Visual Mining of Powersets with Large Alphabets

by

Qiang Kong

B.Eng., Zhejiang University, 2003

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

December, 2005

Abstract

We present the PowerSetViewer visualization system for the lattice-based mining of powersets. Searching for items within the powerset of a universe occurs in many large dataset knowledge discovery contexts. Using a spatial layout based on a powerset provides a unified visual framework at three different levels: data mining on the filtered dataset, browsing the entire dataset, and comparing multiple datasets sharing the same alphabet. The features of our system allow users to find appropriate parameter settings for data mining algorithms through lightweight visual experimentation showing partial results. We use dynamic constrained frequent set mining as a concrete case study to showcase the utility of the system. The key challenge for spatial layouts based on powerset structure is handling large alphabets, because the size of the powerset grows exponentially with the size of the alphabet. We present scalable algorithms for enumerating and displaying datasets containing between 1.5 and 7 million itemsets, and alphabet sizes of over 40,000.

Contents

List of Tables

List of Figures

Acknowledgements

First of all, I would like to thank my supervisors, Dr. Raymond T. Ng and Dr. Tamara Munzner, for their both academic and personal support during my Master work at the University of British Columbia. I am very grateful for their endless efforts to guide me in the data mining and information information research areas. The experience working with them will be my life-long assets.

Deep thanks to my committee member, Dr. Michiel Van De Panne, who spent time reading and reviewing this work. Special thanks to James Slack who is always helpful especially when I started this work. Many thanks to the members of the Imager lab and database lab who helped me a lot during this work, Dan Archambault, Ciáran Llachlan Leavitt, and Jason Harrison.

I also would like to thank all my friends here, especially Yuhan Cai, Shawn Feng, Mingming Lu, Mingyue Tan, Suwen Yang, Kangkang Yin, and Xiaodong Zhou. I will remember the time we spent together.

Last, but not least, I would like to thank my parents and aunt for their strong support. Without their love and understanding, this work could not happen. Thank you very much!

Chapter 1

Introduction

Data mining and information visualization share the same ultimate goal: identifying trends, patterns, and outliers in a sea of information. Data mining approaches the tasks in a statistical way while information visualization in a visual way. By combining these two areas, we can achieve the goal more efficiently and more effectively.

1.1 Motivation

A database is a collection of individual records, where a single item or record may convey limited information to the user. These records may appear to be unrelated to each other. The relationships between various items or records, however, may reveal much more useful information that is critical to a decision making process. Identifying trends, pattern, and other hidden relationships in large databases is the core task of knowledge discovery and data mining (KDD). Data mining is "the nontrivial extraction of implicit, previously unknown and potential useful information from data in a database" [10]. In general, data mining techniques have been successfully applied to commercial domains, such as customer relationship management, market basket analysis and credit card fraud detection, and to scientific and

engineering applications.

Most of the data mining tasks involves exploring how objects are related to each other within a universe of objects. Primary examples include:

- Finding association rules and frequent sets: association rules identify co-occurring events, and finding frequent sets is the first step to discover association rules [1]. For example, in a sales transaction dataset, a typical association rule would be: a certain percentage of customers who buy milk will also buy bread. A frequent set is a set of items that are frequently bought together.

- Mining sequential patterns: In many domains, such as finance, data can be ordered based on certain dimensions, for example, time or location. To identify all the subsequences that appear frequently given that ordering will help us to predict what will happen next [2].

- Building decision trees: Decision trees are used in classification-related problems [4]. A decision tree is a tree representation of a decision procedure for assigning a class label to a given example, which distinguishes one group from another. At each internal node of the tree, there is a test, and a branch corresponding to each of the possible outcomes of the test. At each leaf node, there is a class label. A class label is assigned to an entity by traversing a path from the root to a leaf.

- Clustering: clustering is a division of data into groups of similar objects [18]. Information retrieval, web analysis, customer relationship

management, marketing, medical diagnostics, computational biology, and many other data mining applications require data to be clustered, where items in a particular cluster share certain similar properties.

After a close look at these aforementioned tasks, it is obvious that they all require searching for "interesting" groupings of objects within a search space, which consists of part or all possible groupings of objects. It is almost the same as searching sets within a powerset space, which is a collection of all possible sets. At the same time, analysts may be interested in not only the individual "interesting" set, but also the *inter-set* information of the individual set in the context of other "interesting" set or in the context of the entire powerset space. The traditional text-based data mining applications perform quite well when the cardinality of the final result is small. However, when the number of items returned by the data mining engine exceeds a certain threshold, typically 20, the traditional text-based application will probably fail to provide analysts with those contextual information which is critical to most of the data mining tasks, since the results provided by the text-based applications are organized in a way that is hard for the analysts to gain any useful inter-set information.

Information visualization is the use of interactive representations, typically visual, of abstract data to facilitate humans' interaction for exploration and understanding. It is a complex research area that involves information design, computer graphics, human-computer interaction and cognitive science [32]. A primary goal of information visualization is to utilize the high-bandwidth channel of the human visual system to facilitate the process of

identifying trends, clusters, gaps, and outliers. The representations offered by a well-designed information visualization application have the following advantages over the traditional text-based representations.

First, more information is available to the user given the same screen space. The information provided by the traditional text-base applications is limited by the size of the text [14]. In order to show more information on the display, we have to decrease the size of the text. However, the text becomes illegible when the size of the text passes a certain threshold. In contrast, information visualization applications usually use marks to represent records in the original dataset, where different visual or retinal properties are used to encode fields of the records. Therefore, a lengthy record in the original dataset is rendered as a mark on the display. If users are interested in the original information encoded by a particular mark, users can simply move the mouse over the mark and the detailed information will be shown in either a pop up dialogue or a dedicated information panel.

Secondly, an overview of the dataset is available to the user. By allowing more and more data in the original dataset to be visualized at the same time, users can get an overview of the dataset, which is vital to many analysis tasks. Human's high bandwidth visual system can quickly identify trends, pattern, or outliers in the dataset. These hidden gems are unlikely to be easily acquired by looking at thousands of lines of characters.

Last, users are able to fully interact with the data. Users may select an area of interests and zoom in to investigate the local or detailed information in that area. Users can also filter out some "uninteresting" data points or highlight certain "hot" data points, in order to focus their attention to the

interesting items.

Furthermore, some of the data mining tasks require a number of parameters to be set [20]. It is very likely that the final result, which takes hours if not days to get, is not what the user needed because of the inaccurate or inappropriate parameter settings. Offering users the ability to not only explore the search space but also see and steer the computation would help significantly.

1.2 Thesis statement

The ideal system is able to offer a meaningful visualization based on the huge search space without losing any important information. The ideal system also provides users with the ability to explore in the search space, which require at least 10 frames per second to ensure good user interaction experience. Moreover, it is critical for the system to maintain a "fixed" layout of the groupings, since keeping the relative positions of the groupings enables users to easily compare different datasets that share the same alphabet.

1.3 Contributions

We present the contributions of our system, which are ordered by their significance, in this section. They will be discussed in detail in Chapter 4.

- **Powerset enumeration**: We propose an enumeration of powerset for PSV that is ordered first by cardinality, then by lexicographical ordering. Creating a spatial layout that maps a related family of datasets

into the same absolute space is a powerful visualization technique, which enable the user to compare different datasets that share the same alphabet. We also develop an algorithm for calculating the index of a powerset in the powerset enumeration that is linear to the cardinality of the set and independent of the size of the alphabet.

- **Scaling to large alphabet**: We propose new data structures and develop an algorithm which enables PSV to accommodate large alphabet of more than 40,000 items, which has 2^{40000} potential sets in the entire powerset space.

- **Rendering and picking in PSV**: Real-time interactivity requires frame rates of at least 10-20 frames per second, which is a challenge when rendering millions of objects distributed within the sparse powerset space. PSV can handle more than six million records using a new rendering and traversal algorithm that is optimized for sparse data. As will be explained in Section 6.2, the rendering time is near-constant after a threshold data set size has been reached, and this constant time is very small: 60 milliseconds per scene, allowing over 15 frames per second even in the worst case.

1.4 Outline of the thesis

This chapter presented the motivation, thesis statement, an overview of our approach, and the contributions. The remaining chapters of the thesis are organized as follows. Chapter 2 introduces related work. Chapter 3 gives

detailed overview of our approach and the features that are supported by PSV. Chapter 4 explains our approach and implementation issues. Chapter 5 presents a concrete example to showcase the utility of PSV. Chapter 6 details the performance of PSV. Finally, Chapter 7 discusses future work as well as a conclusion that highlights the contributions of our research. This document was based in part on a technical report coauthored by Tamara Munzner, Raymond T. Ng, Jordan Lee, Janek Klawe, Dragana Radulovic, and Carson K. Leung [22].

Chapter 2

Related work

In this chapter, we first give a brief introduction to Information Visualization in Section 2.1. Then, we focus the discussion of related work on three categories: database visualization systems in Section 2.2, mining results visualization systems in Section 2.3, and steerable visualization systems in Section 2.4. Finally, we introduce two applications that are also based on the Accordion Drawing infrastructure in Section 2.5.

2.1 Information visualization

Information visualization bridges two most powerful information processing systems: the human mind and the computer. It transforms data into a visual form and takes advantage of humans' remarkable abilities in identifying trends, patterns, and outliers. Information visualization systems enable users to acquire the information they need through observation, search, navigation and exploration of the original data. One of the challenges in Information Visualization is how to design a meaningful spatial mapping of the original data that is not inherently spatial. A well designed spatial layout will reveal some useful hidden patterns that is probably unavailable in the plain representation of the original data. In addition to the spatial

position, information visualization applications also take advantage of other visual channels to encode information, such as shape, size, orientation, color, and texture [7]. However, only part of these visual encodings are helpful for a given task, which imposes another design challenge. A typical information visualization application will employ some or all of the following techniques to help users identify the information they need [25]:

- Overview: Providing an overview of the entire dataset allows users to understand its global structure.
- Zoom: Users are able to zoom in to have a detailed view of the "interesting" items, which reveals some local information.
- Filter: Users can filter out "uninteresting" items.
- Detail-on-demand: Provides detailed information on selected items at the user's request. This technique avoids potential clutter in the visualization without losing important information.
- Related: Users can view relationships among items.

Information visualization and data mining share the same ultimate goal: identifying hidden patterns in the datasets. Therefore, visualizing databases is one of the active areas in the field.

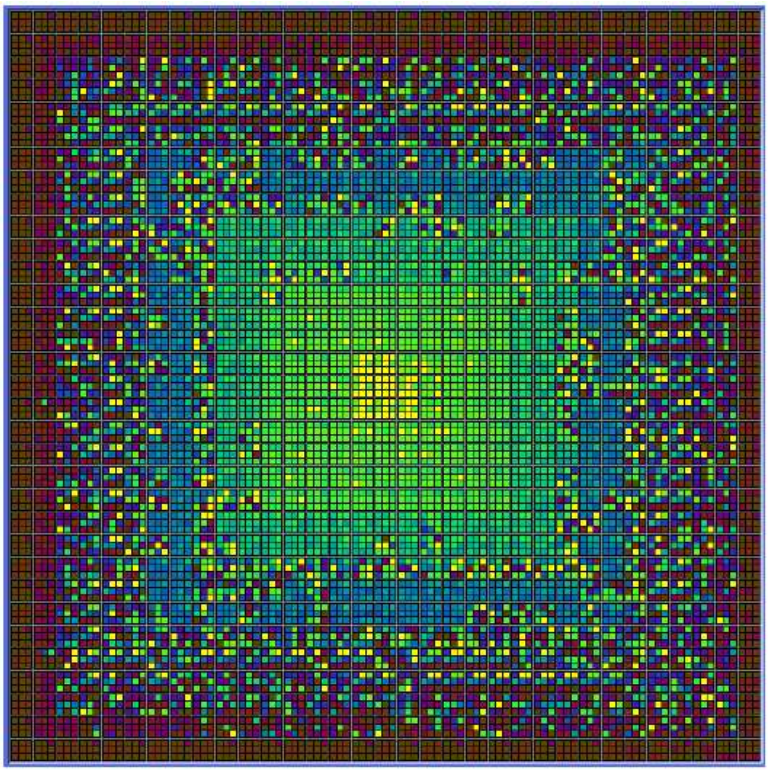

Figure 2.1: The VisDB system [15], showing a dataset with eight dimensions and 1000 items. The most relevant record with respect to a query is color coded in yellow and is centered in the box. The least relevant records are positioned far away from the center of the box and are colored in black.

2.2 Database visualization systems

VisDB

VisDB [15], shown in Figure 2.1, is a visualization tool that allows users to explore large multidimensional databases in terms of a query result. The VisDB system uses pixel-oriented techniques, where each record in the database is represented by a single pixel or a group of pixels. VisDB takes advantage of spatial encoding and color to indicate the relevance of the data

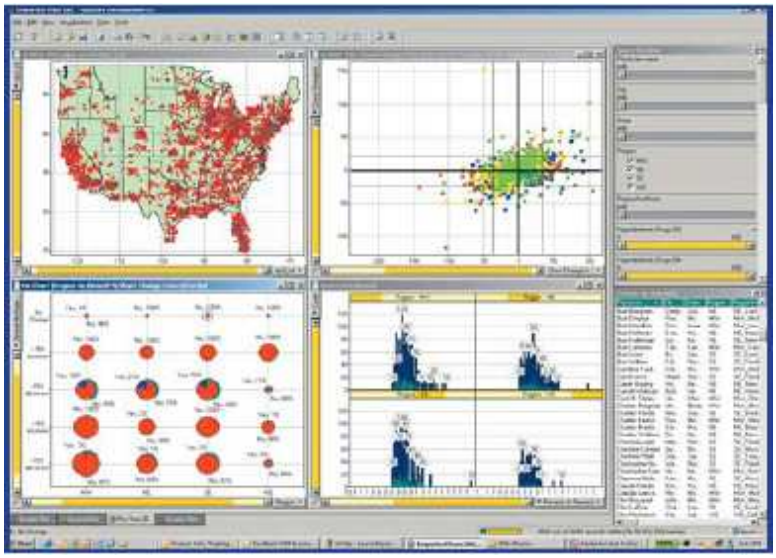

Figure 2.2: The Spotfire system [3], showing different types of visualizations on a sales dataset, which includes a geographical chart and two-dimensional plots.

items with respect to the query in order to give the user an overview of the query result. VisDB also supports dynamic query refinement, where users can use sliders to change the parameters of the query based on the immediate visual feedback. However, VisDB only provides information about the relationship between individual items and the query. It does not offer any information about the relationships among items in the dataset, which is critical to most of the data mining tasks. Moreover, the size of the dataset that can be accommodated in VisDB is limited by the number of pixels on screen. The current version of VisDB only allows an interactive database exploration for datasets containing up to 50,000 data items [15].

2.2.1 Spotfire

Spotfire [3], as shown in Figure 2.2, is a database exploration system that employs several information visualization techniques, including dynamic queries, brushing and linking, and other interactive graphic techniques. This system focuses on visualizing the original dataset and uses graphical interface to help users to identify trends, patterns, and outliers. However, Spotfire does not provide visualization of any data mining results. The Spotfire system also suffers from a lack of scalability and cannot handle databases that information are summarized using a hierarchical, where users are able to acquire information with different granularity.

2.2.2 Independence Diagrams

Independence Diagrams [6], as shown in Figure 2.3, help users identify correlations between any two attributes in a particular database. The Independence Diagram systems divide each attribute into ranges. For each pair of attributes, the combination of these ranges forms a two dimensional grid. Brightness is used to encode the number of data items within that cell. Since Independence Diagrams can only show the relationship between two attributes at once, users may have to try several times before they can find the "interesting" pairs. Moreover, users are not able to find any correlations or pattern among these independence diagrams. In contrast, PSV is designed to put all these "combinations" of dimensions in a single window, where users can identify individual interesting combinations (any two dimensions that are highly correlated or highly independent) as well as the

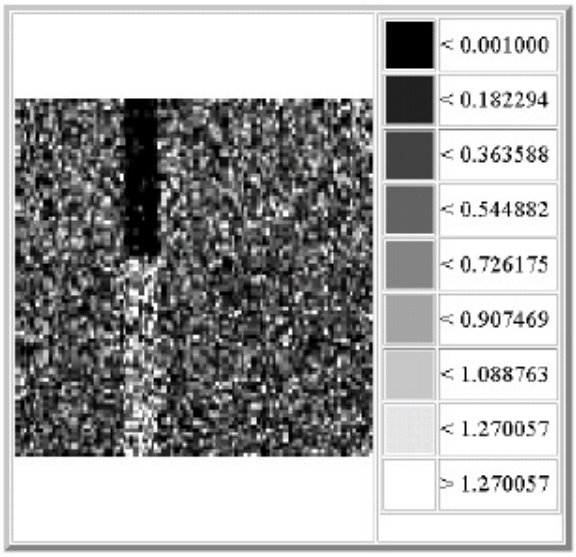

Figure 2.3: An Independence Diagram with legend [6], showing a synthetic dataset. The brightness is used to encode the degree of dependence. A darker region means that there are less items in it. The two attributes are independent, except for a range of X-attribute values, where the region is brighter.

correlations or patterns among them.

2.2.3 Polaris

Polaris [30, 29], as shown in Figure 2.4, is an interactive visual exploration tool that facilitates exploratory analysis of multidimensional datasets with rich hierarchical information. Polaris builds a visualization of a grid of small multiples [31] given a user query. Polaris also provides a visual interface to help the user formulate complex queries against a multi-dimensional data cube. By taking advantage of the hierarchical structure of the data cube, Polaris allows the analyst to drill down or roll up data to get a complete overview of the entire dataset before focusing on detailed portions of the

Figure 2.4: The Polaris system [30, 29]. Each chart displays the profit and sales over time for a hypothetical coffee chain, organized by state.

dataset. However, the utility of Polaris is still limited to processing basic queries against the data cubes as apposed to offering any supports to explore powerset space.

In summary, these four systems provide features to arrange and display data in various forms. However, these systems are not designed to display data mining results nor could they be easily changed to accommodate powersets enumeration. The PSV system, in addition to allowing the raw data to be visualized and explored, provides a *unified* visual framework to the user to examine the data mining results as well.

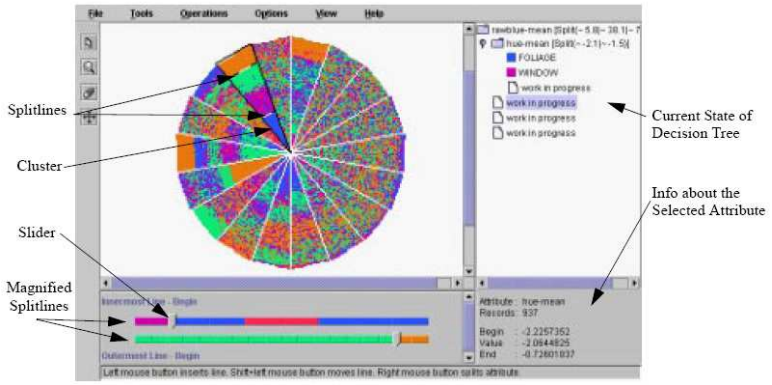

Figure 2.5: The PBC system [4]. Each sector shows a dimension of the original dataset, where individual data points are ordered by the values of the attribute and color coded based on the class label.

2.3 Mining results visualization system

2.3.1 Decision trees

The PBC framework proposed by Ankerst et al. [4], as shown in Figure 2.5, is not a general visualization system. It focuses on involving the user in the process of building decision trees, which is based on a pixel-based multidimensional visualization technique and interaction capabilities. The system overcomes the limitation of most decision trees which are fixed to binary splits for numerical attributes. However, since the visualization is only used as an indicator which shows how a split value affects the classification result, the user is not able to get any information about the individual records in the original database or the relationships among them.

2.3.2 Association rules

The rule visualization system developed by Han and Cercone [11] focuses on the discretization of numeric attributes and discovery of the numerical association rules for large data set. The system uses a two-dimensional plot to show the original datasets and parallel coordinates to show the mined association rules. However, the system only takes two attributes into consideration throughout the mining process. Once the data point is laid out on the display, users are not able to get the detailed information of the individual records. In contrast, PSV allows the user to operate on all attributes of the records. Hofmann et al. [13] use a variant of mosaic plots, called double decker plots, to visualize association rules. Their focus is to help users understand association rules. PSV instead operates at the level of frequent sets. Furthermore, unlike the two previous frameworks, PSV supports the steering of the mining process midstream. Again, the use of a spatial layout based on a powerset is unique.

2.3.3 Clustering

The visualization method developed by Koren and Harel [18] is designed for clustering analysis and validation. They integrate a dendrogram, which contains hierarchical information, and a low dimensional embedding. The leaf nodes of the dendrogram are well ordered so that similar nodes are adjacent to each other. The data points in the low dimensional embedding are drawn exactly below the corresponding leaf node in the dendrogram so that the user is able to mentally connect the two parts, as shown in

Figure 2.6. This system is a specific visualization tool for exploring clusters and provides no information at the level of individual records. Moreover, the size of the dendrogram is limited by the number of screen pixels, which limits the utility of this system.

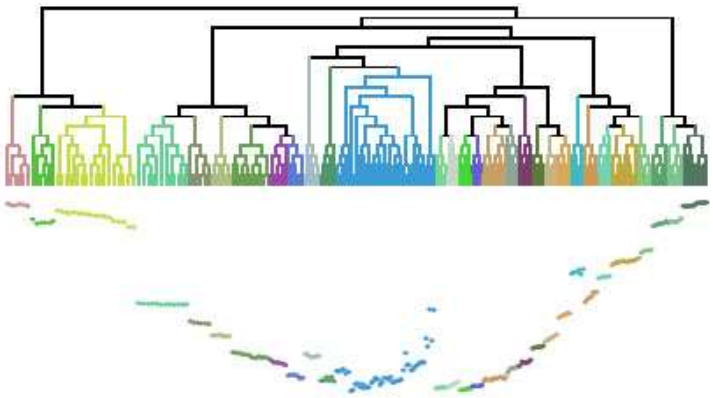

Figure 2.6: The two-way visualization system for clustered data [18].

The visual metaphors of all the mining result visualization systems are quite different from the PSV system, which uses a spatial layout based on the powerset of an alphabet. They use dots lines, or rectangles to represent the data points in the original datasets. These on-screen elements may overlap with each other when the number of data points exceeds a certain threshold. However, all these systems fail to offer elegant solutions to occlusion when large datasets are laid out using these systems, which limit their abilities in handling large datasets. Moreover, they do not provide users with the ability to explore the original datasets.

2.4 Steerable visualization systems

2.4.1 SCIRun

SCIRun [24] is a scientific programming environment that allows user to refine parameter settings at any phase of the computation based on the visual representation of the partial result. It offers users the ability to find the cause-effect relationships within the simulation. However, SCIRun is designed to provide volumetric representation of the data, such as medical imaging and scientific simulation, and uses a very different visual representation than that of PSV.

2.4.2 MDSteer

MDSteer [33] is an interactive visualization tool designed to apply Multidimensional Scaling (MDS) to very large datasets. The user can steer the computation of the algorithm to the areas of interests by creating a rectangular box on the screen. Figure 2.7 shows a partial layout after 20 seconds of interactive steering, which successfully reveals the large-scale structure.

2.4.3 Discussion

These two application are not designed to steer data mining engines. In contrast, PSV can be connected to a data mining engine and allows the user to explore both the original dataset and the mining result.

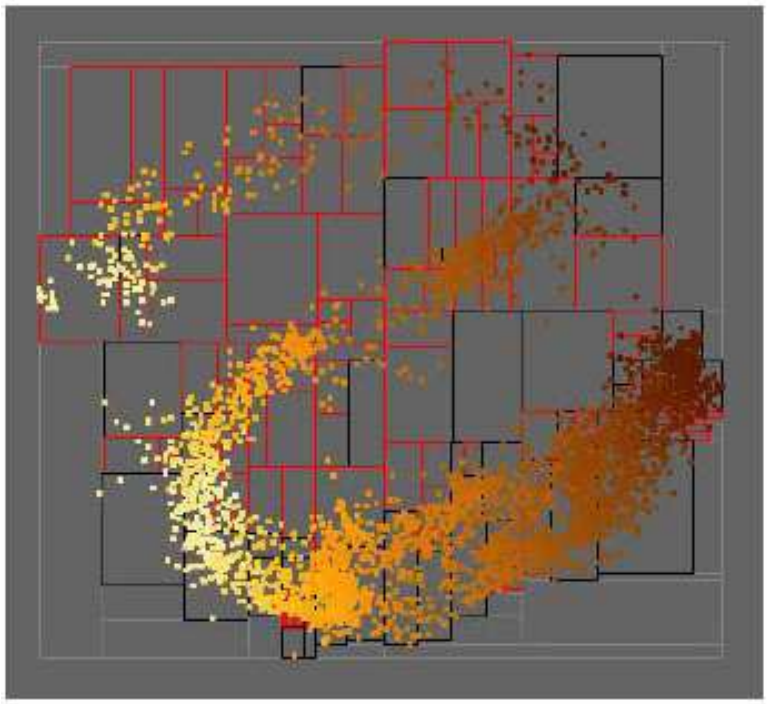

Figure 2.7: The MDSteer system [33], showing a partial layout of a $50,000$ point S-shaped benchmark dataset. Users can steer the computation by creating boxes.

2.5 Accordion Drawing

Accordion drawing is an information visualization technique that features rubber sheet navigation and guaranteed visibility. **Rubber-sheet navigation** allows the user to select any rectangular area to stretch out, showing more detail there, and that action automatically squishes the rest of the scene. All stretching and squishing happens with smoothly animated transitions, so that the user can visually track the motions easily. Parts of the scene can become highly compressed, showing very high-level aggregate views in those regions. However, no part of the scene will ever slide out of

the field of view, following the metaphor that the borders of the malleable sheet are nailed down. A second critical property of accordion drawing is the **guaranteed visibility** of visual landmarks in the scene, even if those features might be much smaller than a single pixel. Without this guarantee, a user browsing a large dataset cannot know if an area of the screen is correctly blank because it is truly empty, or if it is misleadingly blank because marks in that region happen to be smaller than the available screen resolution.

Accordion drawing was originally proposed for browsing phylogenetic trees [21, 5], as shown in Figure 2.8, and was then adapted for the task of visually comparing multiple aligned gene sequences [28], as shown in Figure 2.9. The powerset-based spatial layout used by PSV was another extension to the generic accordion drawing framework[27]. This previous work focuses on dense and static data, while PSV addresses sparse and dynamic data.

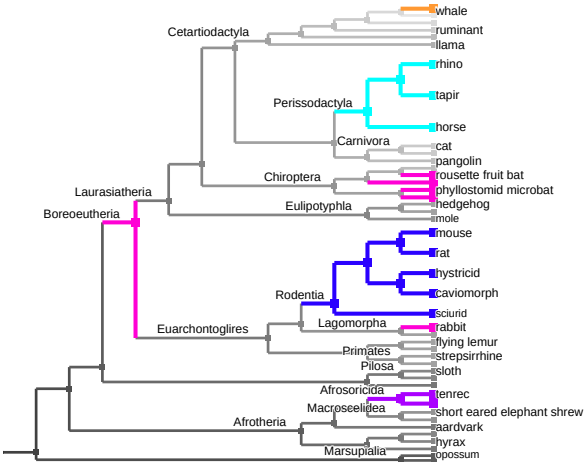

Figure 2.8: The TreeJuxtaposer system [21], showing a phylogenetic tree with hierarchical structure. Some subtrees are highlighted due to the user's interaction.

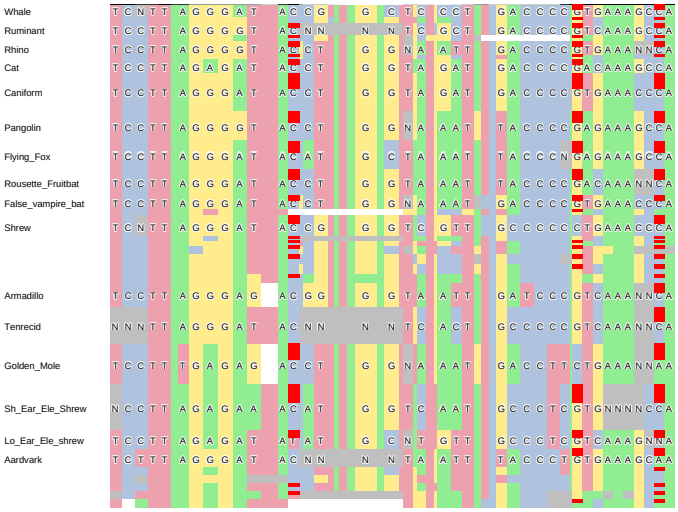

Figure 2.9: The SequenceJuxtaposer system 2.9, showing a set of related gene sequences. These sequence are vertically aligned for easy comparison.

Chapter 3

PSV overview

In this chapter, we give a detailed overview of our approach. Sections 3.1.1 through 3.1.4 explain the features that are supported by our system. Section 3.2 explains the client-server architecture of PSV. Finally, Section 3.3 introduces PSVproto, an initial prototype, and its limitations.

3.1 Features

PSV provides not only a visual framework for examine individual powersets in the context of the entire powerset space, but also an interface where users are able to find appropriate parameter settings for data mining algorithms through lightweight visual experimentation showing partial results. This lightweight visual experimentation also shows how various parameter settings would create a filter for the mined data with respect to the entire dataset. Moreover, when the mining algorithms are steerable, dynamic display of intermediate or partial results helps the user decide how to change the parameters settings of computation in midstream. In our unified framework, we can support setting the filter parameters to let all items pass through, so that the entire input dataset is shown to the user. Another benefit of using the powerset for spatial layout is that users can even meaningfully compare

images that represent two different datasets that share the same alphabet, for example by comparing the distribution of purchases between two chain stores in different geographic regions. Our system provides the following features to assist users with the frequent set mining.

3.1.1 Visual metaphor

PSV uses the visual metaphor of **accordion drawing** [27], which is introduced in Section 2.5. Although the absolute on-screen location of an itemset changes, the relative ordering of the itemset with respect to its neighbors is always preserved in both the horizontal and vertical directions. Accordion drawing also allows interactive exploration of datasets that contain many more itemsets than the fixed number of pixels available in a finite display.

3.1.2 Layout

PSV introduces a novel layout that maps a related family of datasets, those sharing the same alphabet of available items, into the same absolute space of all possibilities. That space is created by enumerating the entire powerset of a finite alphabet as a very long one-dimensional list, where every possible set has an index in that list. That linear list is wrapped scanline-style to create a two-dimensional rectangular grid of fixed width, with a small number of columns and a very large number of rows. We draw a small box representing a set if it is passed to the the visualizer by the miner, located at the position in the grid corresponding to its index in this wrapped enumeration list. Without guaranteed visibility, these boxes would be much smaller than pixels in the display for alphabets of any significant size because

of the exponential nature of the powerset. This guarantee is one fundamental reason why PSV can handle large alphabets.

In areas where there is not enough room to draw one box for each set, multiple sets are represented by a single aggregate box. The color of this box is a visual encoding of the number of sets that it represents using saturation: objects representing few sets are pale, and those representing many are dark and fully saturated. Color is also used in the background to distinguish between areas where sets of different cardinality are drawn: those background regions alternate between four unobtrusive, unsaturated colors. The minimum size of boxes is controllable, from a minimum of one pixel to a maximum of large blocks that are legible even on high-resolution displays.

In this layout, seeing visual patterns in the same relative spatial region in the visualization of two different datasets means they have similarities in their distribution in this absolute powerset space. Side by side visual comparison of two different datasets sharing the same alphabet is thus a fruitful endeavor.

3.1.3 Interaction

Interactions that can be accomplished quickly and easily allow more fluid exploration than those that require significant effort and time to carry out. The PSV design philosophy is that simple operations should only require minimal interaction overhead. The rubber-sheet navigation, where the user sweeps out a box anywhere in the display, and then drags the corner of the box to stretch or shrink it, is just one example. Mouseover highlighting occurs whenever the cursor moves, so that the box currently under the

cursor is highlighted and the names of the items in that highlighted itemset are shown in a status line below the display. Mouseover highlighting is a very fast operation that can be carried out many times each second because it does not require a redraw of the entire scene. Highlighting the superset of an itemset can be done through the shortcut of a single click on the itemset's box.

The layout and rubber-sheet navigation provide a spatial substrate on which users can explore by coloring sets according to constraints, as described above. We do not support changes in the relative spatial position of itemsets, because it would then be impossible to usefully compare visual patterns at different times during the interaction. The underlying mechanism for coloring is to assign sets to a group, which has an assignable color. A group is a collection of sets that satisfy certain constraints. Users can create an arbitrary number of colored groups, so they can be a mechanism for tracking the history of both visualizer and miner constraints, by saving each interesting constraint choice as a separate group. The priority of groups is controllable by the user; when a particular set belongs to multiple enabled groups, the highest priority group color is shown.

3.1.4 Monitoring

As will be discussed in Chapter 5, the visualizer shows several important status variables which helps the user to adjust the parameter settings:

- *total*: the total number of itemsets in the raw dataset;
- *processed*: the number of itemsets processed so far by the miner;

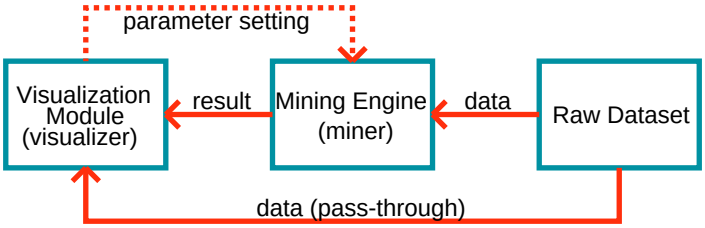

Figure 3.1: PSV has a client-server architecture, with a visualizer that can show either the filtered results from the miner or the raw data directly.

- *shown*: the number of itemsets passed on to the visualizer to display;
- *rows*: the number of visualizer rows needed so far;
- *maxrow*: the biggest visualizer row needed so far;

Comparing these numbers helps users make choices: for instance, *total* vs. *processed* is the progress of the miner, and *processed* vs. *shown* shows the amount of filtering done by the miner. The *rows* and *maxrow* pair shows the average distribution density of itemsets. The higher the *rows* to *maxrow* ratio, the denser the distribution, since the higher ratio means there are less empty rows. Comparing *shown* with *processed* gives the user feedback on whether the miner constraints should be changed to make the filter tighter or looser.

3.2 Client-server architecture

PSV has a client-server architecture, as shown in Figure 3.1. The server is a steerable data mining engine, the miner, that is connected through sockets with a client visualization module, the visualizer, that handles graphical

display. The visualizer client includes interface components for controlling both itself and the miner server. The client and server communicate using a simple text protocol: the client sends control messages to the server and restart. The miner sends partial results to the visualizer as they are completed, allowing the user to monitor the progress.

3.3 PSVproto

PSVproto, an initial prototype of PSV, had been implemented when this thesis began. The server-side mining engine was first developed by Carson Leung and was modified by Dragana Radulovic afterwards so that the engine was able to communicate with the visualizer via plain text protocol. The client-side of PSVproto was developed by Jordan Lee. Figure 3.4 shows the original PSVproto interface. On top is the rendering window, which supports Accordion Drawing navigation. The lower control panel allows users to communicate with the data mining engine. In addition to the main control panel and rendering window, PSVproto also provided the users with the following panels:

- Grouping panel: as shown in Figure 3.2, users can select a group of items that satisfy the supplied constraints, and highlight them for further exploration.
- Visualization panel: as shown in Figure 3.3, users can turn on the background color so that sets that are of different cardinalities could be easily identified.

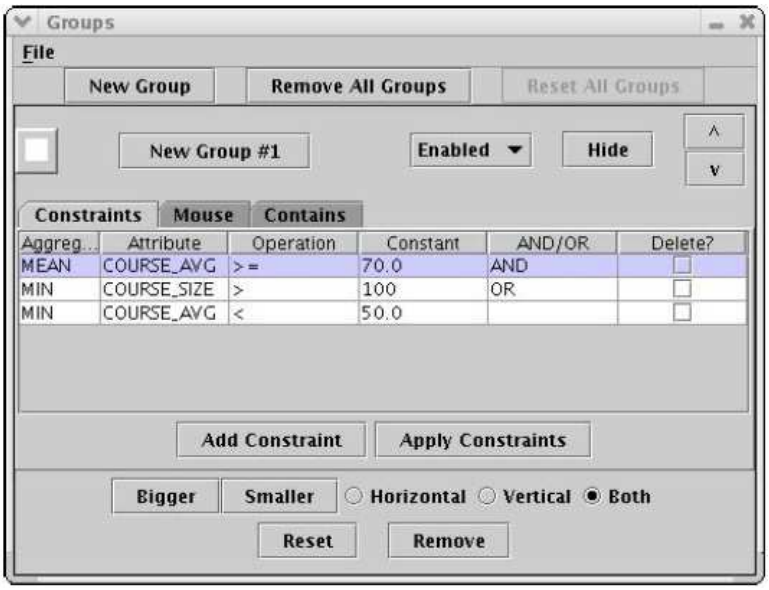

Figure 3.2: The grouping panel of PSVproto. [19]

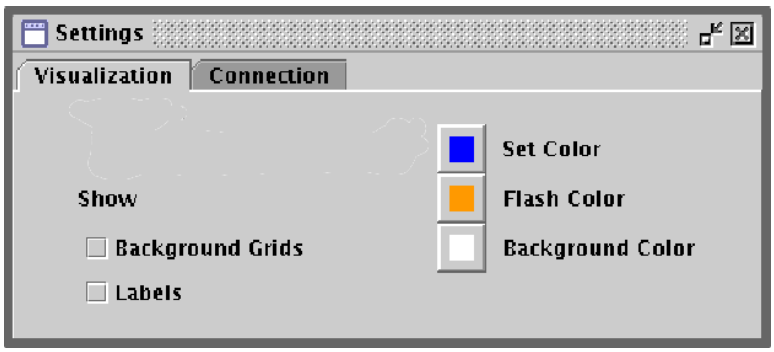

Figure 3.3: The visualization panel of PSVproto. [19]

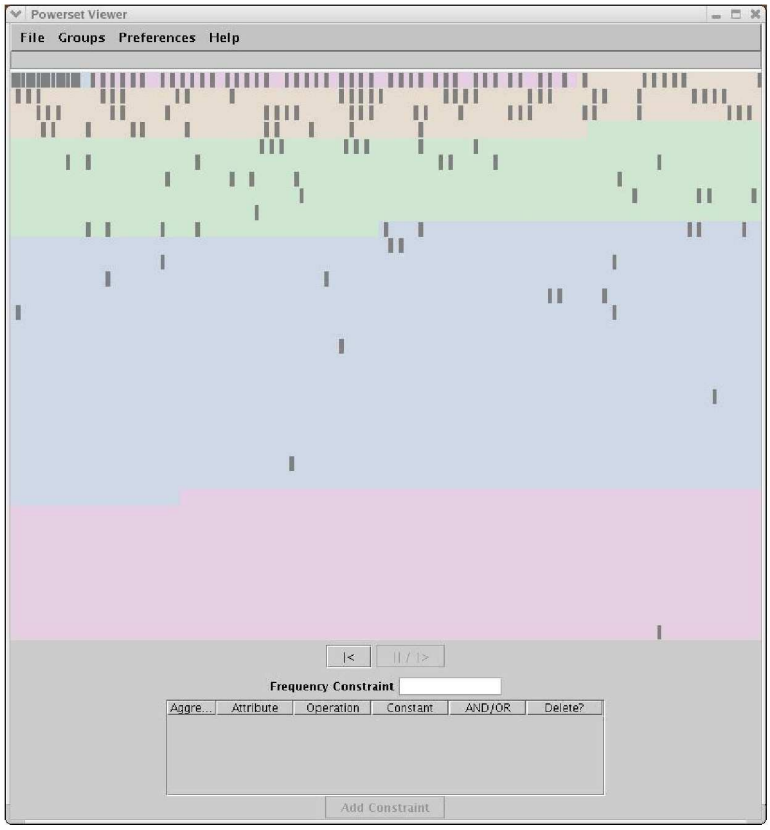

Figure 3.4: The interface of the PSV prototype. [19]

Those features supported by PSVproto offers users with powerful tool to explore the datasets. However, PSVproto still suffers from the following limitations:

- **Enumeration**: The prototype uses a brute force algorithm to calculate the index of a set in the entire powerset enumeration. Given a set s, the algorithm traverses the powerset space from the very beginning of the enumeration until it encounters s. On average, this approach took 40 milliseconds to compute the index of a set. Since the operation

was performed every time when a set came into PSVproto, it is not ideal for processing very large datasets.

- **Scalability**: The PSV prototype cannot display a dataset with an alphabet size larger than 30 [19]. Since the PSV prototype uses the plain `Integer` type variable as the index of a set in the powerset enumeration, the size of the alphabet is in turn limited by the maximum value of the `Integer`, which is 2^{31}.

- **Rendering**: Since PSV prototype cannot handle a dataset with an alphabet size larger than 30, the rendering was not efficient. The PSV prototype employed a quad-tree data structure. A quad-tree is very similar to a binary tree, but any node in a quad-tree can contain up to four children. The quad-tree maintains a hierarchical structure of the rendering window, where each leaf node corresponds to a set returned by the server. Since we subdivide the screen as we are traversing from the root to a leaf node, a large portion of the screen space will not be used since the distribution of the sets is extremely sparse. Therefore, quad-tree is not an ideal data structure for rendering any mid-size or large-size datasets. Moreover, it takes PSVproto half a second to render a scene that contains only a few thousand items.

In summary, due to the aforementioned limitations of PSVproto, it is obvious that PSVproto is not suitable to process most real world datasets, whose alphabet sizes far exceed its limit. In the next chapter, we will explain how PSV are designed to eliminate those limitations.

Chapter 4

Approach

As discussed in Chapter 3, our approach can be divided into three parts. The first part is to map each set to a box on the display, where the relative positions of the sets should be maintained. The second part is to build and maintain related data structures to support fast rendering and navigation. The last part is to render the sets on screen. In this chapter, we discuss in detail these three parts in section 4.1 to section 4.3 respectively. In section 4.4, we explain how PSV is designed to handle datasets with large alphabets.

4.1 Mapping

Mapping is the process of finding the position of a set in the enumeration of the powersets and using an on-screen box to represent it, so that the relative position of the set with respect to other sets is maintained.

4.1.1 Challenges

A meaningful ordering of the powerset is critical to the data mining tasks, because useful hidden patterns will be made available if a meaningful ordering of the powerset is chosen. Moreover, each set that comes into PSV

will go through the mapping process. Therefore, an appropriate yet efficient algorithm is required to map each set to a position on the screen.

4.1.2 Our Approach

There are many ways to enumerate powersets, but when visually represented, most do not yield a helpful mental model. The enumeration we use is ordered first by cardinality, then by lexicographic ordering. All singleton sets are shown before the two-sets, two-sets before the three-sets, and so on. The steerable data mining engine that motivated this application uses an underlying lattice structure, so first sorting by cardinality is a good match. Within a given cardinality, we choose a lexicographic ordering for alphabet items, again to match the powerset traversal order of many lattice-based mining algorithms. For example, an alphabet of {a,b,...,z} yields the enumeration {a}, {b}, ..., {z}, {ab}, {ac}, ..., {yz}, {abc}, ... We assume the underlying alphabet has a canonical lexicographic ordering; for example, $a = 1, b = 2, \ldots, z = 26$. Another design guideline was to choose a single spatial layout and allow users to find patterns by changing the colors of data elements. If we used spatial proximity to show membership of some chosen element, for example by grouping sets containing the element b, the layout would change drastically and visual patterns from different times could not be meaningfully compared. The rubber-sheet navigation can change the absolute position of boxes in space, but preserves relative ordering of marks in all directions. All computations involving sets assume that their internal item ordering is also lexicographically sorted.

The mapping from a set to a box that is drawn in a display window has

two main steps:

- convert from an m-set $\{s_1, \ldots, s_m\}$ to its index e in the enumeration of the powerset
- convert from the enumeration index e to a $(row, column)$ position in the grid of boxes

We present an efficient O(m) algorithm for the first stage of computing an enumeration index e given an arbitrary set. The second stage is straightforward: *row* is e divided by the width of the grid, and *column* is e modulo the width. We start with an example of computing the enumeration index $e = 1206$ of the 3-set {`d`,`h`,`k`} given an alphabet of size 26.

Given a particular m-set, the computation of the index in the enumeration of the powerset is done in two steps. The first step is to compute the total number of k-sets, for all $k < m$. These are all the sets with a strictly smaller cardinality. For the {`d`,`h`,`k`} example, the first step is to compute the total number of 1-sets and 2-sets, which is given by $\binom{26}{1} + \binom{26}{2} = 26 + 325 = 315$. The general formula, where A is the size of the alphabet, is

$$\sum_{i=1}^{m-1} \binom{A}{i}.$$

The second step is to compute the the number of sets between the first m-set in the enumeration and the particular m-set of interest. For the {`d`,`h`,`k`} example, the second step computes three terms:

- the number of 3-sets beginning with the 1-prefixes {`a`}, {`b`}, or {`c`}: $\binom{26-1}{2} + \binom{26-2}{2} + \binom{26-3}{2} = 300 + 276 + 253 = 829$. Picking a as a 1-prefix leaves 2 other choices that yield a 3-set containing a; there are

25 other items left in the alphabet from which to choose 2. Similarly, when b is then picked as the 1-prefix, there are only 24 choices left; since the m-set is internally ordered lexicographically, neither a nor b are available any more as choices.

- the number of 3-sets beginning with 2-prefixes {`d,e`}, {`d,f`}, or {`d,g`}, which is given by $\binom{26-5}{1} + \binom{26-6}{1} + \binom{26-7}{1} = 21 + 20 + 19 = 60$; and
- the number of 3-sets between the 3-prefixes {`d,h,i`} and {`d,h,j`}, which is $\binom{26-9}{0} + \binom{26-10}{0} = 1 + 1 = 2$.

This example suggests a formula of

$$\sum_{i=1}^{m} \sum_{j=p_{i-1}+1}^{p_i-1} \binom{A-j}{m-i}$$

where p_i is the lexicographic index of the ith element of the m-set and p_0 is 0. In the worst case, the number of terms required to compute this sum is linear in the size of the alphabet. However, we can collapse the inner sum to be just two terms by noticing that

$$\sum_{i=0}^{j} \binom{n-i}{k} = \binom{n+1}{k+1} - \binom{n-j}{k+1}.$$

We derive this lemma using the identity $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$. The general formula is thus given by

$$\sum_{i=1}^{m} \left[\binom{A-p_{i-1}}{m-i+1} - \binom{A-p_i+1}{m-i+1} \right] \tag{4.1}$$

Combining these two steps, we can compute the enumeration index as

$$\sum_{i=1}^{m-1} \binom{A}{i} + \sum_{i=1}^{m} \left[\binom{A - p_{i-1}}{m - i + 1} - \binom{A - p_i + 1}{m - i + 1} \right] \tag{4.2}$$

The complexity of computing the index of a set can be reduced to $O(m)$, where m is the cardinality of the set, by using a lookup table instead of explicitly calculating $\binom{n}{k}$. We compute such a table of size $n \times k$ using dynamic programming in a preprocessing step. As we will discuss in Section 4.4.2, the maximum set size k needed for these computations is often much less than the alphabet size n, but we do not want to hardwire any specific limit on maximum set size. Our time-space tradeoff is to use the lookup table for the common case of small k, 25 in our current implementation, and explicitly compute the binomial coefficient for the rare case of a large k.

4.1.3 Knuth's Algorithm

As discussed in Section 4.1.1, an efficient enumeration algorithm which yields a meaningful visualization is essential to PSV. In this section we present an alternative enumeration algorithm proposed by Donald E. Knuth to explore the possibilities of utilizing different enumeration methods in PSV [16]. We will compare Knuth's approach with ours in terms of both complexity and usefulness.

Knuth's algorithm uses the following formula to calculate the index of an arbitrary m-set in the powerset enumeration:

$$\sum_{i=1}^{m-1} \binom{A}{i} + \sum_{i=1}^{m} \binom{p_i - 1}{i} \tag{4.3}$$

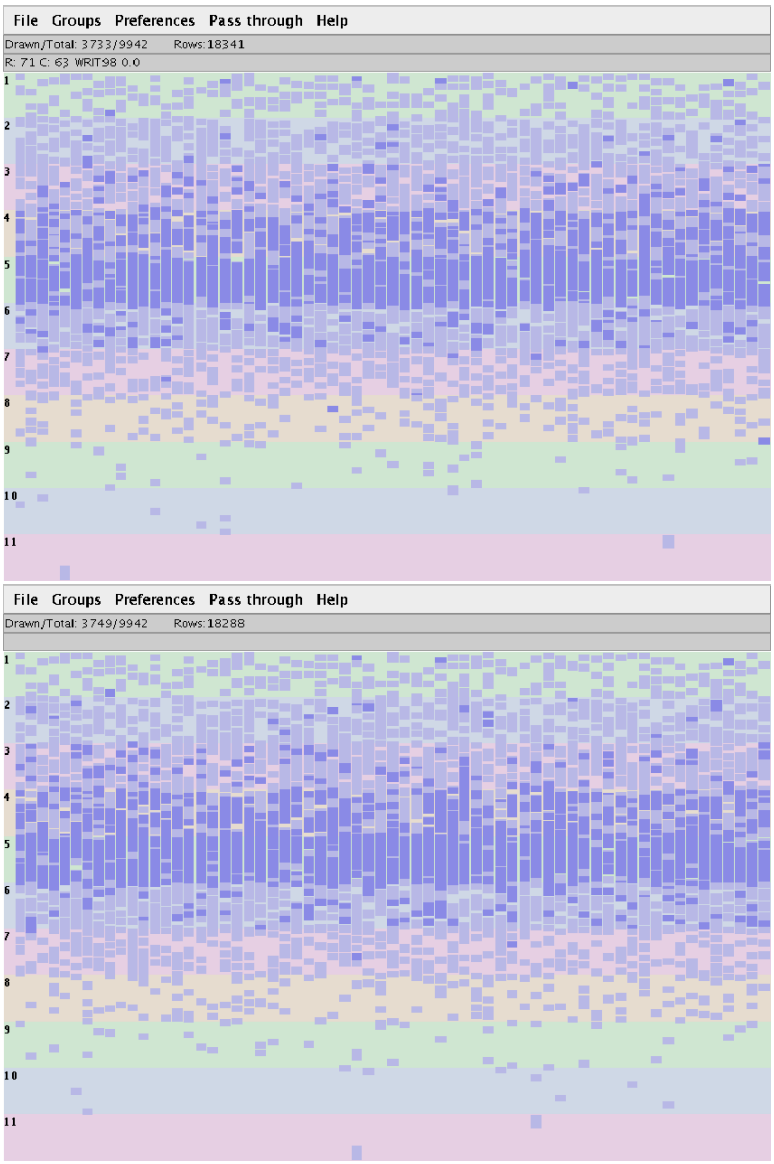

Figure 4.1: Two visualizations with different enumeration functions. Top visualization uses our approach while bottom one uses Knuth's algorithm. These two enumeration methods give very similar distributions.

where A is the size of the alphabet and p_i is the lexicographic index of the ith element of the m-set [16] [1]. This algorithm has the same complexity as ours but the resulting enumeration does not strictly follow the lexicographic order. For example, given an alphabet of {`a,b,c,d,e`}, the set {`b,c,d`} comes before {`a,b,e`}. However, this enumeration has a nice property: any m-set that contains a_i comes after all m-sets that are generated using $\{a_1, a_2, \ldots, a_{i-1}\}$, where a_i is the ith element in the alphabet. This process is especially useful when the alphabet size keeps increasing over time, since the newly generated powersets will be appended at the end of the powerset enumeration, which will leave the layout of the exiting sets untouched. As shown in Figure 4.1, Knuth's approach and ours give two very similar visualizations of the enrollment dataset, which will be introduced in Section 5.1.

4.2 SplitLine Hierarchy

Once the index in the enumeration of an itemset has been calculated, we need to create boundaries as we discuss next.

4.2.1 Challenges

All itemsets will be laid out in a 2-D grid, where each box that represents an itemset is bounded by four movable lines, which we call **SplitLines**. The rubber-sheet navigation is accomplished by moving the SplitLines. The full powerset space is extremely huge. For example, for a database that has an

[1] Background material for this approach is discussed in Volume 4 of *The Art of Computer Programming* [17]

alphabet size of 50, the total number of powersets is 1.12×10^{15}, which makes it impossible to create all SplitLines beforehand considering the memory usage. Moreover, for alphabets of any significant size, the screen space allocated to each of the boxes would be much smaller than a pixel in the display because of the cardinality of the powerset. To guarantee visibility, we need to maintain some aggregation information which will be used by the rendering engine in the rendering stage.

4.2.2 Our Approach

As shown in Figure 4.2 Left, SplitLines are boundaries of boxes which are to be rendered on the display. All boxes in the same row share the same upper and lower SplitLines. Two adjacent rows also share a SplitLine that lies between the two rows. The column case is analogous. Users are able to change the size of part of the screen by dragging SplitLines that are boundaries of that region. As shown in Figure 4.2 Right, after the second horizontal SplitLine being dragged to the left, the region on the left side of the second horizontal SplitLine is squished, while the size of the region on the right side of the third horizontal SplitLine remains the same.

A set of SplitLines provides both a linear ordering and a hierarchical subdivision of space, as in Figure 4.3. Linearly, SplitLine B falls spatially between A and C. Hierarchically, it splits the region to the left of its parent SplitLine D in two, and its range is from the minimum SplitLine to the maximum of its parent SplitLine D. The diagram here shows only the horizontal SplitLines; the vertical situation is analogous. Each SplitLine has a split value, which ranges from 0 to 1. This value indicates the relative position

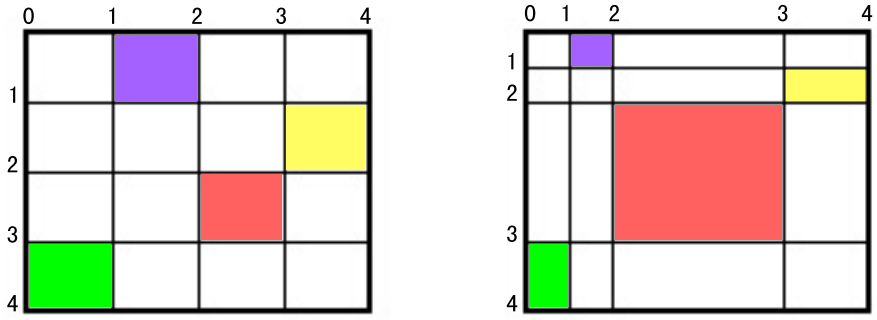

Figure 4.2: *Left*: boxes are bounded by SplitLines. Screen space is evenly distributed before dragging. *Right*: the region occupied by the red box grows bigger by dragging the second horizontal SplitLine left and the second vertical SplitLine up.

of the SplitLine with respect to the boundary.

Rendering the boxes in the correct position on the display requires calculating the absolute positions of SplitLines. PRISAD [27], a generic rendering infrastructure for information visualization applications on top of which PSV is build, makes use of a hierarchical structure to organize the SplitLines to support efficient calculation of the absolute positions of the SplitLines on the fly. When calculating the absolute value of a SplitLine, PRISAD simply traverse the SplitLine hierarchy from the root to the node that represents that SplitLine. The complexity of this operation is bounded by the depth of the tree, which is efficient enough to support real time interaction.

The two major tasks related to the SplitLine hierarchy are construction and maintenance. PRISAD is responsible for maintaining the correct hierarchical information once the SplitLine hierarchy has been successfully constructed. For example, after a user' navigation, PRISAD will update the split values when necessary to reflect the users' interaction. Detail infor-

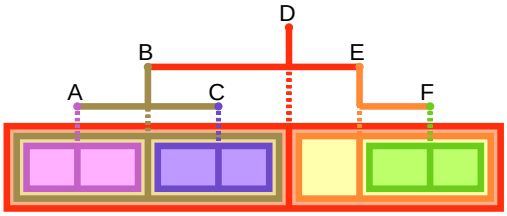

Figure 4.3: An example of a SplitLine Hierarchy.

mation on how to maintain the SplitLine hierarchy can be found in James Slack's Master's thesis [26]. The rest of this section focuses on how to construct the SplitLine hierarchy.

Since the potential powerset space is extremely huge, it is infeasible to instantiate every SplitLine considering the memory usage. Moreover, since sets are returned by the mining engine on the fly, we do not know which SplitLine needs to be instantiated until a set arrives in PSV. Therefore, PSV requires a dynamic hierarchical data structure that efficiently supports adding new SplitLines. We do so by extending the well-known **red-black tree** data structure [9] so that each red-black tree node is associated with a SplitLine. For each of the horizontal and vertical dimensions, we maintain a red-black tree. These two trees are fundamentally the same, except that the number of nodes in the horizontal tree is much smaller than that of the vertical one. We will use the horizontal tree as an example to demonstrate how we create and maintain the SplitLine hierarchy.

Construction of the SplitLine hierarchy can be further divided into two subproblems: (a) when to instantiate a SplitLine and (b) how to update necessary data structures to maintain the correct hierarchical structure. The following example gives an overview of the mapping process on a database

No.	**Set**	**Index**	**Row**	**Col**	**Lines**
1	$\{a\}$	0	0	0	1
2	$\{a, b, c, d, e, f, g, h\}$	254	31	6	31
3	$\{b\}$	1	0	1	-
4	$\{a, b, c, e\}$	93	11	5	11, 12
5	$\{a, b, d, h\}$	100	12	4	13

Table 4.1: As sets are sent to the visualizer, their enumeration index is computed and used to find the row and column in the grid where boxes are drawn.

with an alphabet size of 8. As shown in Table 4.1, as sets are sent to the visualizer, their enumeration index is computed and used to find the row and column in the grid where boxes are drawn.

As shown in Figure 4.4 top left, after the first two sets have been loaded, a single empty row visually separates the two sets, which are the first and last itemsets in the enumeration. The instantiation of the empty row is important in that it servers as a visual cue to the user that the first two sets are not immediately adjacent to each other. A corresponding SplitLine hierarchy is shown in Figure 4.4 bottom left. Figure 4.4 top right shows the visualization after all the sets have been loaded. As more sets are added, the red-black tree stores the SplitLines, rebalances, and changes the root. The horizontal SplitLines hierarchy would require $2^5 = 32$ nodes if it were statically allocated, but only 5 are needed when using dynamic allocation.

In summary, we instantiate a SplitLine when and only when (a) it has not been instantiated and (b) it serves as a boundary of at least one box.

We need to insert a node in the corresponding red-black tree once we have decided to create a SplitLine. Before we insert the node into the red-black tree, we must calculate the default relative split position of each line,

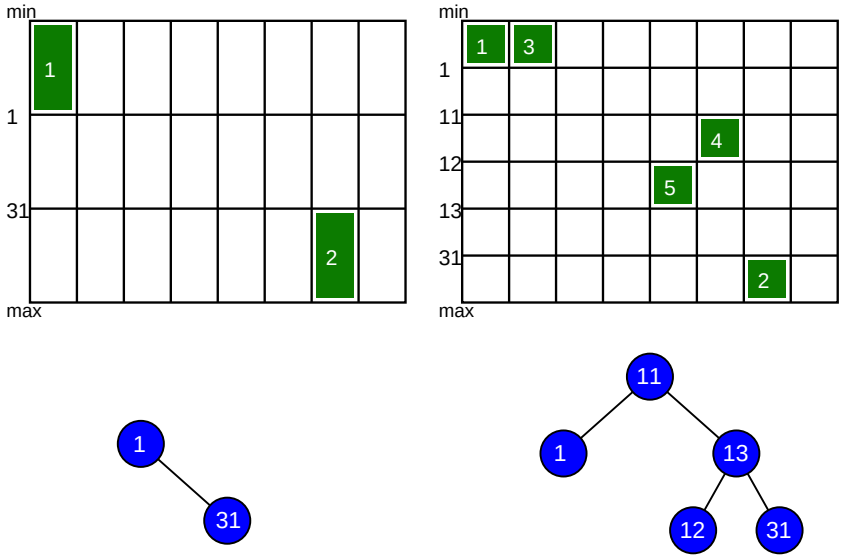

Figure 4.4: Mapping from sets to boxes with alphabet size 8. *Bottom left:* After two sets have been loaded, lines 1 and 31 have been instantiated. *Top left:* A single empty row visually separates the two sets, which are the first and last itemsets in the enumeration. *Bottom right:* As more sets are added, the red-black tree storing the SplitLines rebalances, changing the root. *Top Right:* The grid fills in with boxes, with visual separation between non-contiguous itemsets in the enumeration.

which is used to set or reset its initial absolute position so that the lines look uniformly distributed. If the number of SplitLines were a power of 2, then we could simply set each one to the split position of 0.5. When the binary tree is not perfectly balanced, then the correct value for a SplitLine is the ratio between the regions of its left and right children:

$$SV = L/(L + R). \tag{4.4}$$

Inserting a node z into a standard red-black tree with n leaves has a cost of $O(\log n)$. Node insertion is a two-phase process: first, traversing a path downward from the root node to the correct insertion point at some leaf node; second, rebalancing the tree via a series of rotations. In our algorithm, the traversal and rotation functions must also maintain split positions for the SplitLines attached to the nodes. This additional functionality does not increase the complexity of the operations.

Phase 1: At each node in the red-black tree, we store the number of its left descendants L and right descendants R. When inserting a node into the tree we increment the counters on the appropriate path. The sorting criterion for nodes is based on a key computed from the canonical enumeration of a powerset, as in Section 4.1.2. We also calculate the default relative split positions for all nodes traversed.

If the change in the split position for a line is the result of navigation, we preserve the visible configuration of lines, and update the default split position for use the next time the line positions are reset. Otherwise, when no user navigation has occurred, we want to make room for the newly inserted

line, uniformly distributing the available space between the set of SplitLines. In this case we do update both the split position and the absolute position using the new default.

Phase 2: During a rotation on a node x, there are four cases to consider: x could be either a left or a right child, and there could be a left or right rotation. Figure 4.5 Right shows the right-left case: when doing a left rotation on a node E which is a right child of its parent D, we only need to update the relative split positions of E and its right child F, because for each node in the subtrees α, β, γ, and B, its descendants and ancestors are preserved. Maintaining the L and R counts is straightforward, as described above. However, we cannot simply update the relative split positions of E and F using Equation 4.4 when the relative split positions have been changed by user navigation. Instead, we compute the new relative split positions based on the old one using Equations 4.5 and 4.6. The absolute positions of E and F remain the same after the rotation, preserving the correct SplitLine hierarchy. The remaining three cases are analogous to this one. The bookkeeping operations are local and can be done in constant time.

$$SV_F{}' = (1 - SV_E) * SV_F + SV_E \tag{4.5}$$

$$SV_E{}' = SV_E / SV_F{}' \tag{4.6}$$

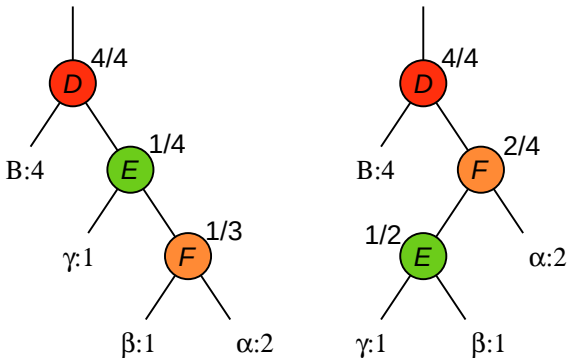

Figure 4.5: Maintaining SplitLine values through red-black tree node rotation. Here we show the case of a left rotation on node E which is a right child of its parent. Nodes are labelled with a relative split positions if those values change as a result of the rotation. The resulting absolute positions of E and F are the same as those before the rotation, which preserves the visual configuration of SplitLines.

4.3 Rendering and picking

After all the necessary SplitLines have been instantiated, the boundaries of a particular box can be computed by traversing the SplitLine hierarchy. Therefore, we can start to draw these boxes on the display.

4.3.1 Challenges

The challenges for rendering comes from two parts, underdrawing and overdrawing. Overdrawing happens when several items fall into the same block on the screen. A naïve rendering algorithm will simply redraw that block over and over again, which will dramatically increase the rendering time. At the other end of the spectrum is underdrawing. Certain boxes that should be rendered on display are missing. To avoid underdrawing while minimize overdrawing, we developed an $O(log(n) * b_v * c)$ rendering algorithm for PSV,

where n is the total number of items on the screen, b_v is the number of blocks in the vertical dimension, and c is the number of columns.

4.3.2 Our Approach

Rendering

Rendering in PSV strictly follows the rendering pipeline which is described in PRISAD [27]. The rendering process consists of three steps: partitioning, seeding, and drawing. Partitioning is a discretization process that maps the infinite-precision drawing of rectangular boxes to the desired block-level. In other words, we divide the world space into small regions of equal size, where the number of regions is determined by the number of pixels available on the display and the size of the on-screen blocks. If a particular region contains only one set, we simply draw a box on the screen to represent the set. When a region contains more than one set, which is the most common case, we need to use other visual channels to encode aggregate information in addition to drawing a box on the display. Since SplitLines also serve as the boundary of boxes, the discretization process in PSV is done by dividing the SplitLines into small regions. When the discretization process is finished, we enqueue all the boxes that need to be rendered. Finally, we just traverse the rendering queue to render the boxes one at a time.

We maintain a red-black tree, NodeTree, for each column. The item set that is mapped to a column will be attached to a node in the corresponding red-black tree, where the index of the node is the row number of that item set.

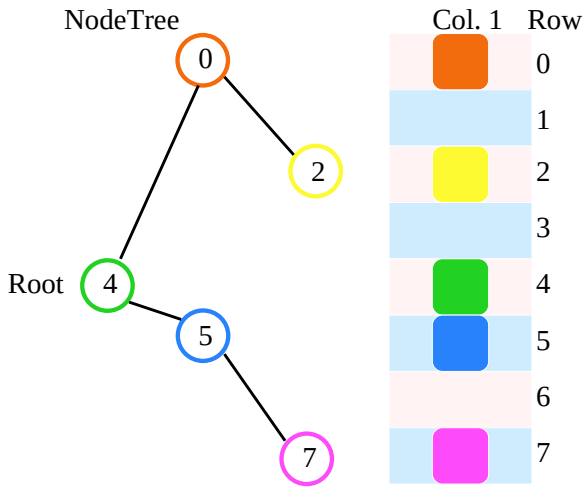

Figure 4.6: A NodeTree of Column 1.

The NodeTree provides both a linear ordering and a hierarchical organization of the item sets, as shown in Figure 4.6, which is similar to the SplitLine hierarchy. Linearly, Node 5 falls spatially between 4 and 7. Hierarchically, it splits the region to the left of its parent Node 4 in two. Moreover, Node 5 covers the region from the upper boundary of itself to the lower boundary of node 7. For a node n, we cache the following variables that are used to facilitate traversal.

- `leftChild`: a reference to the left child of n;
- `rightChild`: a reference to the right child of n;
- `parent`: a reference to the parent node of n;
- `descendant`: an integer that records how many descendants n has.
- `minLine`: a reference to one of the SplitLine in the vertical SplitLine, which is the upper boundary of n;

- `maxLine`: a reference to one of the SplitLine in the vertical SplitLine, which is the lower boundary of n;
- `aggregate`: a boolean value that indicates whether n represents more than one box;
- `aggregateNumber`: an integer that keeps record of how many boxes n represents;
- `fixed`: a boolean value that indicates whether n should be drawn in a fix size or in its actually size;

In addition, we compute the following variables on the fly using standard tree traversal algorithms:

- nodeRange: the region that is split by node n. As shown in Figure 4.6, nodeRange of node 5 begins from the upper boundary of node 4 to the lower boundary of node 7;
- nodeCoverage: the region that is covered by node n. As shown in Figure 4.6, nodeCoverage of node 5 begins from the upper boundary of node 5 to the lower boundary of node 7.

We perform a recursive binary subdivision on the NodeTree hierarchy. Recursion is dependent on either the world-space criterion that the Node has no children, or the image-space criterion that the Node's range currently subtends less than one block.

Figure 4.7 illustrates the three cases that determine whether the recursion should terminate or continue. In case 1, for Nodes 4 and 0, the node

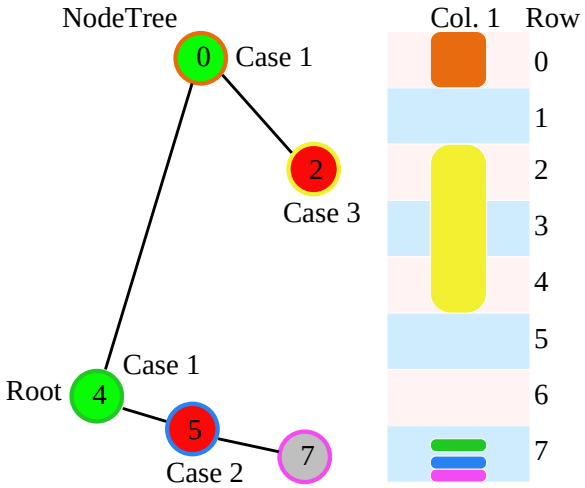

Figure 4.7: The three cases used to determine how to descend the NodeTree hierarchy.

coverage is larger than one block size and has child nodes so we descend to both children. Node 5 shows case 2 since its coverage spans less than one block. We stop recurse and draw an box to represent Node 5 and its descendant (Node 7). Finally, Node 2 is a leaf node in the NodeTree hierarchy and recursion stops.

Algorithm 1 shows the pseudocode of the enqueuing function, `enqueueRecurse(Node` *node*`)`. Line 1 to line 6 ensure that *node* does not overlap with any boxes that are already in the rendering queue. If *node* overlaps with a box b in the rendering queue, b will be updated to an "aggregate" box and rendered accordingly. Line 8 to line 16 flag *node*.fixed variable if its actual size is smaller than one block. Line 18 evaluates if *node* is a leaf node and line 20 decides whether *node*'s coverage is less than one block. The number of recursion is bounded by the number of boxes in the vertical direction, b_v. Each of the recursions requires traversing the SplitLine data structure with a complex-

Algorithm 1 Enqueuing Function: `enqueueRecurse(Node` *node*`)`

```
1: if node.minLine−node.nodeRange.minLine<unitBlockLength then
2:    node.nodeRange.minNode.aggregate = true
3:    node.nodeRange.minNode.aggregateNumber++
4: else if node.nodeRange.max−node.minLine<unitBlockLength then
5:    node.nodeRange.maxNode.aggregate = true
6:    node.nodeRange.maxNode.aggregateNumber++
7: else
8:    if node.maxLine.getPosition()−node.minLine.getPosition()≥unitBlockLength
      then
9:       node.fixed=false
10:      node.aggregate=false
11:      enqueue node;
12:   else
13:      node.fixed=true
14:      node.aggregate=false
15:      enqueue node;
16:   end if
17: end if
18: if node is a leaf node then
19:   return
20: else if node.nodeCoverage<unitBlockLength then
21:   return
22: else if node.leftChild≠null then
23:   renderRecurse(node.leftChild)
24: else if node.rightChild≠null then
25:   renderRecurse(node.rightChild
26: end if
```

ity of $O(log(n))$, where n is the total number of item sets. Therefore, the overall complexity of the enqueuing function is $O(log(n) * b_v)$.

In the second phase of the rendering pipeline, we traverse the rendering queue and render as many boxes as we can in a given time slot. This technique is called **progressive rendering**. Progressive rendering is a technique that renders the entire scene in a prioritized fashion. It always starts to render the most important part of the scene and then fills in the details. If the time that is allocated for rendering this scene is used up, progressive rendering just stops rendering the current scene and starts to draw the next scene immediately, which makes the system very responsive. In PSV, we start the rendering process from the columns that are within the user's navigation box, because user's navigation implies that this area is of great interests to the user.

Picking

Picking is a process of identifying which object was rendered at a particular (x, y) pixel location. Picking also requires traversing both NodeTree and SplitLine hierarchy. Picking is done in two steps. The first step is to traverse the horizontal SplitTree to find the correct column. In the second step, we retrieve the NodeTree from the correct column and do a traversal, which is similar to the binary search, to find the correct node if there exists one. Algorithm 2 shows the pseudocode of the picking function. The input of this function is the root node of the selected column's NodeTree.

Sometimes, the actual size of a box could be too tiny to be easily picked by the user. A well selected *fuzzConstant* is used to ensure that every box

Algorithm 2 `pickNode(Node` *node*) function.

```
1: if node is null then
2:   return null
3: else
4:   if mousePosY<node.minLine.getPosition()−fuzzConstant then
5:     pickNode(node.leftChild)
6:   else if mousePosY>node.maxLine.getPosition()+fuzzConstant then
7:     pickNode(node.rightChild)
8:   else
9:     return node
10:  end if
11: end if
```

that is visible on screen is pickable by allowing a few pixels' offset. The complexity of this function is $O(log(n))$, where n is the total number of boxes on screen.

4.4 Scalability

4.4.1 Challenges

The powerset grows huge as the alphabet size increases: a universe of only 24 items outstrips the number of pixels on the screen, and universes of over 32 or 64 items are difficult to even store in standard data formats. Our PSV system advances the state of the art by providing scalability in the size of the alphabet.

4.4.2 Our Approach

We hope that PSV can handle datasets not only with large alphabets, but also with maximum set size. In the following sections, we will show that

PSV can scale very well in both aspects.

Handling large alphabets

When the alphabet size A is large, the powerset size P is a huge number: 2^A. Dynamic allocation of the SplitLine hierarchy, as discussed above, is necessary but not sufficient. The indices in the power enumeration do not fit into an integer or a long when the alphabet size is greater than 31 or 63, whereas we support alphabets over 40,000. The naïve approach would be to simply switch data structures from integer to **bignum** everywhere that indices are used in the SplitLine hierarchy. However, computations using bignums are far more expensive than those using integers or longs, and storing them imposes a heavy memory footprint, so we would like to minimize their use. In contrast, the visualizer must store all N sets actually shown in main memory, so our algorithms are optimized for the case where $N << P$. In the current implementation, N is limited to the range of 1.5 to 7 million sets, a number far smaller than the two trillion limit of integer data storage. Operations that use the number of shown sets N can be done much more efficiently, as opposed to those that use the alphabet size A or the powerset size P.

Our spatial layout does fundamentally depend on the powerset size P, so we cannot completely eliminate bignums. The key insight is that we only need to use these high-precision values when adding SplitLines from the hierarchy as sets are added to the scene. Specifically, we use bignums in two computations: finding the enumeration index as described by Equation 4.2, and then when dividing that index by the fixed width of the grid to get a

bignum row index. The row index does need to be stored: as illustrated in Figure 4.4, a full-precision SplitLine row index may be necessary when resolving the spatial relationship between the box in question and boxes that are added or deleted later. By storing this row index as a bignum, we support lazy evaluation and avoid unnecessary computation. Although the computation of the enumeration index requires bignums, we do not need to incur the memory overhead of storing it, so we throw it away after its use in computing the row index.

Our rendering and navigation routines remain fast because we do not need to use bignums when traversing the SplitLine hierarchy. Although the bignum row indices must be stored at each node of the hierarchy, we can traverse the tree without them by maintaining pointers or object references in the node data structure linking it to its children and parent.

Extending the maximum set size

Often the dataset semantics dictate that the maximum set size is much smaller than the alphabet size. For example, it is essentially impossible to buy every item in a grocery store in one shopping trip or to take the thousands of courses offered at a university during the same term. Figure 5.3 shows that the university enrollment dataset with alphabet 4616 has a maximum set size of 13, and the market basket data in Figure 5.7 has a maximum set size of 115 out of the 1700 items in the alphabet. In contrast, although the particular software engineering dataset shown in Figure 6.1 has a maximum set size of 48 files checked in together during a bug fix out of 42,028 files in the alphabet, the domain semantics could allow a maximum set com-

mensurate with the entire alphabet. For instance, if the copyright notice on top of each file needs changing, every file in the repository would be touched. An important property of our algorithm is that there is no hardwired prior limit on the maximum set size; we can accommodate a maximum set size up to the cardinality of the alphabet itself.

Chapter 5

Case study

PSV is a general information visualization system for data mining that provides a unified visual framework at three different levels: data mining on the filtered dataset, the entire dataset, and comparison between multiple datasets or data mining runs sharing the same alphabet. In this chapter, we present a case study to showcase the utility of PSV.

5.1 Datasets

Two real datasets are used in this chapter to demonstrate the power of PSV: a student course enrollment dataset, `enrollment`, and a retail transaction dataset, `market`. In this section, we give brief introductions to these datasets.

- `Enrollment`: an itemset is a set of courses taken by a student during a particular term, which includes the student number, course numbers, and grades. The student numbers in the original dataset have been sanitized using random integers for privacy purpose. The 95,776 itemsets cover the six terms of the academic years 2001, 2002, and 2003. The alphabet size, 4616, is the number of courses offered.

- `Market`: a record in this dataset is a set of items that are bought together by a particular customer during a single visit to a store. This dataset was downloaded from UCI Machine Learning Repository [1] and had been sanitized. This dataset contains 300,000 transactions and has 1657 items.

5.2 Enrollment dataset

5.2.1 Frequent Set Mining

We choose dynamic, constrained frequent set mining as a concrete case study. Frequent set mining is a process of identifying in a dataset groupings of co-appearing objects, whose frequencies exceed a user specified threshold. The appeal of constrained frequent set mining is well known [23, 8, 20]. One key problem that has not been addressed in previous work is how to support users in choosing and changing constraint thresholds and parameters. This unsolved problem makes dynamic constrained frequent set mining a perfect case study to showcase the power of our visualization system.

At the beginning of the task, the analyst specifies a set of constraints and a frequency threshold. Sets that both satisfy the constraints and pass the frequency threshold will be returned to PSV. Finally, PSV will render the sets that are returned by the mining engine on the display to give analysts immediate visual feedback of the mining result.

PSV is able to communicate with the mining engine using a simple text protocol, which includes `constraint settings`, `pause`, and `play`. User can

[1] http://www.ics.uci.edu/ mlearn/MLRepository.html

pause the mining engine to tighten or relax the parameter settings based on the partial result.

In the current implementation of PSV, users are able to specify the following types of constraints for interactive constrained frequent-set mining:

(a) **aggregation constraints** of the form "agg(resultSet.A) θ *const*", where (i) agg is an aggregate operator (e.g., MAX, MIN, MEAN, MEDIUM, SUM), (ii) A is an auxiliary attribute (e.g., class size, class average for the student record database), (iii) θ is a comparison operator (e.g., $=, >, <, \geq, \leq$), and (iv) *const* is an integer constant. For example, the constraint "MAX(resultSet. classSize) $<$ 100" finds all the sets of courses having maximum class size less than 100.

(b) the **frequency constraint** "frequency(resultSet) $\geq$ *threshold*", which finds all the sets whose occurrences in the dataset meet or exceed the user-specified *threshold*. For example, the constraint "frequency(resultSet) $\geq$ 0.001" finds all the sets that appear in at least 0.1% of the transactions in the dataset.

(c) **containment constraints** of the form "resultSet contains *ConstSet*", where *ConstSet* is a set of constants. Here, this type of constraints finds all the sets that contains *any* of the constants in *ConstSet*. For example, the constraint "resultSet contains {CPSC124, CPSC126}" finds all the sets that contain the course CPSC124 or CPSC126.

Constraints are processed at different locations within PSV: some can handled by both the visualizer and the miner, while others are only processed

by the miner or only processed by the visualizer. Frequency constraints are computationally intensive, and are thus "pushed inside" to the miner, in order to provide as much pruning as possible. The miner can handle a combination of frequency and aggregation constraints. Sets that satisfy these miner constraints are sent to the visualizer for display. Specifying constraints for the miner is also a way to filter datasets larger than the current 7-million itemset capacity of the visualizer, which maintains all loaded itemsets in main memory to support fluid realtime exploration.

The visualizer supports aggregation and containment constraints by visually highlighting the matching sets from among those it has loaded. It can handle multiple simultaneous constraints, coloring each with a different color.

The visualizer supports immediate exploration of multiple simple constraints but has the limited capacity of 7 million itemsets, whereas the miner can handle very large datasets and more sophisticated constraints but requires a longer period of time for computation.

5.2.2 Usage Scenarios

We present four scenarios of using PSV features during a data mining task. These scenarios are built around the real course enrollment database, where an itemset is the set of courses taken by a student during a particular term. The 95,776 items cover the six terms of the academic years 2001, 2002, and 2003. The alphabet size, 4616, is the total number of courses offered. Our example user is an undergraduate course coordinator interested in finding sets of courses when taken together, so that she can minimize conflicts when

scheduling courses for the next year.

Scenario 1: She first considers medium-sized courses, and decides to start with the enrollment threshold of 100. Her first constrained frequent set query is $frequency \geq 0.005$ and $max(courseSize) \leq 100$. She watches the visualizer display as it dynamically updates to show the progress of the miner, and notices from the sparseness of the display that her initial parameter setting might not be appropriate. Figure 5.1 shows the constrained frequent sets computed so far when she pauses the computation after 10% of the itemsets have been processed. The sets are shown as a distribution of small boxes ordered by cardinality, from singleton sets at the top to 5-sets on the bottom. Hereafter we use the convention of k-sets to denote sets of size k. Each cardinality has a different background color, and within each cardinality the sets are enumerated in lexicographic order, as discussed in Section 4.1. The coordinator loosens the frequency constraint to .001, and tightens the enrollment parameter to 80 or fewer students before resuming the computation. Figure 5.2 shows the final result with the new constraints. After all ninety-six thousand itemsets have been processed by the miner, the largest constrained frequent itemset is a 9-set, whereas the the largest set in the partial result in Figure 5.1 is a 5-set.

Scenario 2: The coordinator now returns to the question of what course size would represent her intuitive idea of "medium". Instead of filtering the itemsets with the miner, she loads in the entire database in **pass-through** mode so that she can quickly explore by highlighting itemsets that satisfy different attribute constraints directly in the visualizer, as shown in Figure 5.3. She tries several values for the maximum enrollment constraint,

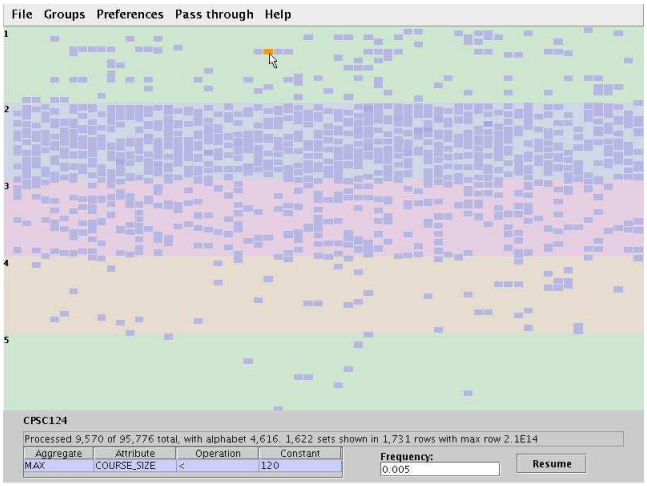

Figure 5.1: Scenario 1 for data mining with PSV. The user pauses after 10% of the itemsets are processed to loosen the frequency constraint and tighten other the parameters.

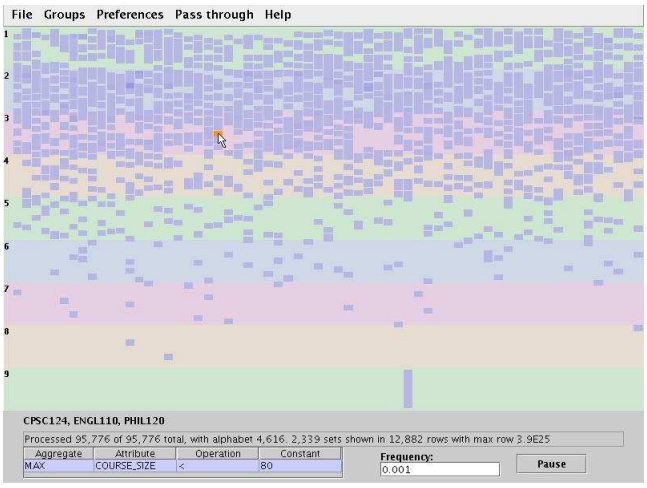

Figure 5.2: Scenario 1 for data mining with PSV. After the user's refinement, the view is considerably denser, and higher cardinality sets are shown, after all the itemsets are processed.

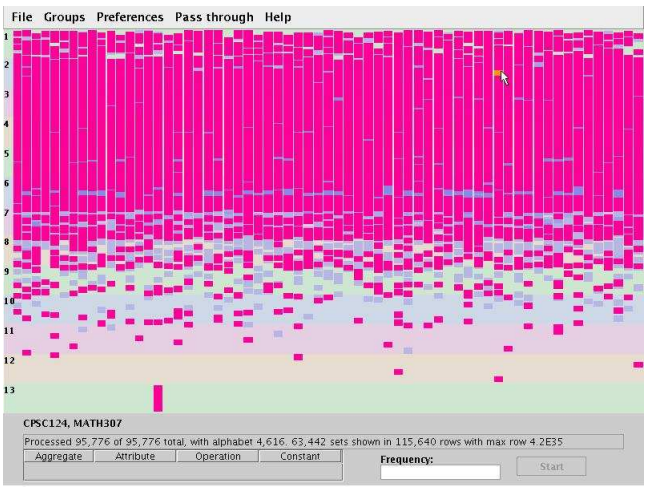

Figure 5.3: Scenario 2 for data mining with PSV. The user loads in the entire raw dataset to find good parameter settings with lightweight experimentation, using the same unified framework as when the miner was filtering data. Courses with enrollment less than 100 are highlighted.

and in less than a minute settles on the value of 100 let say.

Scenario 3: She continues by zooming in to 2-sets to investigate details that cannot be resolved from the overviews that she has seen so far, which show aggregate information about multiple sets if they fall into the same spatial region in the layout. When she zooms far enough in, each on-screen box in the zoomed-in region represents only a single set, as show in Figure 5.4. The relative ordering of itemsets is preserved in both the horizontal and vertical directions. She can still see the highly aggregated information about 1-sets on top and higher cardinality sets on the bottom, so she can easily keep track of the relative position of the area that she has zoomed. She can browse many itemsets in a few seconds by moving the cursor over individual boxes to check the course names reported in the lower left corner of the display. Those highlighted sets show the courses which are frequently

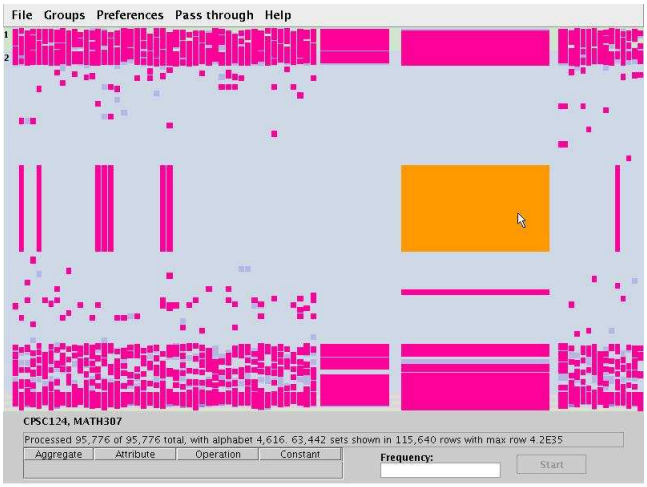

Figure 5.4: Scenario 3 for data mining with PSV. The rubber-sheet navigation technique of stretching and squishing a box shows details.

taken together by students this year. Therefore, by looking at those highlighted sets, she easily identifies which courses to avoid scheduling at the same time as `CPSC 124`.

Scenario 4: Having found a good enrollment threshold of 100 that characterizes medium-sized courses, she is ready to investigate individual courses and whether the set of courses frequently taken together changes over time. Instead of looking at the combined data over all academic years, she selects only the 2001 data, and returns to using the miner to filter with the constraints of $frequency \geq 0.001$ and $max(courseSize) \leq 100$, and clicks on the box representing the 1-set `CPSC 124`. Figure 5.5 shows that this 1-set and all of its supersets are highlighted. In other words, she can see the upward closure property of the containment relation. Using lattice terminology, the highlighted elements form a lower semi-lattice with `CPSC 124` as the bottom element, and they satisfy all the specified constraints. The

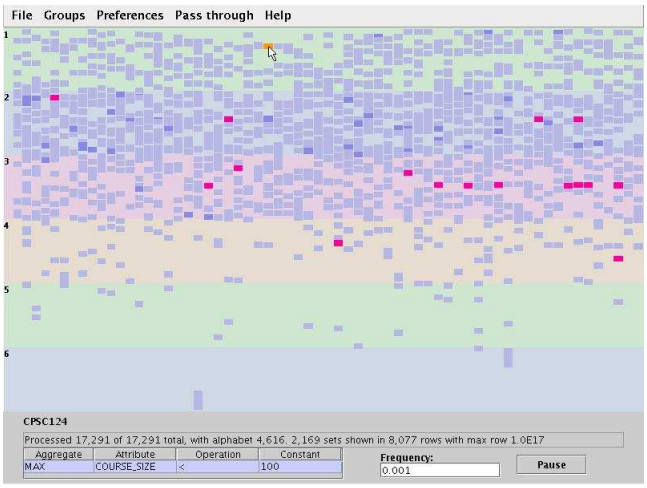

Figure 5.5: Scenario 4 for data mining with PSV. `CPSC 124` and its supersets are highlighted for the academic year 2001.

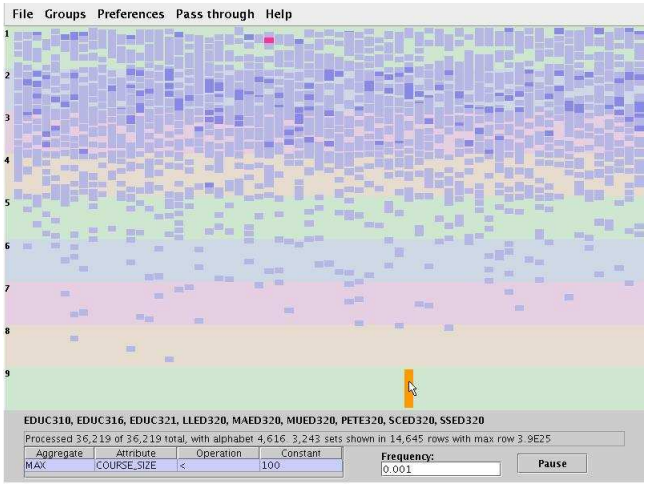

Figure 5.6: Scenario 4 for data mining with PSV. `CPSC 124` and its supersets are highlighted for the academic year 2003.

courses contained in these highlighted sets are the ones to avoid scheduling simultaneously with `CPSC 124`. She then starts up a second copy of PSV with the same configuration on the academic year 2003 data, as shown in Figure 5.6. With the 2001 and 2003 displays side by side, she can quickly spot differences and mouseover those boxes to find the names of courses that became less popular to take with `CPSC 124`.

5.3 Market dataset

Figure 5.7 shows the result of the `Market` dataset. The numbers on the left side of the windows indicate the size of the sets or the number of items that were bought together by a customer during a single shopping trip. It it easy to find that the upper part of the window is much denser than the lower part of the window, which means that customers are less likely to buy a lot of items at a time. After a close look at the final visualization, we further confirm that the sizes of a majority of the transactions are less than six. Interestingly, the top part of the region for singleton set is denser than the lower part, which implies that the products listed in the head part of the alphabet are more popular than those reside in the tail part. Unfortunately, we are unable to identify those products to conduct further analysis, since the `Market` dataset has no attribute.

5.4 Discussion

The aforementioned case study has demonstrated the power of the PSV system. PSV is good at identifying global distribution of the frequent sets

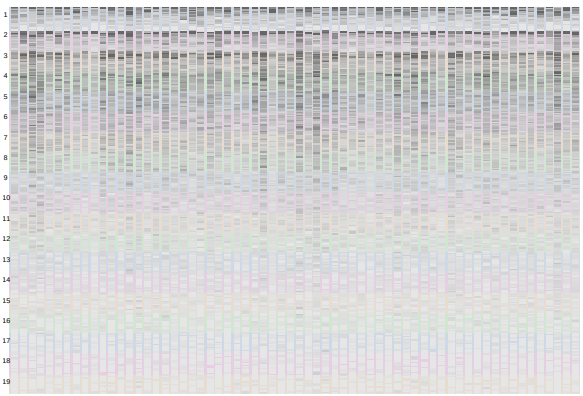

Figure 5.7: A visualization of the `Market` dataset. The `basket` dataset has over a half-million itemsets and an alphabet of 1700 items.

as well as the difference between two datasets that share the same alphabet. However, it may not be as useful as the traditional text based interface when PSV is serving as a driving interface to the steerable mining engine, since the decision of when and how to relax or tighten the constraints is solely based on quantitative information, which can be accurately and efficiently represented using numbers. More specifically, if the cardinality of the final result is small, then traditional text-based interface may be a good choice because users can get useful information immediately by reading a short list. In contrast, finding rectangles require additional navigation. However, if the users care more about the distribution of the final result, PSV is obviously a better choice, since the contextual information provided by PSV will provide information about the distribution.

In the student enrollment example, given a course, a typical user might only be interested in the top 10 courses that are taken together with that course in a specified period. The result can be shown to the user simply

using a short list. The additional contactual information does not facilitate the user's job. On the contrary, unnecessary navigation resulted from discovering the detailed information requires extra efforts.

We tried to showcase the utility of PSV in other domains, such as software engineering and algorithms. However, based on our research and interviews with several domain experts, it was difficult to find examples that required the full power of PSV. PSV has the ability to allow users to explore in a huge search space, but most of the domain specific problems do not require showing or searching the result in the context of the entire search space.

Chapter 6

Performance

In this chapter, we discuss the performance of the PSV system with both real and synthetic datasets, documenting that PSV can scale to datasets of up to 7 million itemsets and alphabet sizes of over 40,000, while maintaining interactive rendering speeds of under 60 milliseconds per frame.

6.1 Datasets

6.1.1 Mozilla Dataset

We use the Mozilla dataset to show that PSV can accommodate datasets with alphabet sizes of over 40,000. The Mozilla dataset is a software engineering dataset from the open-source Mozilla project [1]. It was downloaded from the Bugzilla [2] bug database and was compiled by Anne Ying. An itemset is a set of files that have been checked in together after a revision task is done. Each itemset contains the file names and their version numbers. There are 33,407 check-in records and 42,028 files. The final visualization is shown in Figure 6.1.

[1] http://www.mozilla.org
[2] http://www.bugzilla.org/

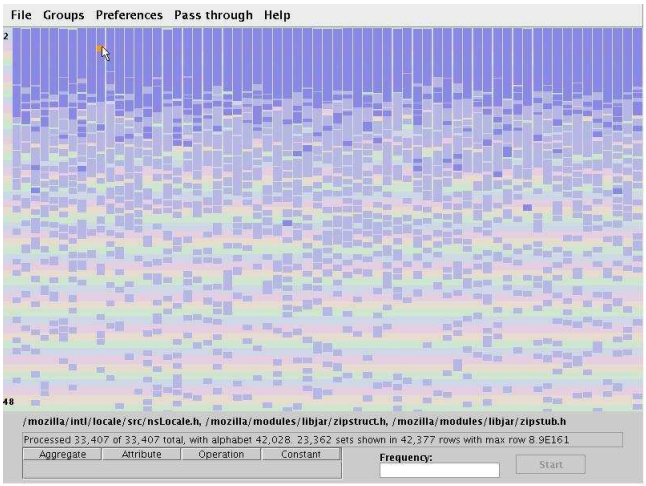

Figure 6.1: The `Mozilla` dataset has 42,028, itemsets and an alphabet of 42,028.

6.1.2 Synthetic Datasets

To showcase the memory usage and rendering performance, two synthetic datasets, `sparse` and `dense`, are generated. The `dense` synthetic dataset is the extreme case of the densest possible distribution: items in the dataset are the first 10 million items in the full powerset of an alphabet of 10,000 items. The `sparse` synthetic dataset has a distribution density roughly similar to the `market` dataset, and use the same alphabet.

6.1.3 Datasets Summary

Table 6.1 gives a summary of the datasets introduced in the previous sections.

Name	Alphabet Size	Transaction Size
`Enrollment`	4,616	95,776
`Market`	1,657	300,000
`Mozilla`	42,028	33,407
`Sparse`	10,000	10,000,000
`Dense`	10,000	10,000,000

Table 6.1: A summary of the datasets.

6.2 Scalability and Benchmarks

Figure 6.2 and Figure 6.3 shows the PSV performance results for memory usage and render speed for the aforementioned three real-world datasets: `enrollment, Mozilla, market` and two synthetic datasets. As shown in Figure 6.2, datasets that share the same alphabet size of nearly 5000 items have the same initial memory requirements. The `Mozilla` dataset has the much larger alphabet size of over 40,000 items, and requires more initial memory. The reason why a large alphabet requires a relatively larger initial memory is that we need to build up a lookup table in order to speed up the calculation of the index of a set in the powerset enumeration, where the size of the table is proportional to the size of the alphabet. Another reason why the difference of the initial memory requirements is noticeable is that most elements in the lookup table are BigIntegers, which requires much more memory than primitive variables.

PSV can handle 7 million itemsets from the `dense` dataset before running out of memory, giving an upper bound on supportable dataset size. The limits of PSV depend on the distribution of the dataset within the powerset. Sparser datasets require the instantiation of more SplitLines than dense

ones, resulting in less total capacity in the visualizer. The `sparse` dataset represent the typical use case. PSV can handle over 1.5 million itemsets from this dataset family before its memory footprint outstrips the maximum Java heap size of 1.7GB. We reiterate that when using the miner as a filter, PSV as a client-server system can handle much larger datasets than the limits of the visualizer.

Figure 6.3 top shows that the rendering time is near-constant after a threshold dataset size has been reached, and this constant time is very small: 60 milliseconds per scene, allowing over 15 frames per second even in the worst case. The render time also depends linearly on the horizontal width of the grid.

We are also interested in the performance when part or all of the on-screen sets are highlighted, since grouping and highlighting are two most commonly used features in PSV. In PSV, a set can belong to multiple groups at the same time. The color of the set is determined by the color of the group that has the highest priority. However, we do not determine the color on the fly, because the cost of querying and the cost of changing OpenGL context may be very high. In the current implementation, when a set belongs to n groups, we simply redraw the set n times in n different depth based on the group priorities. We rely on the Z-buffer to assign the correct color to the set. Based on the results of our empirical experiments, the cost of redrawing a set multiple times is much less than that of determining the color on the fly, considering that a set are not like to belong to more than 3 groups at the same time. In the worst case, when each of the n groups contains all sets, the rendering time is n times that of the normal case where no sets is

highlighted.

The main challenge with PSV was to accommodate large alphabet and maximum set sizes, a difficult goal given the exponential nature of the powerset used for spatial layout. We have done so, showing examples of PSV in use on alphabets ranging from 4,000 to over 40,000. Larger alphabets require more bits in the bignums used in the enumeration, affecting both the speed and memory usage of PSV.

All performance results are based on the following configuration: a 3.0GHz Pentium 4 with 2GB of main memory running SuSE Linux with a 2.6.5 kernel, Java 1.4.1_02-b06 (HotSpot), an nVidia Quadro FX 3000 graphics card, and an 800x600 pixel window.

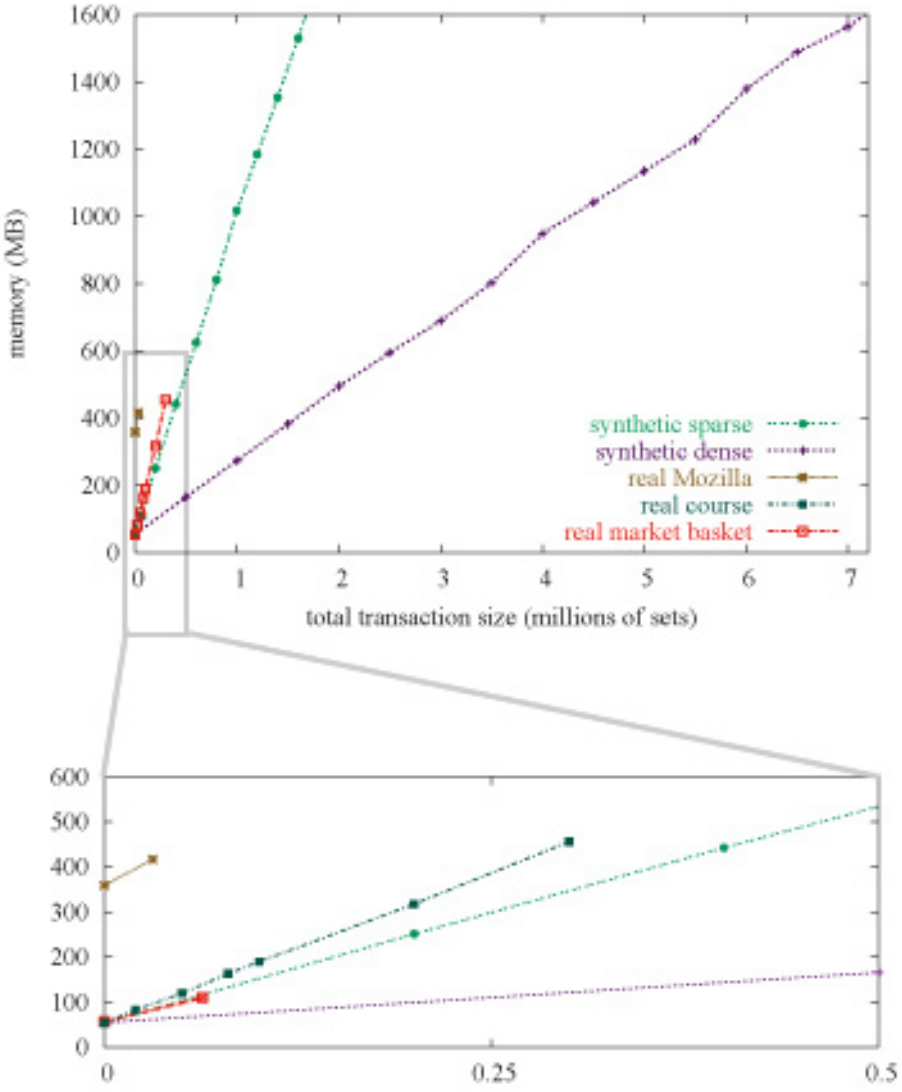

Figure 6.2: *Top:* PSV memory usage is linear in the transaction log size, and depends on the sparsity of the dataset distribution within the powerset. *Bottom:* Inset showing memory usage for small datasets.

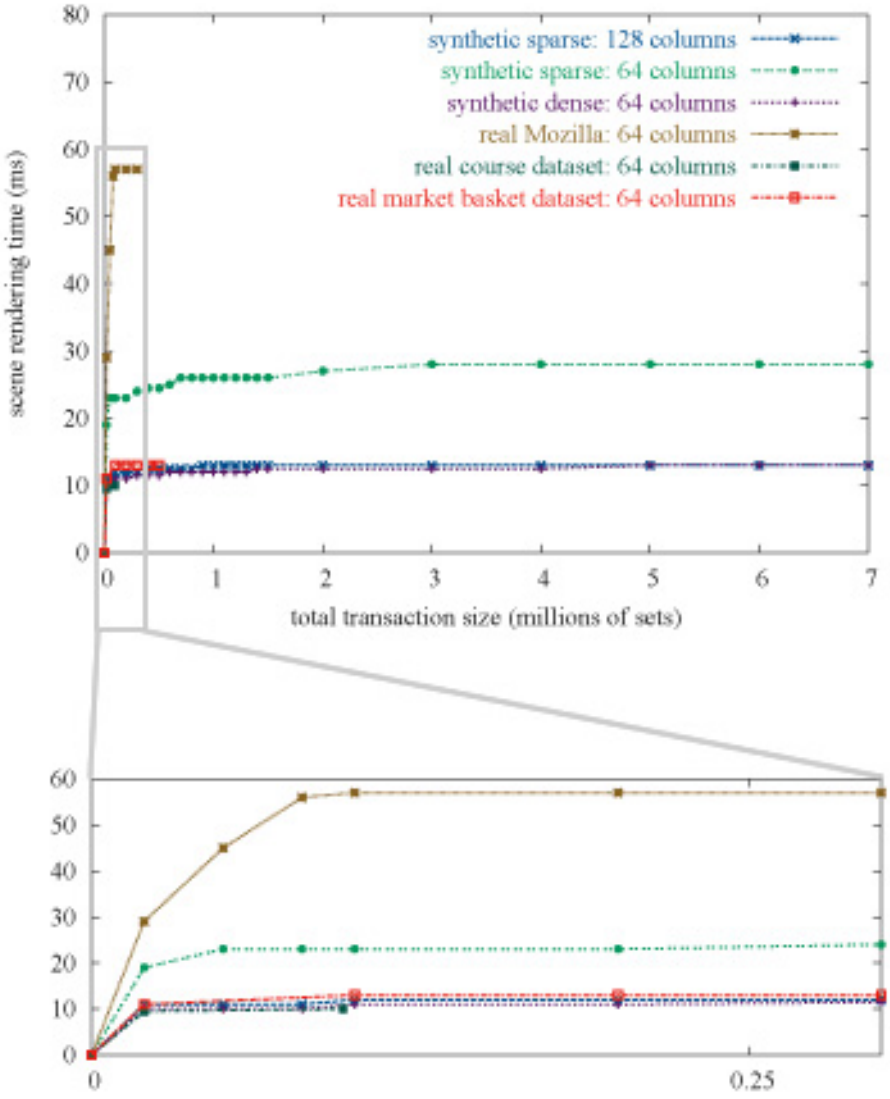

Figure 6.3: *Top:* PSV rendering time is under 60 milliseconds per frame, near-constant after passing a threshold, and linear in the width of the grid. *Bottom:* Inset showing render times for small datasets.

Chapter 7

Conclusion and future work

In this thesis, we have reported the design and implementation of the PSV visualization system for lattice-based mining of powersets. PSV provides a unified visual framework at three different levels: data mining on the filtered dataset, the entire dataset, and comparison between multiple datasets sharing the same alphabet. PSV is also connected to a dynamic frequent set mining server, showcasing how this visualization approach helps users exploit the power of steerability.

The key technical challenge for PSV is the size of the alphabet. We develop a fast scheme for computing the enumeration position of a given m-set in O(m) time, devise a dynamic data structure for managing SplitLines, and handle bignums carefully to avoid inefficiencies. We conduct case studies of the PSV system with real datasets. We also use synthetic datasets to expose the limits of the current implementation of the PSV system. The empirical evaluation shows that the current version is capable of handling an alphabet size over 40,000 items and a transaction dataset exceeding 7 million transactions. Maintaining high frame rates is critical to the success of interactive visual mining, and our framework succeeds in keeping the time to render the entire visible scene below one tenth of a second.

Although PSV has successfully achieved the scalability and rendering goals, we identify some problems that is key to justify the utility of PSV. As the main motivation of PSV, the frequent set mining itself is not a perfect example to showcase the power of PSV, since showing the final result in the context of the entire search space is not a key factor in achieving the task. This leading to a central question for PSV: when do we need the distribution information? Although it is not easy to find compelling examples in other domains, we are still in search of clear tasks and/or datasets with more attributes, where the full power of PSV will be utilized.

There are several directions of future work that we would like to pursue.

First, we would like to characterize the effectiveness of different enumeration orderings in helping users find visual patterns that convey important information about the dataset. We would like to offer users the flexibility in ordering the powerset. Specifically, people may attach different weights to all or part of the items. The second tier ordering of the powersets will be based on the weight of the item. This will ensure that itemsets that contain "heavier" or more important items will appear before "lighter" or less important itemset.

Secondly, we would like to juxtapose different datasets sharing the same alphabet in a single window. In the current implementation, users need to start two instances of PSV in order to compare two datasets that share the same alphabet. Users need to switch back and force to compare the two visualizations. It would be ideal to offer users the opportunity to put different datasets in a single instance of PSV. We may also employ brushing and linking technique, so that the user may easily identify the relationships

among different datasets.

Finally, we would like to adapt the current version of PSV to support comparison of different partial data mining outcomes for other data mining tasks, including sequential pattern mining [2], decision tree construction [4, 12], and clustering [18]. Moreover, since finding frequent set is the first step in identifying association rules, we would like enable PSV to support discovering association rules. Moreover, we also would like to keep searching the possibility of applying PSV in other domains, such as software engineering and algorithms.

Bibliography

[1] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proc. VLDB*, pages 487–499, 1994.

[2] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *Proc. ICDE*, pages 3–14, 1995.

[3] Christopher Ahlberg. Spotfire: an information exploration environment. *SIGMOD Rec.*, 25(4):25–29, 1996.

[4] Mihael Ankerst, Christian Elsen, Martin Ester, and Hans-Peter Kriegel. Visual classification: an interactive approach to decision tree construction. In *Proc. KDD*, pages 392–396, 1999.

[5] Dale Beermann, Tamara Munzner, and Greg Humphreys. Scalable, robust visualization of large trees. In *Proc. EuroVis*, pages 37–44, 2005.

[6] Stefan Berchtold, H. V. Jagadish, and Kenneth A. Ross. Independence diagrams: A technique for visual data mining. In *Proc. KDD*, pages 139–143, 1998.

[7] J. Bertin. *Semiology of Graphics*. University of Wisconsin Press, 1983.

[8] Cristian Bucile, Johannes Gehrke, Daniel Kifer, and Walker White. Dualminer: A dual-pruning algorithm for itemsets with constraints. *Data Min. Knowl. Discov.*, 7(3):241–272, 2003.

[9] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[10] W. J. Frawley, G. Piatetsky-Shapiro, and C. J. Matheus. Knowledge discovery in databases - an overview. *AI Magazine*, 13:57–70, 1992.

[11] Jianchao Han and Nick Cercone. AViz: A visualization system for discovering numeric association rules. In *Proc. PAKKD*, pages 269–280, 2000.

[12] Jianchao Han and Nick Cercone. RuleViz: a model for visualizing knowledge discovery process. In *Proc. KDD*, pages 244–253, 2000.

[13] Heike Hofmann, Arno P. J. M. Siebes, and Adalbert F. X. Wilhelm. Visualizing association rules with interactive mosaic plots. In *Proc. KDD*, pages 227–235, 2000.

[14] Tomonari Kamba, Shawn A. Elson, Terry Harpold, Tim Stamper, and Piyawadee Sukaviriya. Using small screen space more efficiently. In *CHI '96: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 383–390, New York, NY, USA, 1996. ACM Press.

[15] Daniel A. Keim and Hans-Peter Kriegel. Visualization techniques for mining large databases: A comparison. *IEEE Trans. Knowledge and Data Engineering*, 8(6):923–938, 1996.

[16] Donald E. Knuth. personal communication, November 2004.

[17] Donald E. Knuth. *The Art of Computer Programming*, volume 4. Addison-Wesley, 2005.

[18] Yehuda Koren and David Harel. A two-way visualization method for clustered data. In *Proc. KDD*, pages 589–594, 2003.

[19] Jordan Lee. Powersetviewer: An interactive data mining application. UBC CPSC533C 2004 course project final report.

[20] Carson Leung, Laks V.S. Lakshmanan, and Raymond T. Ng. Exploiting succinct constraints using FP-trees. *ACM Trans. Database Systems*, 28:337–389, 2003.

[21] Tamara Munzner, François Guimbretière, Serdar Tasiran, Li Zhang, and Yunhong Zhou. TreeJuxtaposer: scalable tree comparison using Focus+Context with guaranteed visibility. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 22(3):453–462, 2003.

[22] Tamara Munzner, Raymond T. Ng, Qiang Kong, Jordan Lee, Janek Klawe, Dragana Radulovic, and Carson K. Leung. Visual mining of powersets with large alphabets. Technical Report TR-2005-25, University of British Columbia, 2005.

[23] Raymond T. Ng, Laks V. S. Lakshmanan, Jiawei Han, and Alex Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proc. SIGMOD*, pages 13–24, 1998.

[24] S.G. Parker and C.R. Johnson. SCIRun: A scientific programming environment for computational steering. In *Supercomputing*, 1995.

[25] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *IEEE Visual Languages*, pages 336–343, 1996.

[26] James Slack. A Partitioned Rendering Infrastructure for Stable Accordion Navigation. Master's thesis, University of British Columbia, May 2005.

[27] James Slack, Kristian Hildebrand, and Tamara Munzner. PRISAD: A partitioned rendering infrastructure for scalable accordion drawing. In *Proc. InfoVis*, 2005. To appear.

[28] James Slack, Kristian Hildebrand, Tamara Munzner, and Katherine St. John. SequenceJuxtaposer: Fluid navigation for large-scale sequence comparison in context. In *Proc. German Conference on Bioinformatics*, pages 37–42, 2004.

[29] Chris Stolte, Diane Tang, and Pat Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Trans. Vis. Comput. Graph.*, 8(1):52–65, 2002.

[30] Chris Stolte, Diane Tang, and Pat Hanrahan. Query, analysis, and visualization of hierarchically structured data using Polaris. In *Proc. KDD*, pages 112–122, 2002.

[31] Edward Tufte. *Envisioning Information*. Graphics Press, 1990.

[32] Colin Ware. *Information Visualization: Perception for Design*. Morgan Kaufmann Publishers, 2000.

[33] Matt Williams and Tamara Munzner. Steerable, progressive multidimensional scaling. In *Proc. InfoVis*, pages 57–64, 2004.