\) after \(a\), but after discovering \(b\), the acceptance as "potentially accepted", of sequences like \(aaa, aaaaa\) (and so on) will not be permitted (as the consequence of emptiness of the top of the stack).
Summing, the use of context-free languages, instead of regular languages, to define a constraint (in terms of ePDAs instead of DFAs) will, more probably, result in a more useful set of patterns discovered in the sequential pattern mining. To be able to deal with context-free languages (through ePDAs), algorithms need to support pushdown automata (compared with the algorithms before that are only able to support regular languages (through DFAs). To achieve this and other goals, the GenPrefixGrowth algorithm was created [Antunes, 2005].

### 2.5.1 GenPrefixGrowth

The GenPrefixGrowth algorithm is an extension of the GenPrefixSpan which follows closely the proposals of the PrefixGrowth algorithm. The principal reason for the creation of this algorithm was not only the possibility of the use of context-free languages (ePDAs) but the allowance of using a class of constraints. To achieve this, GenPrefixGrowth algorithm deals with $\Omega$-constraints. The $\Omega$-constraint is a triple $\Omega = (\varphi, \theta, \sigma)$, where $\varphi$ is a content constraint, $\theta$ is a temporal constraint and $\sigma$ is an existential constraint. This kind of constraint is able to allow an easier representation of background knowledge (especially, temporal and content knowledge). The existence of several constraints simultaneously instead of simple one, forces the algorithm to filtrate the potential patterns analyzed through content, temporal and existential filters. Increasing the complex of constraints applied in the algorithm may lead to size and temporal efficiency problems. So, in order to deal efficiently with this issue the algorithm must avoid the multiple test of each potential pattern. The pseudo code of GenPrefixGrowth is presented in the figure 2.12.

![Figure 2.11: Extended pushdown automaton equivalent to the PDA in the figure 2.10.](image)

One of the strategies applied to deal with this new kind of complexity happens on the discovery of the
frequent \(1\)-itemsets. Instead of just checking which are the frequent elements, the procedure, for example, has to detect the sequences that have events belonging to a specified time interval defined in the constraint (this actions correspond to the \(\text{DiscoveryL1}\) procedure in the figure above).

The definition of this kind of constraints and the \(\text{GenPrefixGrowth}\) algorithm lead to some interesting results, as stated by the authors: the incorporation of \(\Omega\)-constraints in the sequential pattern mining leads to a more efficient process when compared with a post-processing of the result set; and the use of context-free languages doesn’t invalidate all the issues showed before.

2.6 Open Issues

As stated before, there’s a need of incorporating domain knowledge in pattern discovery in data mining algorithms. The incorporation of domain knowledge in data mining has been identified as one of the most important challenges [YANG and WU, 2006], due to the possibility to guide the algorithm through the discovery of more focused and, consequently, returning of more interesting results, besides the representation of domain semantics and user expectations. We can define this embody of knowledge in the process as the need from two points of view, the academic: which wants to optimize the performance and introduce generalization in the process; and the business view which wants to be able to incorporate complex knowledge environment in the algorithms through constraints that can answer in time to the real needs and may guide the decision-making process. There’s a need of making the patterns prompt action in real world [Cao et al., 2007].

Using constraints to incorporate domain knowledge in (sequential) pattern data mining, besides of the advantages stated before, reduce the set of resulted patterns (and possibly the search space in the algorithm), minimizing two problems of data mining field. For this motive the use of constraints is the most explored form of aggregating domain knowledge (there are other techniques of incorporating domain knowledge as using labels or asking for Human interaction). Some formalizations of representing constraints were showed before and each of them have their advantages and disadvantages and represent some intended restrictions more powerfully than others. One premise that has to be present when defining constraints is not to push too restrictive constraints that could limit the patterns discovery to a simple hypothesis testing approach.

A more expressive way of representing domain knowledge is made by graphical models. With these models, users and experts are able to represent the domain with concepts hierarchies and relations. As an example of these models there are the taxonomies, which model an \(\text{is-a}\) relation between concepts. By using these models to define constraints in data mining process, they may lead to more interesting results that could span different levels of the taxonomy in the hierarchy of concepts, enabling the generalization of results and the prune of redundant results [Silva and Antunes, 2014]. Ontologies are other example of a graphical model and are getting the attention of the data mining community in recent years.

An \textit{ontology} is an explicit specification of concepts and relationships that can exist between them. Ontologies are content theories about the objects, their properties and the relations that are possible in a specified domain of knowledge, formalizing the system of knowledge representation for that domain [Chandrasekaran et al., 1999]. It is an agreement of an explicit specification of a shared conceptualization of a domain, not just a representation vocabulary [Gruber, 1993]. Formally, an \textit{ontology} is a tuple \(O := (\mathcal{C}, \leq, \mathcal{R}, \sigma_{\mathcal{R}}, \leq, \mathcal{A})\), where \(\mathcal{C}\) corresponds to the set of concepts defined in the ontology, \(\leq\) to the hierarchy of concepts (\(\text{is-a}\) relation between concepts), \(\mathcal{R}\) is the set of relation identifiers, \(\sigma_{\mathcal{R}}\) the set
of relation functions, \( \leq_R \) the hierarchy of relations, and \( \mathcal{A} \) a set of logical axioms [Stumme et al., 2006]. Ontologies define the environment in which the instances in the KB should be understood. A knowledge base (KB) for an ontology contains assertions about the instances of its concepts and relations. Formally, a KB is a tuple \( KB := (\mathcal{O}, I, inst_c^{-1}, inst_R^{-1}) \), where \( \mathcal{O} \) is an ontology, \( I \) a set of instances, \( inst_c^{-1} \) the concept instantiation function, from \( I \) to \( c \) in ontology \( \mathcal{O} \), that defines to which concept each instance refers to, and finally \( inst_R^{-1} \) the relation instantiation function, from \( R \) to \( 2^{I \times I} \) (in the case of attributes, to \( 2^I \), identifying the existing relations between instances.

Despite of being a graphical model as taxonomies, ontologies are more powerful than taxonomies because they model not only the hierarchy of concepts (an is-a relation) but the relation between hierarchies and axioms as well. With these features and properties, ontologies are an interesting form of conceptualization of domain and thus be a well-accepted form to define constraints used to guide the data mining algorithms getting more interesting rules than the other approaches studied till now. Consequently, similar to constrained data mining, we can also define a set of constraints based on the characteristics of an ontology that can then be used by the presented constrained approaches [Artunes, 2007, 2008].
3 Solution

To achieve the goal of incorporating domain knowledge and the use of ontologies as guiders of the pattern discovery algorithms in data mining, we developed a framework composed by an ontology (On-toC4S - chapter 4) that maps constraints over sequences and an algorithm (SeqD2PrefixGrowth, chapter 5) that uses the ontology to create rules that respect the constraint sequence defined and return frequent patterns over that rules. User defined inputs (constraint set and taxonomy) are defined in sections 3.4 and 3.5. The framework overview is described in section 3.3. This work is inspired by and a continuation of the Onto4AR [Antunes, 2007; 2008], in section 3.1, and D2PM [Antunes, 2011], section 3.2, frameworks and in the ontology language PSL, in section 4.1, which we chose to serve as a guide to the development of the ontology proposed as solution to the problem stated in the previous chapter.

3.1 Onto4AR (Ontologies for Association Rules)

The Onto4AR framework [Antunes, 2007, 2008] was proposed, aiming for addressing the problem of transactional pattern mining in the presence of domain knowledge. It was based on the use of an ontology and assumed a new formulation of the problem, where the meaning of an item was clearly defined in the context of the ontology and the mining process was guided by a constraint. In its origin the Onto4AR framework proposed several types of constraints, from interestingness measures, such as support and confidence, to content constraints based on the relations defined in the ontology.

The Onto4AR framework proposed a first frame for using domain knowledge in the pattern mining process, but doesn’t explain clearly how constraints are defined based on the available knowledge.

3.2 D2PM Framework (Domain Driven Pattern Mining)

The D2PM framework [Antunes, 2011] has the goal of supporting the process of pattern mining with the use of domain knowledge, and encompasses the full range of pattern mining methods, from transactional to sequential and structured pattern mining. This framework was conceived based on previous work on the Onto4AR framework. In this framework the problem of pattern mining is formulated in the presence of domain knowledge, represented by an ontology, and define the mining process guided by constraints defined over the represented domain knowledge. In particular, describe what constraints are, and how can they be applied.

Our goal is to extend the D2PM framework through the definition of constraints to discover sequential patterns. The idea is to form an ontology that allows the existence of sequential and concurrent events to embody domain knowledge in the patterns discovery. To create the ontology, we analyzed some ontologies and languages independent of the application domain they were made but following the features we referenced before. The Process Specification Language [Schlenoff et al., 1999] was identified as the most promising instance of that ontology.
3.3 OntoC4S framework

Based in the former frameworks and with the goals showed before, we created the OntoC4S framework. As stated before, this framework is composed by an ontology (OntoC4S ontology), and an algorithm (SeqD2PrefixGrowth) to incorporate the constraints defined in the ontology inside the pattern mining process and limit the return set by the interest of the user. For a user to be able to use this framework he will need to define the constraint sequence (by which he wants the frequent patterns to respect) and the taxonomy of items present in the dataset from where patterns will be extracted from.

The framework overview is presented in the figure 3.1. In the figure, it is possible to observe the main flow of the framework since the definition of the constraint sequence to the extraction of the final set. As indicated, the first phase consists in the definition of the constraint sequence and the taxonomy, followed by the read of these inputs to the OntoC4S ontology (more details about this step is present in subchapters 3.4, 3.5, and chapter 4). After mapping from the input to the ontology, the algorithm SeqD2PrefixGrowth is responsible for the interpretation of the ontology and the translation of the constraint sequence to the respective rules that will be used to force the frequent patterns of the dataset through the algorithm (in chapter 5). In resume, the framework is composed by 4 steps that accomplish the goals proposed in the document. The rest of the document will be focused in the explanation of these steps.

Next, we will to present details about the framework and the components that make part of it.

3.4 Constraint sequence representation

As stated before, the framework receives as user input, the constraint sequence defined by the user that will define the return set of patterns, the taxonomy of the elements which belong to the alphabet of the dataset, and the dataset over which the algorithm will extract the return set.
The constraint sequence defined by the user is explicated by an semi-structured data file representation, in this case the XML document type. The schema of this document is defined in order to enable the user to define all the elements of the constraint sequence: the constraint, the elements participating in each constraint, the gaps between the relations, if the constraint is concurrent or sequential to another constraint, and some metadata to identify the constraint sequence. In the next paragraph we will show how the schema metadata allow the user to represent each element of the constraint set.

The integral definition of a constraint sequence is defined under a node of the XML, the "constraintSet" node. The child nodes of this parent node contain the nodes that represent the elements of the constraint sequence. The first nodes are the ones that enable the identification of the constraint, allowing the definition of the name ("title" node) and the description ("note" node).

![Figure 3.2: Constraint sequence XML DOM node tree.](image)

The second type of elements to define are the constraints itself. The XML node (called "constraint") has an attribute that identifies the constraint in the constraint sequence ("id" attribute). This attribute will allow the unique identification of the constraint in order to, as an example, order the sequence of the constraints. As a child of the "constraint" node comes the node that indicate what is the constraint type (e.g. "exists" constraint that forces the existence of a specific item) that the user wants to introduce (in the figure 3.2 these node are not represented due to the specification of the representation of each constraint). The attributes and child nodes of each type of constraint depends on the constraint (in the case of a constraint that relates two items, as the "precedence" constraint, the items should be indicated and the gap between them). The number of constraints defined in the constraint sequence can be in the range $[1, +\infty]$. Consider the XML code where a constraint (with a $id = 2$) of precedence is defined over the items 1 and 4. The distance between them must be at maximum of 3 itemsets and they must be in different itemsets ("isParallel" attribute has the value false). In the case of gap with value over 0, the constraint could never be concurrential because it will force the constraint to happen all over the same
itemset. The respective code could be seen next.

```xml
<ConstraintDefinition>
  <constraintSet>
    <title>Numbers ordering</title>
    <note>Force a sequentiality of specific numbers.</note>
    <constraint id="1">
      <exists>1</exists>
    </constraint>
    <constraint id="2">
      <precedence gap="3" isParallel="false">
        <antecedent>5</antecedent>
        <consequent>6</consequent>
      </precedence>
    </constraint>
    <constraintProperties>
      <constraintSequence type="start" gap="0"/>
      <constraintSequence type="sequential" rel1="1" rel2="2" gap="2"/>
    </constraintProperties>
  </constraintSet>
</ConstraintDefinition>
```

As a sequence of constraints, the order of each constraint should be indicated. To enable the ordering the node "constraintProperties" is the parent node of the elements that forces the order. Each node (named "constraintSequence") defines the order between two constraints (where the id of each constraints are identified - attribute "rel1" identifies the constraint by its id that appears first and attribute "rel2" identifies the next constraint). In this node is possible to explicit a maximum gap between the constraints ("gap" attribute) and if the constraint coexist sequentially or concurrently ("type" attribute), that is, the order defined could indicate that the constraints happen at the same time. In sequential pattern mining the concurrence between two constraints means that both are present in the same itemset, while sequent constraints represent ones that are in sequent itemsets. The XML schema is rich enough to enable the representation of this properties between the constraints and in the constraint sequence itself.

As an example, consider a constraint set with the name "Numbers ordering" and a description "Force a sequentiality of specific numbers.", in which we declare that the item 7 should exist (constraint with id = 1) and the element 5 should precede the element 6 with a maximum gap between them of 3 itemsets (constraint with id = 2). The constraint with id = 1, has a gap of 0 since the start of the sequence and this constraint comes first than the constraint with id = 2 with gap of 2. Both constraints are sequential to each other.
This complete example will allow the algorithm to get frequent patterns that respects the introduced constraints in the returned set.

The most important primitive constraints, that the framework is able to deal, are: begin, exists, and precedence. The meaning of these constraints primitives are presented in the following table 3.1 and the respective representation's schema of the constraint in the XML in table 3.2.

<table>
<thead>
<tr>
<th>Constraint primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin (item X)</td>
<td>An item X which starts a frequent pattern</td>
</tr>
<tr>
<td>Exists (item X)</td>
<td>An item X which exists in the frequent pattern</td>
</tr>
<tr>
<td>Precedence (item X, item Y)</td>
<td>An item X that appears before an item Y</td>
</tr>
</tbody>
</table>

Table 3.1: Table of the primitive constraints.

<table>
<thead>
<tr>
<th>Constraint</th>
<th>XML representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin (item X)</td>
<td>&lt;begin&gt;X&lt;/begin&gt;</td>
</tr>
<tr>
<td>Exists (item X)</td>
<td>&lt;exists&gt;X&lt;/exists&gt;</td>
</tr>
</tbody>
</table>
| Precedence (item X, item Y) | <precedence gap="3" isParallel="false">
|                            | <antecedent>X</antecedent> |
|                            | <consequent>Y</consequent> |
|                            | </precedence>       |

Table 3.2: XML representation of constraints in the constraint schema.

Other constraints could be written as a composition of these primitive constraints (the same could the done only when generating the rules for the algorithm to deal with, more can be seen in the SeqD2PrefixGrowth - chapter 5).

3.5 Alphabet items taxonomy representation

A big advantage of this framework is the ability to deal with a taxonomy of items. A taxonomy is composed by the greek words taxis and nomos, order and law respectively, which means a forced order or arrangement of something. In this context, a taxonomy represents an order among concepts in a representation meaning, that is concepts that represents other more concrete concepts.
This approach will enable the relaxation over the returned frequent patterns set due to the possibility of affirming that a concept in a constraint is a concept and set over other allowed concepts. The framework allow the user to define as many parent concepts as he wishes. This feature is very powerful because it doesn't force the user to run various instances of constraints for each of the combinations of items and so it will gave the possibility of make the process more productive and spent less time.

As in the case of the constraint sequence declaration (where XML DOM node tree is represented in figure 3.3), the user will declare the taxonomy of concepts in a semi-structured data file representation (XML). The schema of this XML file is composed by two types of nodes, the nodes to declare atomic items ("element" node) and the nodes to declare composed items ("composedElement" node). The first type of nodes has the value of the item and no attributes. In the case of the composed type of nodes, the parent node as an attribute, called "id", which identifies the node and, at the same time, is the name of the composed item. As children of this kind of nodes we could have atomic items ("value" node) or other composed elements ("composedValue" node).
As example, consider a set of items where the items with the value 1 and 4 have no parent node (besides the top node), and 4 composed items with values A, D, C, B. As well as composed items contain atomic ones, the composed items A and B contains other composed items where composed item A contains the composed item D, and B contains the composed item A (and by inference the composed item D). A tree representation of this example could be seen in the figure 3.4, and the respective XML code next.

```xml
<taxonomy>
<!-- atomic element -->
<element>1</element>
<element>4</element>

<!-- a composed element -->
<composedElement id="A">
    <composedValue>D</composedValue>
    <value>2</value>
    <value>5</value>
    <value>6</value>
</composedElement>

<composedElement id="D">
    <value>12</value>
</composedElement>

<composedElement id="C">
    <value>12</value>
</composedElement>
</taxonomy>
```
In the following chapters we will explain, in more detail, the OntoC4S ontology and the SeqD2PrefixGrowth algorithm.
4 OntoC4S: Constraints Ontology for SPM

With the goal of trying to resolve the issues pointed before, we developed the OntoC4S. The OntoC4S is an ontology to represent constraints through the specification of sequential and concurrent events and their relations. It is integrated in the D²PM framework [Antunes, 2011], a framework that has the goal of supporting the process of pattern mining with the use of domain knowledge, and encloses the full range of pattern mining methods, from transactional to sequential and structured pattern mining. With the use of this ontology, users are able to easily and more expressively define constraints to be used in pattern mining algorithms as weren’t possible before. For this ontology, we considered a restriction as being a composition of relations, in which we say that a sequence accepts the restriction if it accepts all the relations of the restriction.

To create the OntoC4S ontology, we analyzed some ontologies and languages independent of the application domain they were made but following the features we referenced before. The Process Specification Language [Schlenoff et al., 1999] was identified as the most promising instance of that ontology and took by reference in the design of the new ontology.

This chapter is organized as follows: introduces the ontology that was used as the base for OntoC4S ontology, the PSL ontology in section 4.1; explain in detail the OntoC4S ontology, section 4.2; demonstrate how the taxonomy is integrated in the ontology (4.3); and ends by giving some practical examples of the ontology (section 4.4).

4.1 PSL (Process Specification Language)

The Process Specification Language (PSL) [Schlenoff et al., 1999] is an ontology developed by the National Institute of Standards and Technology (NIST) with the goal of providing a domain and tool neutral representation (formally describing concepts, along with their properties and relationships) of manufacturing processes. PSL has the advantage of providing a formally defined semantics and allows for multiple syntaxes; it is intended to be a very general ontology for manufacturing processes and the typical approach to specify processes consists of imposing restrictions on configurations that describe valid scenarios associated with the world under modeling.

The PSL ontology is composed of PSL Core and a partially ordered set of extensions. The ontology is structured as a set of theories in first-order logic, whose axioms and definitions are written in Common Logic Interchange Format (CLIF) [ISO24707:2007, 2007], a formal language developed based on Knowledge Interface Format (KIF) [Genesereth et al., 1992] but with different goals from this one. Meanwhile KIF was intended to stay as a common target notation to which other languages will translate to, CLIF was intended to be used for information interchange without requiring any kind of translation to be done.

Resuming, the PSL ontology is organized as a layer of theories composed by:

- **PSL Core**: the most basic elements of the PSL ontology;
- **Core theories**: extensions to **PSL Core** with widely application;
- **Extensions**: definitions of process terminology for different applications, based on the **Core**.
The purpose of *PSL Core* is to axiomatize a set of intuitive semantic primitives that is adequate for describing the fundamental manufacturing concepts. Concretely, the core ontology consists of four concepts: activities, activity occurrences, time points, and objects. Activities define a class or type of action; activity occurrences represent an action or event that happens at a specific place and time that corresponds to an occurrence of an activity; time point is the instant between two states; and object represents anything that is not one of the formers. An example can be seen in the figure below. All these classes are disjoint to each other.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Departure of a plane in an airport</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity occurrence</td>
<td>A plane departing from Lisbon Airport at 2PM on August 4, 2013</td>
</tr>
<tr>
<td>Time point</td>
<td>The point at which the plane starts the track but before it lifts off</td>
</tr>
<tr>
<td>Object</td>
<td>The plane</td>
</tr>
</tbody>
</table>

*Table 4.1:* Example of a plane departing through the key concepts of the *PSL Core.*

*PSL Core* is incomplete to provide definitions of the many auxiliary notions that are needed to describe all the intuitions of the domain. To fulfill the concepts defined by the *PSL Core*, the ontology offers a set of extensions. All extensions within PSL are consistent extensions of the *PSL Core* and can be consistent extensions of other extensions but not all extensions need be mutually consistent or conservative of other core theories.

A particular set of theories was created for axiomatizing all other concepts in the PSL ontology, creating the outer core (figure 4.1). Namely, these theories are:

- **Occurrence Trees:** an occurrence tree denotes the set of all discrete sequences of activity occurrences, with a unique initial activity occurrence. Concretely, it represents all possible occurrences of all activities in the domain rather than a particular occurrence of an activity;

- **Discrete States:** introduces the notion of (discrete) state. A change of states is achieved by the occurrence of activities. Activities have preconditions (states that must be true before an occurrence) and effects (states that turn true after an occurrence);

- **Subactivities:** this core theory axiomatizes intuitions about subactivities. The only constraint imposed within this theory is that the subactivity relation is isomorphic to a discrete partial ordering;

- **Atomic Activities:** atomic activities axiomatizes intuitions about the concurrent aggregation of primitive activities;

- **Complex Activities:** this core theory provides the foundation for representing and reasoning about complex activities and the relationship between occurrences of an activity and occurrences of its subactivities;

- **Activity Occurrences:** the complex-activities theory only axiomatizes constraints on atomic subactivity occurrences.
To illustrate the sequential and concurrent PSL virtues, as an example, consider the nursery protocol that stands for the injection of an oncology patient. The protocol is constituted by 5 activities: tourniquet application; disinfectant application; disinfectant dry; catheter insertion; and patient injection. If we understand the activities as being sequential the UML ([Unified Modeling Language, version 2] [Bock, 2003] in the figure 4.2 is an illustration of the protocol.

Assuming that the patient injection protocol is an activity composed by the subactivities stated above, the corresponding PSL (written in KIF language) representation is:

```
-- for all the occurrences of the activity PatientInjectionProtocol
(forall (?occPatientInjectionProtocol)
 (implies
  (exists (?occTourniquetApplication ?occDisinfectantApplication ?occDisinfectantDry
                  ?occCatheterInsertion ?occPatientInjection)
   -- implies the occurrence of the subactivities
   (and (occurrence_of ?occTourniquetApplication TourniquetApplication)
        (occurrence_of ?occDisinfectantApplication DisinfectantApplication)
        (occurrence_of ?occDisinfectantDry DisinfectantDry)
        (occurrence_of ?occCatheterInsertion CatheterInsertion)
        (occurrence_of ?occPatientInjection PatientInjection)
        (subactivity_occurrence ?occTourniquetApplication
                                   ?occPatientInjectionProtocol)
   )))
```
(subactivity_occurrence ?occDisinfectantDry ?occPatientInjectionProtocol)

-- imposing the sequential order of the subactivities
(next_subocc ?occTourniquetApplication ?occDisinfectantApplication PatientInjectionProtocol)
(next_subocc ?occDisinfectantApplication ?occDisinfectantDry PatientInjectionProtocol)
(next_subocc ?occDisinfectantDry ?occCatheterInsertion PatientInjectionProtocol)

-- start and end subactivities
(leaf_occ ?occPatientInjection ?occPatientInjectionProtocol))))

The PSL code starts to impose that for all the occurrences of the activity PatientInjectionProtocol that besides the occurrence of activity, implies the happening of the occurrence of activities composing the activity. After imposing the relation of suboccurrence of all the subactivities, the sequential order of this activity is imposed by the keyword next_subocc(?occ1 ?occ2 activity) which says that ?occ2 occurs immediately after ?occ1 in activity. The last lines define the TourniquetApplication as the first subactivity to happen and PatientInjection as the final state of the PatientInjectionProtocol activity.

To add a concurrent feature to this last example, lets assume that the subactivity Catheter Insertion and Disinfectant Dry can be executed at the same time, as shown in the UML of the figure 4.3.

![Figure 4.3: UML representing a concurrent nursery protocol.](image)

The correspondent PSL code to represent nursery protocol with the sequential and concurrent features is:

```plaintext
-- for all the occurrences of the activity PatientInjectionProtocol
(forall (?occPatientInjectionProtocol)
  (implies
    (exists (?occTourniquetApplication ?occDisinfectantApplication ?occDisinfectantDry ?occPatientInjection)
      -- implies the occurrence of the subactivities
      (and (occurrence_of ?occTourniquetApplication TourniquetApplication)
```
The code is similar with the previous example except the conc keyword that imposes the subactivity Catheter Insertion to be concurrent of the Disinfectant Dry.

4.2 OntoC4S

The development of the ontology had some steps. The first approach to the problem was the most simplistic. For that, we consider a connection between the class Constraint and Item in which the relation type is defined by the attributed property restrictions. As an example modeled by this approach, we can define that an instance of the class Constraint (C1) accepts all the sequences which have a as the initial item (C1 hasBegin with a).

![Figure 4.4: First approach to the ontology with a simple relation.](image)

This approach has a huge drawback. In the case of a relation that connects two or more items, this approach is ambiguous. In a binary relation we just need with two concepts and a property linking them, in this case, the constraint and the item related, but when the relation is between two items there isn’t an unambiguous way of relating the relation with the constraint. If we have a restriction with more than one ternary relation, it is difficult to say which item is related with each other. One example of such a ternary property constraint is the hasPrecedent that forces an item (the precedent) to be before than other item
(the consequent). In this first approach, we could have defined two property relations that illustrate an antecedent item and the consequent to this first. Let's say we could have the property hasAntecedent and hasConsequent in which the antecedent item is related with the first relation and the precedent item is related with the second one, and illustrates that the first item comes before the second. But in the case we have two or more hasConsequent which item belongs to each relation isn’t obvious.

This took us to a new stage of the ontology where we developed a solution to this issue. We incorporate the notion of reification. Reification stands for a technique that enables the representation of n-ary relations using a language that only enables unary (classes) or binary (properties), such as OWL. To achieve this goal, the n-ary relation is described as a new object that incorporates all the arguments of the n-ary relation as objects of properties. As an example, if we consider the relation Rel holding between i1 and i2 on the relation Rel holding between i1 and i2 on the constraint C2, in OWL language, this is asserted as a new object with Rel, i1, i2 and C2 being constituted as objects of properties. Applying this approach to our issue, we define the relation that holds the constraint and the items as a new object Relation. In the example of the relation that forces an order between the items, a constraint is related with the relation object Precedent that holds the relations with the interesting objects, the precedent and the consequent items. Additionally, we can define some interesting properties as a maximum gap between the items that forces that the antecedent item can hold a maximum number of items (in the value of the gap) until the consequent one.

As we stated before, a constraint, in this context, is a composition of relations where a sequence is accepted if respect all the relations that composed it, so is possible that a constraint has more than one relation. We can say that a relation that forces an item to start the sequence, a relation of precedence between two items and a final item closing the sequence compose a constraint. In the former approach, this notion of composition between relations in a constraint isn’t well defined. Let's assume that the relations have an order, which means, a relation occurs before, after or concurrent to another relation in the restriction. As an example, we can say that a restriction forces an item a to be the first item of the accepted sequence, then item b occurs before the item c in a distance of 2, the item e occurs next to the item d and the final item of the sequence should be a f, and all these relations are verified in this order. So this restriction accepts the sequences <abcdef>, as well as the sequence <abgcdef> and <agihbcdef> but don’t accept the sequence <bcadef> or <abcdcf> (this last were accepted only if the first precedence relation were concurrent with the second).

These requirements take us to an ontology proposal where the relations of a constraint hold properties between and we define a new object to specify the attributes of that connection. This new attribute was named RelationSequence and could define the various aspects including the distance between the relations or forces them to be sequential or concurrent. The relations Begin and End have a fixed order (first and last relation of the restriction) and, if they exist, their cardinality is, naturally, resumed to one.
4.3 Taxonomy integration

In order to integrate the taxonomy explained in chapter 3.5, the OntoC4S ontology has the entities and the properties needed to represent the concepts identified before. The entities Element and ComposedElement represent the atomic items and the composed items, respectively. Unlike the taxonomy declaration in the XML file, the taxonomy relations of this two concepts represents an Element as the parent entity of the ComposedElement entity in order to declare that a composed item is itself an item (as can be seen in the figure 4.7).

Figure 4.6: OntoC4S architecture.

To declare the relation between these concepts the containsElement and isContainedBy properties were created. These properties have an inverse relation between each other, that is, the entities of the domain of one are the entities of range of the other and vice-versa. In this case, ComposedElement contains elements (Element entity) - property containsElement with domain ComposedElement and range Element (could have as many elements as it is desired), and Element is contained by one composed

Figure 4.7: Taxonomy relations in the OntoC4S ontology.
element (ComposedElement) - property isContainedBy with domain Element and range ComposedElement. To represent the element that contains all the other, as a parent node or a root for all the other nodes, was created a ComposedElement called "TopElement". This instance will be useful, specially when the algorithm reads the ontology to instantiate its meaning in a object oriented representation. As a final note, the recursive reference of concepts is not allowed due to the resulting complexity and the violation of the taxonomy structure.

To finish the overview of the ontology, we must now update the figure 4.6 by adding the concepts related with the taxonomy (figure 4.8).

4.4 Practical Examples

To illustrate the use of this ontology, let's consider an example in the healthcare domain. Consider the nursery protocol that stands for the injection of an oncology patient. The protocol is constituted by 5 activities: tourniquet application; disinfectant application; disinfectant dry; catheter insertion; and patient injection. If we understand the activities as being sequential the UML (Unified Modeling Language, version 2) [Bock, 2003] in the figure 4.9 is an illustration of the protocol.
To adapt to the ontology representation, we are considering a gap between the activities. Concretely, these gaps could represent, in a constraint, the number of items that are between the items of the consequent relations.

As can be seen in the figure 4.10, all the states and the order between the activities are explicit in the ontoC4S representation. The restriction starts by expressing the restriction instance with the first relation object, in which is said that the item tourniquet application is the first of the accepted sequence. Then, the order between the items is explicit and the gap inter-relation is defined in the object that relates the relations.

In this case we chose to use the relation Precedence to express the order of the items but other relation
objects could use. Consider the case where two relations are concurrent, i.e. they happen simultaneously. This example forces the item *catheter insertion* to happen when the *disinfectant dry* is executed (and vice-versa). The difference between the *ontoC4S* representation of the sequential sequence and this concurrent version is the way concurrent relations are connected.

![Diagram](image)

**Figure 4.11:** Sequence of a nursery protocol with concurrency and the gaps between the activities.

In this case, the connection between the Precedence relations isn’t sequential but concurrent, i.e. these two relations are happening at the same time. In the context of a constraint, this means that the algorithm is validating the two relations (predicates) at the same time and not one after another as happens in the sequential instance. One particularity of this concurrent example is that there is a gap (+1) with the relation, which happens before these concurrent relations. This aspect forces both relations to share the same gap. These examples were an illustration of the potentiality of the OntoC4S. More complex domains could be modeled by this ontology and, unlike other approaches, the user deal with a friendlier interface to define the constraints according to his expectations.
Figure 4.12: Example of the Figure 4.11 through the ontology ontoC4S.
SeqD2PrefixGrowth: Constraints Algorithm for SPM

The SeqD2PrefixGrowth algorithm is an algorithm created with the goal of incorporating constraints provided by an ontology in the process of sequential pattern mining. The ontology used to provide the constraints is the OntoC4S ontology introduced in chapter 4. The OntoC4S ontology has the capacity of represent constraints over sequences with sequential or/and concurrential events. As stated in the ontology’s chapter, beyond allowing the representation of this two kind of events, the ontology offers the facility of incorporate gap between the constraints enabling the relaxation of the constraints with impact over the result set of frequent patterns (while respecting the constraints). At the same time, there is the objective of making the extraction’s process of patterns over rules more efficient and effective in terms of memory space and I/O computation.

This algorithm is based on the GenPrefixGrowth algorithm [Antunes, 2005], an extension of the GenPrefixSpan [Antunes and Oliveira, 2003] algorithm (GenPrefixGrowth is based on the PrefixGrowth algorithm [Pei et al., 2002]). GenPrefixSpan algorithm follows the PrefixSpan approach (the build of projected databases is redefined when compared with the former approaches), with the special feature of enabling the existence of gaps when counting the alphabet elements in the sequences from where the frequent patterns will be extracted from (in the PrefixSpan the elements that matter when the count is being done are the first elements after the one that was processed). As stated before, the PrefixGrowth is an algorithm which follows a strategy adopted by the PrefixSpan [Han et al., 2001], allowing the incorporation of prefix-monotone constraints in the mining process (enabling the possibility of the use of monototic and anti-monotonic constraints, e.g. regular expressions). The possibility of incorporation of a constraint in the mining process enable the return of sequences where only the sequences that satisfies the constraint are returned (only prefixes of sequences that respect the constraint are accepted). The principal reason for the creation of this algorithm was not only the possibility of the use of context-free languages (ePDAs) but the possibility of the use of a class of constraints. The core of the SeqD2PrefixGrowth algorithm is done over the GenPrefixSpan besides the fact that was profoundly inspired in the GenPrefixGrowth algorithm.

In the following sections the algorithm flow will be explained, since the read of the ontology (section 5.1) until the return of the final set of frequent patterns that respect the constraint sequence and the taxonomy relations defined by the user (section 5.1.2).

5.1 OntoC4S ontology read and mining algorithm execution

The algorithm starts after the ontology finishes the reading of the inputs (constraint sequence XML file and the taxonomy XML file) and represents the knowledge in its representation model. Only when the algorithm takes into scene the dataset is taken in account. The algorithm has 2 different phases:

- **first phase**: the read of the ontology’s knowledge and the representation of it in a object oriented structure (using Java object oriented programming language);

- **second phase**: sequence pattern mining algorithm execution.

Next, we will detail about the first phase. Posteriorly, the other phase will be showed.
5.1.1 Algorithm's first phase: ontology processing

The first part of the ontology read is the instantiation of the taxonomy in a class structure designed to accommodate it. To able to do it, a set of classes and relations among them were created. Translating directly the entities in the taxonomy, we created the classes **ComposedElement** and **Element**. As expected, the first represents the instances of the taxonomy entity with the same name (i.e. represents the entity that encapsulates the concept notion of a composed element which could have other elements), and the second class has the same pattern (represents the notion of an simple element, the leaf of the taxonomy tree). Besides this classes, other was created in order to enable an easy manipulation and access to the elements and walk through the taxonomy tree, when reading, writing or validating the items. This class was called **AbstractElement**, and, as the name suggests, is an abstract **Java** class. The complete view of these three related classes is illustrated in the figure below.

![Diagram](attachment:image.png)

**Figure 5.1**: Taxonomy Java classes in UML modelling language notation.

The extraction's process of the taxonomy's knowledge from the ontology starts with the creation of a top parent element (or node), likewise it was done when instantiating the taxonomy in the ontology. This parent element, called "**TopElement**", is the root from where the building of the other elements instances will start. In order to do it, the algorithm search all the elements in the ontology that are connected to the top element of the ontology through the property **containsElement** (as shown in the **OntoC4S** explanation - chapter 4). The elements found as children of the top element, are instantiating as being following the parent node. As expected, before instantiating the child element it is needed to determine if the element is an simple element or a composed one. For that, the procedure is equal to the search of the children of the top element, that is, the algorithm search if this node has the property relation of containing other(s) element(s) (**containsElement** property). If yes, it assumes the element has being a composed element, if no it is instantiated as a simple element. This process is a deep-first process because as long as a node is a composed element it will search into its children instead of following a breadth search method of exploring the sibling nodes before searching the node space vertically. As an example, consider the taxonomy represented in the **XML** structure below:

```xml
<taxonomy>
  <composedElement id="A"> <!-- a composed element -->
    <composedValue>B</composedValue>
    <value>5</value>
  </composedElement>
  <composedElement id="B"> <!-- a composed element -->
    ...<value>40</value>...
  </composedElement>
</taxonomy>
```
As can be seen in this XML code, the top element (which is not represented in this taxonomy definition) has 5 children nodes: A, B, C, 1, and 2. The first 3 elements are composed items while the others are simple ones. Following the deep-first approach, the algorithm will read the top element, then find the children of this node. The first node to find is the item A. After discovering this node, it will search possible children of this one. It will discover the item B and will repeat the process till the discover of the simple item 7. As this last item is a leaf of the taxonomy tree, the algorithm will explore the siblings and will up the tree till A where it will find other simple element (items 5). The process will end when the node at the rightmost and more deep is discovered.

After the instantiation of the taxonomy (which is present in the OntoC4S ontology instance) the algorithm will process the constraint sequence present in the ontology in order to incorporated them in the mining process. This phase will translate the constraints into rules. The rule is the concept adopted to guide the mining process to return only the interesting frequent patterns that the user declared by defining the constraint sequence XML file.

As it was done to read the taxonomy, the algorithm will walk through the constraint sequence and
instantiating the elements when they are found. To start the read process, the algorithm will search for the first entity of the sequence, an instance of the entity `ConstraintComposition`. The goal of this entity, besides the fact of being the first element of the constraint sequence, is to define the start conditions of the sequence in the ontology, concretely the gap that precedes the first constraint. If we think about a constraint that explicit the start of the accepted sequence with an item there is no gap but when other kind of constraints are defined first this start gap could be present by this entity. As it will be explained in the next pages, this start gap will be present in the first rule of the rules that represent the constraint sequence. After the read of this entity, the algorithm will continue the search by identifying the constraint entity that is related with this first entity by using the property `nextContraint`. When the constraint is identified, according to its kind, it will be instantiated in rules. The number of rules depends on its complexity (e.g. the `precedence` constraint is represented in two rules while an `exist` will return one rule). The concept of rule is represent by its own Java class. The attributes that characterizes a rule are the same independent of the kind of constraint it will represent. In the next list they are identified and explained:

- **Value**: identifies the item that the frequent pattern should have to respect the constraint (or part of it);
- **RuleOrder**: the number of the rule. This number represents the order in which the rule were defined;
- **ConstraintOrder**: the number of the constraint being processed. This number could be different of the rule order due to the possibility of a constraint to be represent by more than one rule;
- **Gap**: the maximum gap (or itemsets) allowed to the discovery of a situation which respects the rule;
- **isParallel**: a binary possible value that indicated if this rule should be parallel with other previous rule (in order to be possible to represent parallel constraints), i.e. must be present in the same itemset.

These attributes are sufficient to represent and make possible to accomplish all the features that this framework was created and intended to respect. As said before, the number of rules depend on the complexity of the constraint to be represented. To illustrate this fact, consider the `exist` constraint. This constraint says that an identified item (simple or complex) must be present. By considering the attributes above, it is possible to say that they are sufficient to represent this constraint with just one rule. The figure 5.3 is a metaphoric figure to illustrate the `exist` rule.

![Diagram](image-url)

**Figure 5.3**: Rule's representation of the `exist` constraint.
To compose this situation, consider the *precedence* constraint. This constraint forces an item $X$ to be antecedent to other item $Y$. In the figure 5.4 it is possible to verify the of two rules instead of one. The presence of a constraint which relates two items forces the existence of two rules even if the fact of a rule only enable the identification of one item. As in the case of the *exists* constraint, the item could be simple or complex. The figure shows us the relation of the two rules and their attributes. The gap of the first rule identifies the gap before the first rule and the gap before the constraint as a whole while the gap of the second rule indicates the maximum possible itemsets between these items. In the case of these items being parallel, i.e. they coexist in the same itemset, the gap (itemsets between this items) is forced to be equally to be zero, as it is expected. With these two rules is possible the mapping of this complex constraint.

The *begin* constraint is represented with one rule, as the *exists* constraint, but where the gap before is equal to zero. This constraint a specific situation of the *exists* constraints but due to their importance we decide to identify it as a primitive one.

![Figure 5.4: Rule’s representation of the *precedence* constraint.](image)

To close this topic of constraints user definition, ontology mapping and consequent rules we will do a complete example with these 3 phases. Consider the *XML* code below that defines a constraint sequence.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ConstraintDefinition>
  <constraintSet>
    <title>Completeconstraintexample</title>
    <note>Example to show all the steps when instantiating the constraints</note>
    <constraint id="1">
      <begin>B</begin>
    </constraint>
    <constraint id="2">
      <precedence gap="0" isParallel="true">
        <antecedent>9</antecedent>
        <consequent>2</consequent>
      </precedence>
    </constraint>
  </constraintSet>
</ConstraintDefinition>
```
As can be seen in the code above, this user-defined constraint sequence says that the begin item should have the value B. If we consider the taxonomy from the figure 5.2, the user is forcing the first item to be equal to B or 7. As it is easy to understand, if the constraint says that the sequence must start with a specific item so the gap before this constraint needs to be equal to zero (that fact can be seen in the "constraintSequence" where the type as the value start). After this constraint, the user wants that in a space of at maximum 3 itemsets a precedence constraint must be present where the item 9 must precede the item 2. The user specified that this constraint is parallel, that means, the items must be present in the same itemset. This constraint's specifications have a particularity. If the item 9 must precede the item 2 that will not happen in this case due to the fact that they both are in the same itemset. In an itemset, the items are ordered by their value in an ascending order. As 9 is greater than 2, the desirable antecedent item will never before the desirable consequent item. In this case the SeqD2PrefixGrowth will not take in account the effective order in the itemset and the constraint will be validated if both are present in the same itemset (and before the gap defined). At least, the user want that the item A (or some element of the family) must be present at a maximum gap of 1.

The translation of this user-defined constraint sequence to the ontology is illustrated in the figure 5.5. It is very simple to observe the constraints of the XML in the ontology of the figure. The ontology sequence starts with the ConstraintComposition (in order to define the initial gap, in this case the gap has the value 0). Next, the begin constraint is present before the next constraint precedence and the exists. Between the constraints we can observe the entities that specify the connection between two consecutive constraints.
To end the process, this ontology instantiation will be processed and translated into rules. In table 5.1 we can see the correspondent rules for the constraint sequence of the ontology. The result rules translation represent what is sufficient and needed to map the constraint defined by the user in the mining process and force the extracted frequent patterns to respect them. As it was expected, the number of rules correspond to each constraint and the precedence constraint is translated into two rules due to the fact the relates two items.

<table>
<thead>
<tr>
<th>Rule #</th>
<th>Value</th>
<th>Rule order</th>
<th>Constraint order</th>
<th>Gap</th>
<th>isParallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>false</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>false</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>true</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>false</td>
</tr>
</tbody>
</table>

Table 5.1: Rules correspondent to the complete example.

To test these rules, consider the sequence ((7 3 4) 1 5 (2 9) 6 5) that will be tested in the rules of the example. This sequence has 6 itemsets and 9 items (itemsets number 1 and 4 have more than one item). The validation of this sequence can be seen in the next table:

This sequence is valid according to the rules. That is easily verified by the last state of the table with the value “All pass”. This sequence makes use of the taxonomy when the item 7 is an item from
<table>
<thead>
<tr>
<th>Input</th>
<th>Itemset</th>
<th>Gap</th>
<th>State</th>
<th>Rule #</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>Pass</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>Between constraints</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>Between constraints</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>Between constraints</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
<td>Between constraints</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>Pass/Inside constraint</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>0</td>
<td>Pass</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>1</td>
<td>Between constraints</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>0</td>
<td>Pass / All pass</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 5.2: Validation of a sequence over the result rules of the example.

the composed item $B$ and the item 5 that belong to the composed item $A$. The table is organized as follows: the "input" column designates all the items of the sequence (one item per row); the "itemset" relates the item with the itemset it belong to, the "gap" is the actual gap of the input's sequence in the validation of the actual rule (this gap must respect the maximum gap presented in the rules); "state" is the state of the input's sequence where the possible values are "Pass" when a rule was passed with success, "Between constraints" when the input is looking to verify the actual rule, "Inside constraints" when the input is in the middle of a constraint (for constraints with more than one rule, as the case of the precedence constraint), and "All pass" that corresponds to the case where all the rules were passed; "Rule #" indicates the actual rule; and "Constraint" which indicates the actual constraint. This example uses a sequence which accomplish all the restrictions of the rules and as result it is a valid sequence over the rules defined by the user.

5.1.2 Algorithm’s second phase: sequence pattern mining process execution

The SeqD2PrefixGrowth is an algorithm that follows the prefixSpan approach and incorporates the features that the algorithms identified above add to this original approach, that is the possibility of gap between the last element processed and the other elements that could extend the pattern in the count phase before expand the actual pattern, and the application of constraints that make possibility the use of knowledge in the mining process through the richness of the creation of an ontology which contextualizes the constraint set.

After the phases described before were executed, it is time for the sequential pattern mining algorithm to run. As it is easy to understand, the phases before occurred first because the output they return are directly or indirectly needed for the algorithm to run, i.e. the taxonomy and the rules. Besides these inputs to the algorithm, as a follower of the GenPrefixSpan algorithm it will receive the maximum distance allowed when counting the elements for the generation of candidate patterns, the minimum support for a sequence to be considered a frequent pattern (as in all pattern mining algorithms), and the data set itself from where the frequent patterns will be extracted from. The signature of the algorithm method can be seen in the box below.

```java
void SeqD2PrefixGrowth(double minSup, SeqDataset db, int distance, boolean on,
                       Vector<SeqD2Rule> rules, ComposedElement topElement)
```

The arguments, in order, from this method are: minSup is the minimum support, the db container is the dataset, the boolean indicates if we are running the algorithm in debug mode or not (debug mode print some more information to the screen in order to verify some intermediary values for the algorithm
execution), a vector with the rules extracted from the ontology (this process of rules extraction were explained in the section 5.1.1), and the top element of the ontology in order to be able to walk the taxonomy when verifying the items composition.

The methodology of the generation of candidate frequent patterns is similar to the way GenPrefixSpan does it. The first step is the discovery of the 1-itemsets. To do it, the algorithm counts each of the elements in the alphabet of the dataset in each sequence. At most, the same element counts by appearing in a sequence independent of how many times in the sequence. Then, by the order of the support of the elements (to be more efficient), the algorithm validates that the 1-itemset candidate pattern has a support greater or equal to the minimum support value (for a sequence to be considered frequent pattern in a unconstraint algorithm, the elements that build the sequence just must have a support number above or equal to the minimum support value) and if this first item respects the rules (in this case the first rule). If the 1-itemset validates the rule then other elements could be appended to this itemset in order to verify if they are frequent patterns, if not the itemset is discarded.

When an item is identified as being frequent and is created an frequent itemset with it, the algorithm builds a projected database (α-projected database) with this item as the prefix. Conceptually, a projected database is a subset of the original dataset with the sequences that have this frequent itemset as a prefix, and these sequences are truncated until this prefixes in order to be more efficient to search for new items to append. The implementation of this algorithm follows the strategy of GenPrefixSpan implementation and to not expand the memory required to run the algorithm, it just annotates the index of the sequences and the index of each sequence where to search from.

After the verification of the 1-itemsets and the creation of the respective projected database, the algorithm will enter in a recursive method in order to increment the size of the patterns. This algorithm follows a depth-first search, i.e. will increase an frequent pattern till that sequence couldn’t be more increased. Consider this example, the first set of 1-itemsets are constituted by the itemsets {a,b,c,d,e}, the algorithm will then take the first element (the 1-itemset a) and try to append other items of the alphabet. Consider that the frequent 1-itemset with the prefix a (or as used in this algorithm, α-prefix a) will be processed. It will then find other 2-itemsets. If one of the 2-itemsets is the {a,b} itemset, the algorithm will try to find 3-itemsets with this prefix before try to expand other 1-itemset with different prefix than with the ones with prefix a. This behavior will be respected till there are no more prefixes to expand.

Each iteration of this method code is responsible for an increment of one item in the actual frequent patters. This cycle needs to be in account some contextual situations, the α that defines the prefix from where the new element will be appended, the size of this α (number of items which build the actual pattern), the elements of the alphabet in order to know what are the element it will search and append, and the projected database of this prefix (built before). The method's signature can be see in the box below.

Vector<SeqSequence> runRecursively(SeqSequence alfa, int alfaSize, Vector<SeqItem> alphabet, SeqProjectedDB db)

As can be seen, the output of this method is the set (using a vector structure) of all frequent patterns discovered that, besides respecting the rules and the minimum support, are suffixes of the prefix indicated in the method's signature (argument "alpha").

This method is very important to the algorithm because it is the root that enables the patterns build
process. When it is called, it has to perform the count of the items in the projected database. This count is made for sequential and parallel items in order to be able to expand items that belong to the same or different itemsets as the last element of the prefix. After this count is done, it searches for candidates in parallel or concurrent situation. As done in the case of the 1-itemsets, the candidates are then verified and if they respect the rules and minimum support they are appended to the prefix to form a new frequent pattern.

Other important phase of this method is the verification of the compliance of the candidate itemsets over the rules created from the ontology. When a new candidate is received, it's verified with the actual rule. To compare a rule with the candidate item, the algorithm verifies if the itemset's item order in the sequence is inside the allowed gap, and if it is, verifies if the item is equal or belong to the item (in case of a composed item) indicated by the rule. When the item doesn't verify the rule, the candidate is automatically discarded. When the candidate respects the rule it could accomplish the rule or be still in the allowed gap before the items' rule must be present. In the case of just being in the allowed gap, the rule maintains the same and the gap is updated. If the candidate item is equal to the rule's item or belongs to the same taxonomy tree, a new rule is loaded from the processed rules. The process is almost the same for parallel or sequential rules (constraints) where the differences reside in the increase or no of the gap when an item isn't the same of the rule's one (in case of the parallel cases, the gap distance isn't changed because we are searching for items in the same itemset). Each pattern verification is accompanied by a state that identifies uniquely the context of that pattern in that specific moment. The attributes of the state needed to identify the context are:

- **Max allowed itemset**: it is the sum of the last passed rule itemset with the maximum gap allowed in the next rule. The actual candidate item's itemset must be have the itemset order less or equal to this number in order to be accepted.

- **Rule order**: the order number in which the rule were defined;

- **Restriction order**: the order of the constraint being processed and which this rule belongs to. This number could be different of the rule order due to the possibility of a constraint to be represent by more than one rule;

- **Current itemset**: the order number of the current item's itemset;

- **Start rule itemset order**: the first itemset to consider when verifying the constraint (when items of an accepted item's itemset are being caught);

- **Item found in itemset**: identifies the item's itemset order number of the accepted;

- **Validation state**: describe the state of the validation in the actual context. The possible values are: no rules, inside, pass, all passed. A "no rules" state indicates that there are no rules to verify (when a constraint sequence was not defined); "inside" state represents the case when we are dealing with constraints that have more than one rule and when we are at the middle of the rules we are in this state (e.g. precedence constraint); "pass" indicates that a rule was verified (the item or composed item searched was found); and "all pass" happens when all the rules were verified (in this case there is no more patterns expansion).

In overview, the mining process is composed by some concepts beyond the main procedure of the process. The algorithm entity is the concept class that coordinates the process but the features don't just stand on it. Other concepts were created in order to enable the implementation of the algorithm features in order to ease the management and flexibility of the process. The concept of constraint in
an class structure enables the management of the validation and verification of the constraints and the discovery of the projected databases and the count of the elements in the same projected databases or/and initial dataset. The governance of the rules and the rules sequence is responsibility of a concept called "tree". Besides these actions, this concept is responsible for the mining pattern state and the update of it. This name "tree" is derived from the possibility, in next iterations of the project, of having not just one rule sequence (processed from the constraint sequence) but more (from more than one constraint sequence) and other possible relaxations that could be done over the "tree" nodes. The rule itself is another concept with responsibilities in verifying the conformity of the input item with the rule according to the respect of the gap limit or the familiarity between the rule's item and the input one. This concept is called from the manager concept (the "tree"). As stated before, the mining pattern sequence has a context represented by the concept "state" explained above. The concept which represents the actual pattern is the "sequence". The meaning of this concept is the same of its name and is composed by itemsets. Itemsets have items which represents the concept that is being tested each time over the rule.

![Diagram](https://via.placeholder.com/150)

**Figure 5.6: SeqD2PrefixGrowth algorithm mining process overview.**

In the figure 5.6 it is presented an overview of the algorithm mining process, where the initial dataset (SeqDataset) is the input of the algorithm (after the taxonomy and the constrains ontology were processed) and from where the patterns will be extracted from, to the discovery of candidate sequences that are sent to the constraint (SeqD2Constraint) that with the rules manager (SeqD2Tree) will channel to the rule concept which will deliberate and output the respect comparison of the candidate sequence over the actual rule, updating the state (SeqD2State) and calculating the projected database of this new
pattern sequence (if it fulfills the rule).

This algorithm's organization allows an effective and sufficient process mining focused in the goals of extracting the interesting rules embodied by the user while combining the flexibility of the constraints and the taxonomy inserted. In the next chapter, will present some results related with the performance of the algorithm (and the framework) and show how the algorithm manage its resources in order to accomplish with the mail goals.
6 Evaluation

The comparison of sequential pattern mining algorithms over a large range of data characteristics, such as different support thresholds, dataset sizes and sequence lengths has been done by several authors (see for instance [Agrawal et al., 1996], [Srikant and Agrawal, 1996], [Zaki, 1998], [Han et al., 2001] and [Ayres et al., 2002]), nonetheless the results are sensible on the dataset density. For the next evaluations we use a dense dataset with 10000 sequences and a fixed support value of 10%.

Our goal in this section is to measure and understand the impact of those characteristics in the performance of the algorithms (execution's time and consumed memory). In order to do that, we will analyze the algorithm's performance, compare some performance aspects with of GenPrefixSpan and SPaRSe with the performance of SeqD2PrefixGrowth algorithm, and finally a comparison of the return set when executing the process of mining using the SeqD2PrefixGrowth on the same set without constraints and with constraints.

The basic methods used by the different algorithms were the same (basic operations on sequences, support verification, etc) which means that the comparisons are meaningful and repeatable. The GenPrefixSpan and SPaRSe follow the descriptions described in the chapter 2. The datasets were maintained in main memory during the processing, avoiding hard disk accesses.

To perform the study, we used the standard synthetic data set generator from IBM Almaden (described in Appendix A). The dataset used in these experiments has the following configuration:

- **Number of sequences (**$|D|$**):** 10000 sequences;
- **Average Number of itemsets per sequence (**$|C|$**):** 10 itemsets/transaction;
- **Average number of items per itemset (**$|T|$**):** 2 items;
- **Average length of maximal potentially frequent sequences (**$|S|$**):** 4 itemsets;
- **Average length of maximal potentially itemsets/transactions (**$|I|$**):** 2 items;
- **Number of maximal potentially frequent sequences (**$N_s$**):** 5000 sequences;
- **Number of maximal potentially frequent itemsets/transaction (**$N_i$**):** 10000 itemsets/transactions;
- **Number of items (**$N$**):** 10 items.

These values were chosen in order to follow closely the parameters usually chosen in other studies. This dataset is same that was used in [Antunes, 2005] work in order to make the results of this project and her work comparable.

In the next sections we present the performance results (time consumed and memory requirements) achieved by the SeqD2PrefixGrowth, followed by a comparison with GenPrefixSpan and the SPaRSe unconstraint algorithms.
6.1 *SeqD2PrefixGrowth* performance

In this section we will show some performance scores of the *SeqD2PrefixGrowth* algorithm. The goal is to verify the behavior pattern of the algorithm over different gap values (gap related to the counting of the elements). The figure 6.1 shows the performance algorithm in time when gap is equal to 0, 1 and 2.

![Figure 6.1: Execution time with different gap values.](image)

As can be seen in the chart, the execution time of the algorithm increases naturally when the gap increases. This happens because the candidate sequences increase with the gap value. It’s now important to know which is the part of the *SeqD2PrefixGrowth* algorithm where it wastes more time. Let’s first consider the 3 main phases of the algorithm: the process of the XML to the OntoC4S ontology ("XML process"); the mapping of the ontology to rules and the taxonomy to object instances ("Ontology process"); and finally, the sequential pattern mining process ("Mining process"). The figure 6.2 show us the greater timer consumer, the mining process as it was expected.

![Figure 6.2: Execution time by main algorithm phases.](image)

Let’s now understand which is the phase of the mining process that consumes more time. In the figure 6.3, it is represented the time wasted by the different phases of the algorithm in the mining process, "Projected DB" - projecting databases for each of the candidates; "Generate candidates" - candidates
generated over the projected databases for each sequences; "Verify constraints" - verifying the sequences over the rules generated from the ontology and the user defined inputs. This figure allow us to see that the new phase introduced by the SeqD2Constraint, the verification of the candidate sequences over the rule set, consumes insignificant time when compared with the generation of the candidates for each sequence (the phase that wastes more time in the mining process and in the all process of the algorithm and framework). The figure 6.4, give us the same information but comparing the executions time proportions for each mining phase.

![Figure 6.3: Execution time by algorithm mining phases.](image1)

![Figure 6.4: Execution time proportion by algorithm mining phases.](image2)

These results doesn’t differ from the unconstraint algorithm GenPrefixSpan due to the fact that SeqD2PrefixGrowth has is core in this algorithm and the constraint parts of the code are not so time consuming when compared with the methodology of the GenPrefixSpan in the generation of candidates and the projection of databases of the candidates. In the next section we do a direct comparison between the SeqD2PrefixGrowth and some unconstraint algorithms.
6.2 Performance comparison with unconstrained algorithms

The `SeqD2PrefixGrowth` is a constraint algorithm, i.e., an algorithm created to execute the process of sequential pattern mining using constraints that introduce user-defined knowledge in the algorithm in order to extract only interesting patterns and avoid a big and vague result set where decisions cannot be taken from. It’s important to understand if the algorithm didn’t lose the capacities of the unconstrained sequential pattern mining and the features that are shared with these algorithms, in specific with the `GenPrefixSpan`.

To be able to verify these features in the `SeqD2PrefixGrowth`, we compare some parameters of our algorithm with the algorithm from which it was done (`GenPrefixSpan`) and the `SPaRSe`, in section 2.3.4. These results were taken from the work done in [Antunes, 2005]. The values of the measures of the `SeqD2PrefixGrowth` and the `OntoC4S` framework were updated taking into account the differences between the machines from where the previous results were taken.

The first parameter we measured was the execution time of the algorithm. It is important that the `SeqD2PrefixGrowth` fits its time accordingly with these algorithms. In the bar chart of the figure 6.5 it is possible to see that this goal of keeping the same order of execution time was achieved. Despite of the values from the `SeqD2PrefixGrowth` are greater than the execution time of the `GenPrefixSpan` and the `SPaRSe` algorithms, the values are very similar and the impact is not much. The reason for this difference of values is the overhead of objects that are instantiated in the `SeqD2PrefixGrowth` algorithm to, as example, save the state of the sequence being processed in the rule tree (this happens for each processed sequence) and to instantiate the ontology. There is some time wasting in this process that the `GenPrefixSpan` and the `SPaRSe` algorithms don’t make use of. Besides this disparity, the number of sequences generate were very similar (equal when compared with the `GenPrefixSpan` algorithm).

![Figure 6.5: Execution time comparison with unconstrained algorithms.](image)

We have also analyzed the memory requirements of this algorithm when compared with the `GenPrefixSpan` and the `SPaRSe`. The figure 6.6 shows the bar chart with the comparison of the memory consume. As it was expected, the `GenPrefixSpan` needs more memory to execute than the `SPaRSe` due to its specific features (`GenPrefixSpan` algorithm has to maintain multiple indexes for the same sequence and the corresponding pattern position for each occurrence).
As stated in the figure, the GenPrefixSpan requires 5 times more memory than the GenPrefixSpan and the SPaRSe algorithms. As a remember, this result was not the effective result and the restrictions of the machine, where the GenPrefixSpan and the SPaRSe algorithms ran, were took in account and the final result updated. This large difference was expected since, as stated before, the GenPrefixSpan has to process and store the ontology and the taxonomy, and to save a state for each of the sequences being built. This means that if the algorithm is validating 10000 sequences it will have 10000 states (one state for each sequence).

Summing, they have similar performances if we have in mind the features of each algorithm, besides the sizable difference in the memory requirements. We can’t affirm that the algorithm is not likely to be optimized with the use of more efficient structures and constraints verification processes for each sequence analyzed, but in general, it accomplish the goals when compared with unconstrained algorithms. We need also to be conscious that there is a trade-off when we are adding complexity to an algorithm to add new features as the verification of constraints in patterns.

6.3 Patterns extraction from unconstraint vs constraint

We will now test the number of patterns returned when there is a constraint and when there is not using the SeqD2PrefixGrowth algorithm. This test will enable the understanding of the impact of getting information with knowledge and when the algorithm only has the support to restrict the final set. Concretely, one of our main goals is to create a more actionable set of patterns in order to be easy for the user to get new information. The constraint input is represented in the next XML code.

```xml
<ConstraintDefinition>
  <constraintSet>
    <title>EvaluationTest</title>
    <note>Evaluation test</note>
  </constraintSet>
</ConstraintDefinition>
```
We will test the algorithm with different values of support and gap values. It is expected that as far as the gap value increases the number of patterns increases too while the opposite variation is expected when the support increases.

Figure 6.7: Unconstraint vs Constraint when support = 10%.
As can be seen in the figures 6.7, 6.8, and 6.9 the constraint execution returned less patterns than the unconstraint one. We can conclude that the patterns that the user don’t want to see are a very relevant percentage of the amount of patterns returned in the unconstraint execution. It’s is now verified the need of the use of a filtering in the return set through the use of constraints in the algorithm. But besides of this user oriented returned set, there user must be aware of not bias the algorithm in a way where the algorithm is just a hypothesis validation and the user can’t get new information. This results show the need of incorporate knowledge in the \textit{sequential pattern mining} process to enable the extraction of new and interesting information and the success of this framework in to achieve that goal.
7 Conclusions and Future Work

The OntoC4S framework enables the process of getting frequent patterns according to user defined constraints. The user has the capacity of introduce knowledge in the process and get patterns oriented to his expectation and able to act over that information.

In this work, we studied the sequential pattern mining field and algorithm, with more focus on the discovery of patterns with the attention of using constraints, and presented some issues that were identified with the importance and impact of the constraints in the pattern mining process. We understood that there are strong limitations in the expression power of the constraints used until now (as pointed with the limitation of regular languages) which, consequently, affect the returned results when compared with the user expectations. This constraints limitation are due mainly to the power of the tools that are used to represent them and, so, ontologies were proposed as a better way of representing them [Antunes, 2007, 2008].

Using an ontology as a constraint representation is an attempt to enrich the mining control towards the integration of a more elaborated user knowledge in the process and, in this way, limit the return set of patterns to the expectations of the users. The main purpose of this work consisted in the development of an ontology based framework that could be used as a sequential pattern mining algorithms’ constraint input. For that goal, the OntoC4S were created. This ontology shows to be rich enough to represent a constraint sequence with all the features identified as needed (as the capacity to represent sequential and parallel events). The formation of this ontology will be based in an identified ontology, the PSL [Schlenoff et al., 1999], an ontology that was created to define a neutral representation for manufacturing processes that supports automated reasoning and has the desired ontology’s structure qualities identified and presented in this document.

The results of this work showed that while keeping the returned sequences framed in the user defined input (ontology and taxonomy) and domain knowledge, keeps the features and comparative performance of the most efficient unconstraint algorithms (as the GenPrefixSpan [Antunes and Oliveira, 2003]).

This ontology is part of the OntoC4S framework where is present the OntoC4S ontology, the algorithm created in this work, SeqD2PrefixGrowth, and designed inputs to enable the use of the framework to the intended goals. As showed in this document, the framework enables the use of constraints over the sequential mining process, with the possibility of these constraints to be over sequential or parallel events, allow the existence of gap between the elements in the constraint and the specification of complex constraints (based in more simple constraints).

Future work could be done to increase the efficiency and the effectiveness of the SeqD2PrefixGrowth algorithm and the OntoC4S framework in general. The application of the framework in a concrete study case could generate some interesting information and more action oriented when compared with unconstraint algorithms and the actuals constraint algorithms. To allow a more flexible constraint definition, the framework can be changed to deal with a tree of constraints instead of a sequence of constraints, define other kind of constraints (cardinality constraints), and create a visual tool to define the constraints and the taxonomy to ease the process of defining constraints and relations of items.
Bibliography


Appendix

Appendix A

Almaden Introduction and Experimental Setup

The evaluation experiments presented in this document were conducted over a synthetic dataset generated by the QUEST generator.

QUEST is a synthetic data generator developed at IBM Almaden\(^1\) by the Intelligent Information Systems research center. Has been adopted in relevant studies in the field of \textit{sequential pattern mining} (as can be seen in the works [Agrawal et al., 1996], [Srikant and Agrawal, 1996], [Zaki, 1998], [Han et al., 2001] and [Ayres et al., 2002]) therefore be considered a respected standard on the instantiation of synthetic sequential datasets for studies. Nowadays, Almaden’s research community focuses on solving problems across diverse areas as atomic scale storage, nanomedicine, medical image analytics, services science, and food safety.

This dataset generator builds datasets close to real world by trying to resemble veridic transactions, where customers (data creators) tend to make a sequence of transactions involving a set of items (sequential or parallel).

Sequence and transaction sizes are clustered around a mean and some of them may have larger sizes. A sequence may have repeated transactions (or itemsets), but a transaction cannot have repeated items (as a principle of the study field) and they can order by their value in the same itemset (in the case they are comparable). The data generator accepts a set of arguments to build the dataset which specify the dataset characteristics. Those arguments are shown and detailed on the table 7.1.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Signification</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Number of sequences (size of the dataset)</td>
</tr>
<tr>
<td>C</td>
<td>Average number of itemsets per sequence</td>
</tr>
<tr>
<td>T</td>
<td>Average number of items per itemset</td>
</tr>
<tr>
<td>S</td>
<td>Average length of maximal potentially frequent sequences</td>
</tr>
<tr>
<td>I</td>
<td>Average size of itemsets in maximal potentially frequent sequences</td>
</tr>
<tr>
<td>N(_s)</td>
<td>Number of maximal potentially frequent sequences</td>
</tr>
<tr>
<td>N(_i)</td>
<td>Number of maximal potentially frequent itemsets</td>
</tr>
<tr>
<td>N</td>
<td>Number of items</td>
</tr>
</tbody>
</table>

\textbf{Table 7.1: Almaden dataset generator arguments}

The physical device used to execute the experiments was a Intel Core i5 2,5GHz with 8GB 1600 MHz DDR3 of RAM. To make the time measurements more reliable, the device resources were allocated to the experiments. The comparisons with other experiments take in count the different specs of each environment. The operating system used was the OS X 10.9.5 (13F34) and the algorithms were implemented using the object oriented programming language Java (Java SE 7, version 1.7.0_51). The datasets were maintained in main memory (RAM) during the algorithms execution to avoid slowly hard disk accesses.

\(^1\)Project info at http://www.research.ibm.com/labs/almaden/index.shtml and in [Agrawal et al., 1996]