Visual Mining of Power Sets with Large Alphabets

Tamara Munzner[*†] Qiang Kong[†] Raymond T. Ng[†] Jordan Lee[†] Janek Klawe[†‡]
Dragana Radulovic[†] Carson K. Leung[†§]

ABSTRACT

We present the PowerSetViewer visualization system for the lattice-based mining of power sets. Searching for itemsets within the power set of a universe occurs in many large dataset knowledge discovery contexts. Using a spatial layout based on a power set 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 power set structure is handling large alphabets, because the size of the power set 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.

Keywords

frequent set, power set, visualization, interactive data mining

1. INTRODUCTION

Human visualization can play a major role in knowledge discovery from large datasets (KDD). In this paper, we present a visualization system for lattice-based mining of power sets. Searching the **power set**, the set of all **itemsets** using the **alphabet** of items in a given universe, is a fundamental task in many KDD contexts [14]. Prime examples include association rules and frequent set methods [1], sequential patterns [2], decision trees [4], and data clustering [17]. Our PowerSetViewer (**PSV**) system advances the state of the art by providing scalability in the size of the alphabet. The power set 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. Using a spatial layout based on a power set presents algorithmic challenges, but 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.

Users can find appropriate parameter settings for data mining algorithms quickly through lightweight visual experimentation showing how various parameter settings would create a filter for the mined data with respect to the entire dataset. PSV acts as a "windshield" to make the execution of the data mining algorithms transparent. 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 itemsets pass through, so that the entire input dataset is shown to the user. Another benefit of using the power set 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 visualization system consists of two main parts, as shown in Figure 1: the power set visualization module, the **visualizer**; and the data mining engine, the **miner**. In the current version of our system, the data mining engine implements dynamic, constrained frequent set mining as a concrete case study. The appeal of constrained frequent set mining is well known [20, 8, 18]. 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. While dynamic constrained frequent set mining affects the "look" of the visualization system shown here, the contributions of this paper are not in the domain of dynamic frequent set mining, which have been presented previously [18].

[*]e-mail:{tmm,qkong,rng,jordanel,dragana@cs.ubc.ca, jklawe@cs.princeton.edu, kleung@cs.umanitoba.ca}
[†]University of British Columbia
[‡]Princeton University
[§]University of Manitoba

- Our first contribution is a visualization module using a spatial layout based on power sets that scales in both alphabet size and number of itemsets, handling alphabets of more than 40,000 items and datasets of over 7 million itemsets. The visualization system provides users with an explorable view of the full complexity of the actual distribution of the itemsets in a particular dataset with respect to the space of possibilities.
- Our second contribution is the two-part system which connects this visualization module to a dynamic frequent set mining server, showing how this visualization approach helps users exploit the power of steerability. The back-end data mining engine can act as a filter for datasets far larger than the 7 million itemset limit of the front-end visualization module. The ability of the visualization module to handle large alphabets, commensurate with what the data mining engine can support, provides the power to show the distribution of the filtered itemsets within the same visual space of all possibilities.

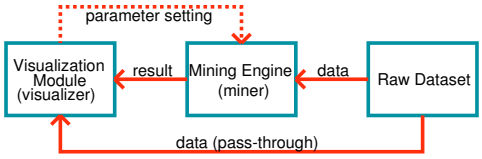

Figure 1: PSV has a client-server architecture, with a visualizer that can show either the filtered results from the miner or the raw data directly.

We begin the paper with a set of example scenarios of use in the domain of frequent set mining and a discussion of the features offered by the visualizer in Section 2. Section 3 covers related work. We present the key algorithms for supporting the huge size of the power set of a large universe in Section 4. Section 5 contains experimental results illustrating the scalability of our algorithms and a discussion of our design choices. We end with conclusions and future work in Section 6.

2. POWER SET VISUALIZATION

PSV supports interactive data mining by displaying the sets of interest with respect to the the power set; that is, the space of all possible sets. Users can choose these sets of interest with by specifying constraints for which objects should be highlighted in the visualizer or filtered by the miner.

2.1 Example Scenarios

We present four scenarios of using PSV features during a data mining task.[1] These scenarios are built around a 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 the next year's courses.

[1]A video showing PSV in action for these scenarios is available at `http://www.cs.ubc.ca/~tmm/papers/psv`.

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 2 Top Left 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 2 Top Right 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 2 Top Left 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. She tries several values for the maximum enrollment constraint, and in less than a minute settles on the value of 100 as shown in Figure 2 Middle Left.

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. 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 give her ideas of 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 2 Bottom Left 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

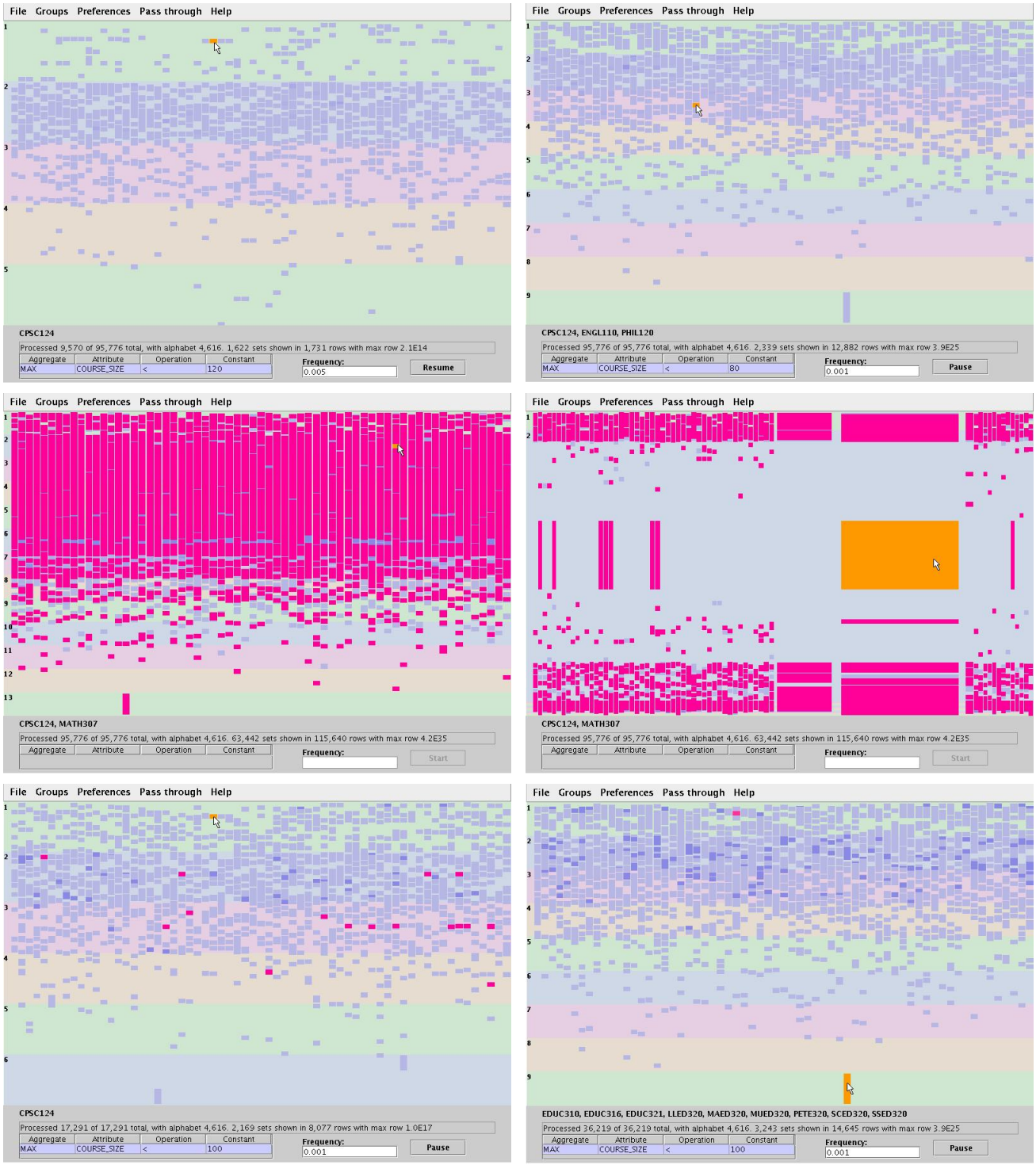

Figure 2: Scenarios for data mining with PSV *Top:* **The user steers the miner by changing a constraint in midstream.** *Top Left:* **The user pauses after 10% of the itemsets are processed to loosen the frequency constraint and tighten other the parameters.** *Top Right:* **The view is considerably denser, and higher cardinality sets are shown, after all the itemsets are processed.** *Middle:* **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.** *Middle Left:* **Courses with enrollment less than 100 are highlighted.** *Middle Right:* **The rubber-sheet navigation technique of stretching and squishing a box shows details.** *Bottom:* **Comparing different datasets that share the same alphabet.** *Bottom Left:* `CPSC 124` **and its supersets are highlighted for the academic year 2001.** *Bottom Right:* **Highlighting the same configuration for 2003 allows the visual comparison between datasets.**

form a lower semi-lattice with `CPSC 124` as the bottom element, and they satisfy all the specified constraints. The 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 2 Bottom Right. 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`.

2.2 Features

The features of PSV integrate the visual perception abilities of human users throughout the data mining process.

2.2.1 Client-Server Architecture

PSV has a client-server architecture, as shown in Figure 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 with a simple text protocol: the client sends control messages to the server, including constraint settings, pause, play, and restart. The miner sends partial results to the visualizer as they are completed, allowing user monitoring of progress. The miner server is written in C, while the visualizer client is a Java program using hardware-accelerated OpenGL with the GL4Java bindings.

2.2.2 Visual Metaphor

PSV uses the visual metaphor of **accordion drawing** [5], an information visualization technique first used for tree browsing and comparison [19]. This technique for exploring two-dimensional spatial layouts features rubber-sheet navigation [21] and guaranteed visibility [19]. 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 view 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. Although the absolute 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 allows interactive exploration of datasets that contain many more itemsets than the fixed number of pixels available in a finite display. 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 itemsets in that region happen to be smaller than the available screen resolution.

2.2.2.1 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 power set 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 power set. 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 power set space. Side by side visual comparison of two different datasets sharing the same alphabet is thus a fruitful endeavor, as described in Scenario 4 above.

2.2.3 Constraints

PSV allows the user to specify the following types of constraints for interactive constrained frequent-set mining:

(a) **aggregation constraints** such as $max(courseSize) \leq 100$, which specifies that all courses in the itemset must not exceed 100 in student enrollment. Other forms of aggregations, such as minimum, sum, and average, are also allowed. The attribute *courseSize* is an auxiliary attribute associated with each item. Examples of other attributes includes the class average, the number of assignments, and so on;

(b) **frequency constraint**, such as $frequency \geq .0001$;

(c) **containment constraints**, which find all the sets that contains *any* of the specified items. In Scenario 4 above, the containment constraint finds all the sets containing the course `CPSC 124`. This allows the user to examine the parts of the lattice of interest.

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. Scenario 1 above showcases the dynamic specification and processing of miner constraints.

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 current proof-of-concept miner handles only a single aggregation constraint, but extending it to handle multiple aggregation constraints would be straightforward. Furthermore, it can be extended to support other more general types of constraints, such as wild card matching, considered in [18].

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. This capacity is also used to visually show history, as described below.

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. Scenarios 3 and 4 above highlight the true power of PSV, in that the user first uses fast and lightweight exploration features on the visualizer side to find an appropriate value for the aggregation constraint $max(courseSize)$. Then, when the user is satisfied, the constraint can instead be pushed to the more heavyweight miner side for efficiency. The more powerful computational miner engine will then prune more sets, imposing less burden on the visualizer.

2.2.4 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. In contrast, the more general aggregation constraints that require setting several parameters are handled through a more heavyweight control panel interaction.

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. 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.

2.2.5 Monitoring

The visualizer shows several important status variables:

- *total*: the total number of itemsets in the raw dataset;
- *processed*: the number of itemsets processed so far by the miner;
- *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. Comparing *rows* with *maxrow* shows the average distribution density of itemsets; and comparing *shown* with *processed* gives the user feedback on whether the miner constraints should be changed to make the filter tighter or looser. In addition to the mining issues discussed in Scenario 1 above, tightening filter constraints is especially important if the *shown* value begins to approach the finite capacity of the visualizer. Section 5 contains a discussion of that limit, which currently ranges from 1.5 to 7 million itemsets.

3. RELATED WORK

Developing effective visualization tools for KDD is the subject of many studies, which can be sub-classified into two general categories. The first category focuses on data visualization systems. Examples include VisDB [13], Spotfire [3], Independence Diagrams [7], and Polaris [23]. These systems provide features to arrange and display data in various forms. For example, VisDB provides pixel-oriented techniques, parallel coordinates and stick figures to the user for exploring large datasets; Polaris provides a visual interface to help the user formulate complex queries against a multidimensional data cube. However, these systems are not connected to any data mining engine, nor are they designed to display data mining results. The PSV system, while allowing the raw data to be visualized and explored, provides a *unified* visual framework to the user to examine the data mining (partial) results as well. This framework allows the user to steer the data mining process midstream, and to compare between multiple data mining runs using the same alphabet.

The second category of related work focuses on visualizing mining results. Examples include decision trees [4, 11] association rules [10, 12], and clustering [17]. The visual framework proposed by Ankerst et al [4] focuses on involving the user in the building of decision trees. The visualization method developed by Koren and Harel [17] is designed for cluster analysis and validation. The visual metaphors of these systems are very different from the PSV system, which uses a spatial layout based on the power set of an alphabet.

The rule visualization system developed by Han and Cercone [10] focuses on the discretization of numeric attributes. The system uses parallel coordinates to show the mined association rules. In [12], Hofmann et al use a variant of mosaic plots, called double decker plots, to visualize associa-

tion rules. Their focus is to help user understand association rules. PSV instead operates at the level of frequent sets and constraints. Furthermore, unlike the two previous frameworks, PSV supports the steering of the mining process midstream. Again, our use of a spatial layout based on a power set is unique.

Accordion drawing was originally proposed for browsing phylogenetic trees [19, 6], and was then adapted for the task of visually comparing multiple aligned gene sequences [22]. The power set-based spatial layout used by PSV was first presented in a recent paper on a general framework for accordion drawing [5]. That paper deals mainly with the graphics challenges of navigation and rendering at interactive frame rates, whereas here we focus on issues of interest in data mining. In particular, that previous work did not support large alphabet sizes, whereas scalable algorithms to do so are the first contribution of this paper. The second contribution of this paper is in combining the visualizer with a data mining engine to provide a unified framework that handles the three levels of data mining on the filtered dataset, the entire dataset, and comparison between multiple datasets sharing the same alphabet.

4. ALGORITHMS

The mapping from a set to a box that is drawn in a display window has three main stages:

- convert from an m-set $\{s_1, \ldots, s_m\}$ to its index e in the enumeration of the power set
- convert from the enumeration index e to a $(row, column)$ position in the grid of boxes
- convert from the $(row, column)$ grid position to a pixel location (x, y) after rubber-sheet navigation has stretched and squished the grid

Figure 3 gives an overview of the mapping process. The first two stages happen once when the set is loaded in PSV, whereas the last mapping must be recalculated for every frame. We present an efficient O(m) algorithm for the first stage of computing an enumeration index e given an arbitrary set in Section 4.1. The second stage is straightforward: *row* is e divided by the width of the grid, and *column* is e modulo the width. The third stage uses the hierarchical data structures of the accordion drawing framework, and Section 4.2 describes the challenges of extending that data structure to handle large alphabet sizes. The details of the graphics algorithms that use the accordion drawing hierarchy for navigation and rendering are discussed in previous work [5].

4.1 Enumeration

The spatial layout described in Section 2.2.2.1 requires an enumeration of the power set. Although many possible ways to enumerate power sets exist, such as Gray codes [15], most of them are not suitable for creating meaningful visual patterns that are easy for data mining users to relate to. In the domain of data mining, lattice structures are often used to traverse the power set in order of set cardinality. We thus base our enumeration on a primary ordering by cardinality: all 1-sets appear before the 2-sets, which appear before the 3-sets, and so on.

Within a given cardinality, we choose a lexicographic ordering for alphabet items, again to match the power set 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, $a = 1, b = 2, \ldots, z = 26$. All computations involving sets assume that their internal item ordering is also lexicographically sorted. The challenge here is to devise an efficient way to convert between an arbitrary set and its index in this enumeration of the power set. 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. The computation is done in two steps.

Given a particular m-set, 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

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

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

$$\sum_{i=1}^{m-1} \binom{A}{i} + \sum_{i=1}^{m} \left[\binom{A - p_{i-1}}{m - i + 1} - \binom{A - p_i + 1}{m - i + 1} \right] \quad (2)$$

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 $n * k$ using dynamic programming in a preprocessing step. As we discuss in Section 4.2.2, 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.

4.2 SplitLine Hierarchy

The accordion drawing framework that handles navigation and rendering in the visualizer uses the core data structure of **SplitLines** that represent a hierarchical subdivision of space. There are two hierarchies, one for the horizontal direction and one for the vertical. A SplitLine can be interpreted in two ways, as a line or as a region, as shown in Figure 4 Left. First, the set can be considered as a linear list of lines, where each line falls between two spatial neighbors, and the line indices can be ordered from the minimum to the maximum absolute spatial position in window space. Second, it forms a hierarchical binary tree structure, where each SplitLine splits a higher-level region into two pieces. The highest-level region is the entire window, which is split by the root SplitLine. The **SplitValue** associated with a line gives the relative position of the region split as a number between 0 and 1, and dynamically changes with user navigation. The **AbsoluteValue**, the absolute position of each line in screen space, can be calculated in $O(\log s)$ time, where s is the number of SplitLines, by recursively finding the absolute location of the boundaries of each line's parent region up to the base case of the window boundaries.

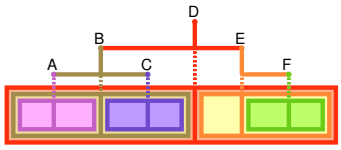

Figure 4: A set of SplitLines provides both a linear ordering and a hierarchical subdivision of space. Linearly, SplitLine B falls spatially between A and C. Hierarchically, it splits the region to the left of its parent SplitLine D in two, and its range is from the minimum SplitLine to its parent SplitLine D. The diagram here shows only the horizontal SplitLines; the vertical situation is analogous.

Previous applications that used accordion drawing statically instantiated the SplitLines hierarchy as a preprocessing step. A critical aspect in supporting large alphabets is to instead dynamically instantiate the SplitLines hierarchy, to handle the case where the distribution of itemsets to show in the viewer is very sparse with respect to the full power set. Figure 3 illustrates this important concept. We use the well-known **red-black tree** data structure [9] for maintaining nearly-balanced binary tree with an insertion and deletion cost of $O(\log n)$, where each of the n tree nodes corresponds to a SplitLine. As discussed previously [5], we extend this data structure so that SplitValues are correctly maintained when rebalancing the tree through local rotations. Dynamic layout is important for horizontal SplitLines, since the number of rows grows very large, but the vertical SplitLine hierarchy is instantiated statically because it has a small fixed width: typically 64 or 128 columns.

Figure 3 illustrates the algorithm for adding SplitLines only as needed. When a node is added to the scene, its enumeration index is calculated, then its row and column number. We check whether we need to instantiate the SplitLines flanking the itemsets: we may need to create both, just one, or no lines. In the example, the alphabet size is 8 and the fixed width of the grid is also 8. The first itemset a has enumeration index 0, calculated with Equation 2, and is mapped to grid position $(0, 0)$. Since the minimum SplitLine already exists, only SplitLine 1 must be instantiated. The next itemset a,b,c,d,e,f,g has index 254 and is mapped to the bottom row: $(31, 6)$. Similarly, the maximum SplitLine is already allocated, so line 31 is created. There is one empty row between the bottom and the top box on the screen. The third itemset, b, is right next to the first one, and the horizontal line beneath it has already been created so the red-black tree storing the SplitLine hierarchy does not change. The fourth itemset, a,b,c,e, has index 93 and is mapped to $(11, 5)$. Two new SplitLines need to be created, and the red-black tree automatically rebalances so that line 11 is at its root instead of line 1. Finally, the fifth itemset a,b,d,h has index 100 and maps to $(12, 4)$. Because line 12 has already been created, only one more SplitLine, namely 13, must be instantiated. The 255 items in the power set would require 31 horizontal SplitLines in a statically allocated grid of width 8; dynamic instantiation exploits the sparsity of the itemset distribution, creating only 5 lines.

4.2.1 *Large Alphabets*

When the alphabet size A is large, the power set 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 power set size P.

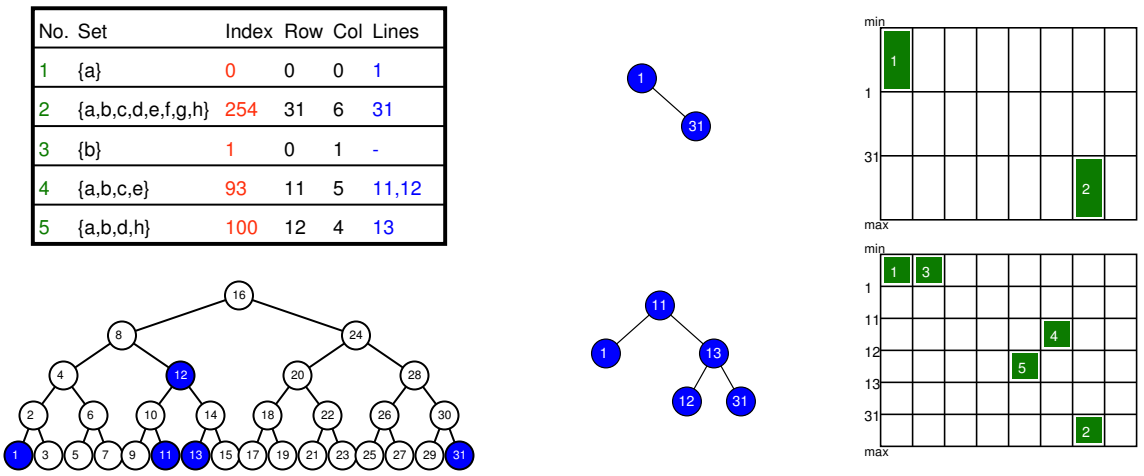

Figure 3: Mapping from sets to boxes with alphabet size 8. *Top Left:* **As sets are sent to the visualizer, their enumeration index is computed and used to find the row and column in the grid where boxes are drawn.** *Bottom Left:* **The horizontal SplitLines hierarchy would require $2^5 = 32$ nodes if it were statically allocated, but only 5 are needed when using dynamic allocation.** *Top Middle:* **After two sets have been loaded, lines 1 and 31 have been instantiated.** *Top Right:* **A single empty row visually separates the two sets, which are the first and last itemsets in the enumeration.** *Bottom Middle:* **As more sets are added, the red-black tree storing the SplitLines rebalances, changing the root.** *Bottom Right:* **The grid fills in with boxes, with visual separation between non-contiguous itemsets in the enumeration.**

Our spatial layout does fundamentally depend on the power set size P, so we cannot completely eliminate bignums. The key insight is that we only need to use these high-precision values when adding and deleting SplitLines from the hierarchy as sets are added and deleted from the scene. Specifically, we use bignums in two computations: finding the enumeration index as described by Equation 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 3, 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.

4.2.2 *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 2 Middle Left shows that the university enrollment dataset with alphabet 4616 has a maximum set size of 13, and the market basket data in Figure 6 Bottom 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 Top 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 commensurate with the entire alphabet; for instance, if the copyright notice on top of each file needed 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.

5. RESULTS AND DISCUSSION

We now discuss the performance of the PowerSetViewer system with several 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.

The real-world `course` dataset shown in Figure 2 contains 95,776 itemsets that represent set of courses taken by a student during a particular term, with an alphabet size of 4616 courses offered. A second real-world `Mozilla` dataset, shown in Figure 6 Top, contains 33,407 itemsets that are the set of source code files checked in for a particular bug fix, with an alphabet size of 42,028 files in the repository. A third real-world `market-basket` dataset, shown in Figure 6 Bottom, has 515,575 itemsets that represent simultaneous store purchases by an individual, with an alphabet size of 1700 items for sale at a large electronics retailer [16].

Figure 5 shows the PSV performance results for memory usage and render speed for these three real-world datsets. The five datasets that share the same alphabet size of nearly 5 thousand items have the same initial memory requirements.

The sixth `Mozilla` dataset has the much larger alphabet size of over 40,000 items, and requires more memory to handle the same number of itemsets. The graphs also show performance for two different families of synthetic datasets, sparse and dense. The `dense` synthetic datasets are the extreme case of the densest possible distribution: they are random samples from the full power set of an alphabet of 10,000 items. PSV can handle 7 million itemsets from this 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 power set. Sparser datasets require the instantiation of more SplitLines than dense ones, resulting in less total capacity in the visualizer. The `sparse` synthetic datasets have a distribution density roughly similar to the market-basket dataset, and use its alphabet. They 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 5 First shows that the visualizer capacity limits are linear with respect to transaction size and depend on the sparsity of the distribution of the dataset with respect to the power set. Figure 5 Third 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. The full details of the data structures and algorithms that affect render time and memory usage are given elsewhere [6, 5]. We show these graphs here to document that our extension of these algorithms to support large alphabet sizes was indeed successful.

The main challenge with PSV was to accommodate large alphabet and maximum set sizes, a difficult goal given the exponential nature of the power set 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 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.

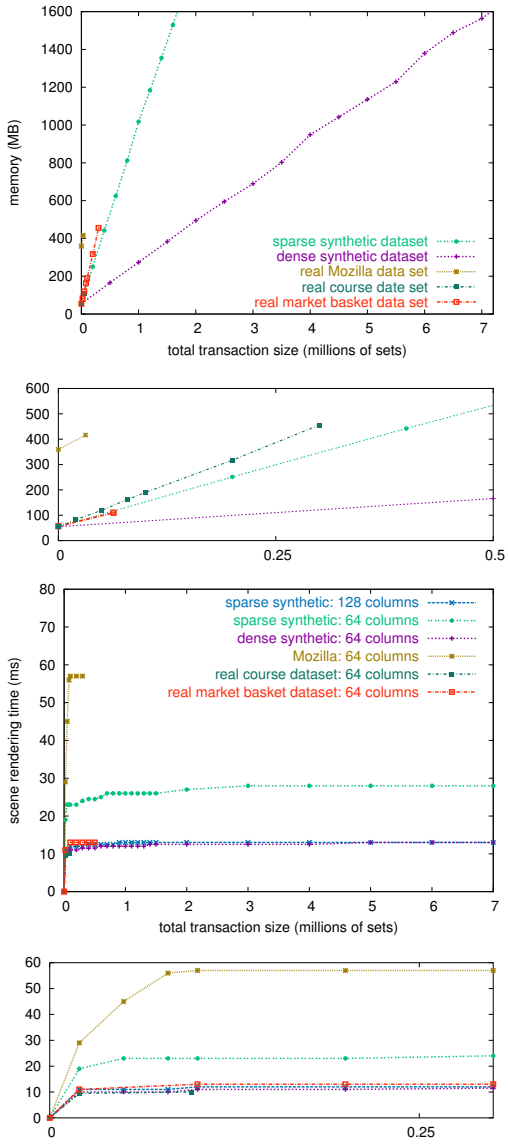

Figure 5: *First:* **PSV memory usage is linear in the transaction log size, and depends on the sparsity of the dataset distribution within the power set.** *Second:* **Inset showing memory usage for small datasets.** *Third:* **PSV rendering time is under 60 milliseconds per frame, near-constant after passing a threshold, and linear in the width of the grid.** *Fourth:* **Inset showing render times for small datasets.**

6. CONCLUSION AND FUTURE WORK

In this paper, we have reported the development of the PSV visualization system for lattice-based mining of power sets. It 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 empirical evaluation of the PSV system with both real and synthetic datasets. The latter datasets are designed 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

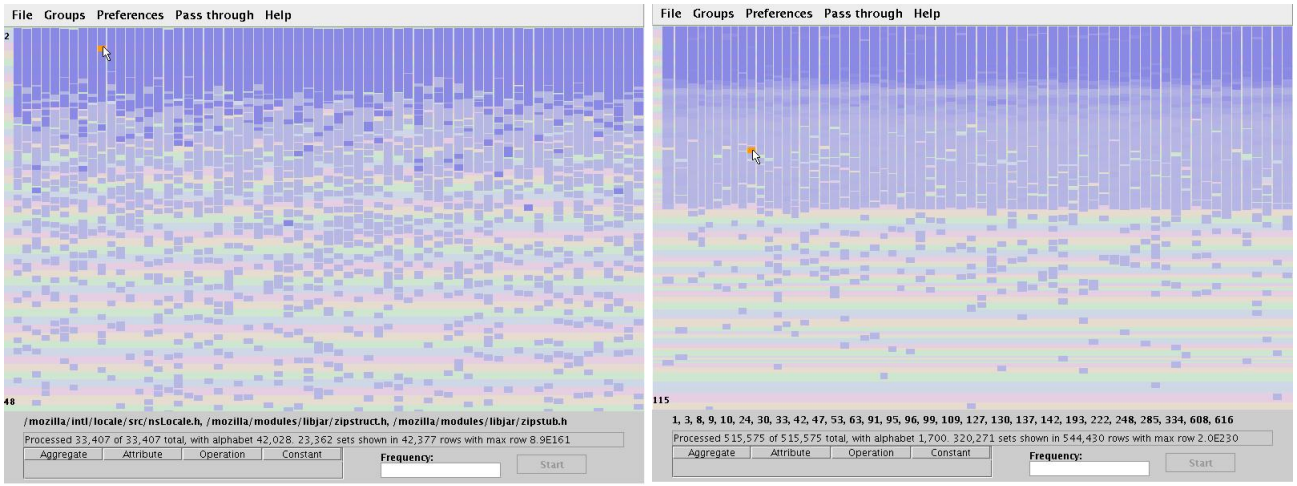

Figure 6: Two real-world datasets. *Top:* **The `Mozilla` dataset has 33,407 itemsets and an alphabet of 42,028.** *Bottom:* **The `market-basket` dataset has over a half-million itemsets and an alphabet of 1700 items.**

tenth of a second.

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. Secondly, we would like to juxtapose different datasets sharing the same alphabet in a single window. 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, 11], and clustering [17].

7. REFERENCES

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

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

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

[4] M. Ankerst, C. Elsen, M. Ester, and H.-P. Kriegel. Visual classification: an interactive approach to decision tree construction. In *Proc. KDD*, pages 392–396, 1999.

[5] Anonymous. Accordion drawing: Scalable visualization of malleable dataset surfaces. *Submitted for publication*, 2005.

[6] D. Beermann, T. Munzner, and G. Humphreys. Scalable, robust visualization of large trees. In *Proc. EuroVis*, 2005. To appear.

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

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

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

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

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

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

[13] D. A. Keim and H.-P. Kriegel. Visualization techniques for mining large databases: A comparison. *IEEE Trans. Knowledge and Data Engineering*, 8(6):923–938, 1996.

[14] J. Kleinberg, C. Papadimitriou, and P. Raghavan. A microeconomic view of data mining. *Data Min. Knowl. Discov.*, 2(4):311–324, 1998.

[15] D. E. Knuth. Generating all combinations, Section 7.2.1.3, The Art of Computer Programming Vol. 4A: Enumeration and Backtracking. Preprint distributed as `cs-faculty.stanford.edu/~knuth/fasc3a.ps.gz`.

[16] R. Kohavi, C. Brodley, B. Frasca, L. Mason, and Z. Zheng. KDD-Cup 2000 organizers' report: Peeling the onion. *SIGKDD Explorations*, 2(2):86–98, 2000.

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

[18] C. Leung, L. V. Lakshmanan, and R. T. Ng. Exploiting succinct constraints using FP-trees. *ACM Trans. Database Systems*, 28:337–389, 2003.

[19] T. Munzner, F. Guimbretiere, S. Tasiran, L. Zhang, and Y. Zhou. TreeJuxtaposer: scalable tree comparison using Focus+Context with guaranteed visibility. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 22(3):453–462, 2003.

[20] R. T. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proc. SIGMOD*, pages 13–24, 1998.

[21] M. Sarkar, S. S. Snibbe, O. J. Tversky, and S. P. Reiss. Stretching the Rubber Sheet: A Metaphor for Viewing Large Layouts on Small Screens. In *Proc. UIST '93*, pages 81–91, 1993.

[22] J. Slack, K. Hildebrand, T. Munzner, and K. St. John. SequenceJuxtaposer: Fluid navigation for large-scale sequence comparison in context. In *Proc. German Conference on Bioinformatics*, pages 37–42, 2004.

[23] C. Stolte, D. Tang, and P. Hanrahan. Query, analysis, and visualization of hierarchically structured data using Polaris. In *Proc. KDD*, pages 112–122, 2002.