PowersetViewer: An Interactive Data mining Application

Jordan Lee

CS 533C: Information Visualization
Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver BC Canada V6T 1Z4
jordanel@cs.ubc.ca

ABSTRACT

In the Information Age, data-mining has become a valuable tool. As long as there are needs and benefits from detecting and analyzing trends, there will be a need for data-mining. Traditionally, an expert in the field is required to search for "gems" of information from hunches and intuition. This task is generally tedious and ineffective as existing support applications only work to confirm these hunches. We introduce PowersetViewer to empower the user with an application that detects, rather than confirms, trends. By taking advantage of human precognitive abilities and intuition, we believe that this application will be highly scalable to large datasets.

1. INTRODUCTION

In the past, when record keeping and data acquisition were in their infancy, searching for trends within datasets was a relatively small task. This task was traditionally small enough that it could be completed without the aid of complex applications and relied mostly on human intuition.

As data acquisition techniques advanced, datasets began to grow exponentially[1]. Initially, throwing computing power at data-mining applications was sufficient. The algorithm would be pointed in a general direction and it would continue to search blindly. Without the help of human intuition, it is possible that the algorithm could get lost or search in the wrong direction[2]. We needed a way to reintegrate the user into the algorithm. To answer this issue, data-mining applications began to incorporate artificial intelligence assigned with the responsibility to make decisions throughout the algorithm. However, artificial intelligence research is expensive and still relies heavily on computing power.

Today, data acquisition techniques are advancing as fast or faster than Moore's law claims we can keep up[1]. A significant change in the progress of data-mining is required to keep up with the advances of data acquisition. We believe that by bringing human intuition back into the process, we will be able to overcome the computing power bottleneck.

2. POWERSETVIEWER

PowersetViewer is a visualization tool to enumerate and display a power set. More specifically, display the interesting sets within a power set. Interesting sets are defined as sets that are either of interest to the user or of interest to the data-mining engine. User-specified interesting sets could include sets based on hunches (confirmation). Data-mining specified interesting sets could include sets occurring frequently in the dataset (detection). In practical applications, interesting sets are scattered sparsely and have relatively low cardinality as compared to the size of the dataset's alphabet. This is a key property of data-mining and was taken into consideration throughout the design phase.

We have developed a series of tools to ensure that users have all the features traditionally found on data-mining applications. And, combined with several information visualization techniques, we believe that PowersetViewer will work simultaneously as a detection and confirmation data-mining tool.

2.1 Data-mining Engine Driver

A driving interface is required to allow the user to control the data-mining engine throughout the search algorithm. It is unlikely that an expert would be able to successfully guess the location of trends. If this were the case, confirmation techniques would suffice. However, as datasets grow the likelihood of knowing where to look decreases dramatically. Therefore, we require trend detection techniques or rather, techniques that enable the user to detect trends.

To accomplish this, we ensure that information received from the data-mining engine is displayed immediately (discussed later) so that the user can make decisions as soon as the information is made available. At any time during the search, the user can interrupt the data-mining engine and tighten or relax constraints. Should he believe that the data-mining engine is thrashing or on a completely useless course, a new set of constraints can be supplied.

[1] http://bioinfo.weizmann.ac.il/cards/knowledge.html
[2] http://www.it-analysis.com/researcharchivepdf.php?id=259

Currently, we use standard statistical calculations to filter the entire dataset. The user can choose from various aggregate functions such as: MAX, MIN, MEAN, MEDIAN, and SUM. Multiple constraints can be applied simultaneously with boolean compound operators. We assume AND operators to hold higher precedence than OR operators. Therefore, A AND B OR C is equivalent to (A AND B) OR C. A sample constraint setup could be:

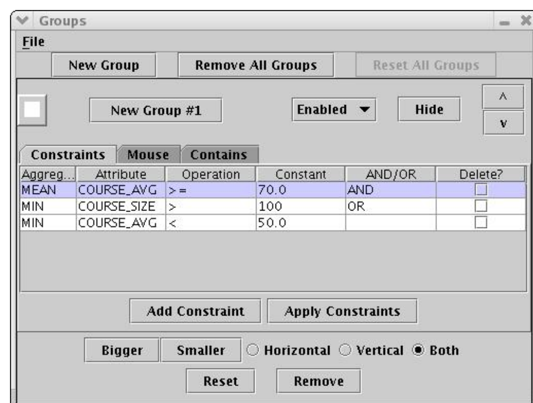

Figure 1: Constraint Specification

Upon receiving these constraints, the data-mining engine would return all student records that have an average course mark greater than or equal to 70% in classes with greater than 100 students and all student records that have one or more failed courses.

2.2 Display and Layout

In order to display an enumerated power set, we require a visualization method that could display vast amounts of information in a grid style configuration while being scalable to grid sizes exceeding the number of pixels available. We also required a method of navigation that enabled a focus+context style zooming. We decided that instead of developing the visualization infrastructure from scratch, we could build upon the AccordionDrawer package[3] (developed by Dr. Munzner). Of particular interest is the quad tree infrastructure, guaranteed visibility and progressive rendering.

2.2.1 Quadtree Infrastructure

We used AccordionDrawer's quad tree to base our layout style. As sets are returned from the data-mining engine, they are assigned to cells on a grid. These grid cells compose the bottom layer of the quad tree. Each grid cell has a parent cell that can hold up to 4 children. We build up each layer until there is one root cell. This infrastructure allows us to reduce rendering costs much like how mipmapping works. That is, if we have 1 pixel to draw 40 grid cells and we require a minimum of 1 pixel per grid cell, we only need to render the grid cell at the level where it contains all 40 cells. Thus, we have reduced rendering overhead with the penalty of requiring O(n log n) memory and a lookup of O(log n).

2.2.2 Guaranteed Visibility

As the size of the dataset grows, we reach a point where we have fewer pixels than we do sets. This point actually occurs very early as we only require an alphabet size of 20 before we start exceeding the capabilities of the standard display (1024x768). Therefore, the average screen space allocated to a cell is less than 1 pixel. However, to prevent information loss, we need to guarantee that all interesting sets are drawn. A grid cell is coloured if it contains one or more descendent grid cells that are noted as interesting. A coloured grid cell is guaranteed at least one pixel.

2.2.3 Progressive Rendering

When the number of entities needing to be drawn reaches a high enough number, we eventually reach a bottle neck around the rendering hardware. As we need to be scalable to multiple systems, we must ensure that we do not overload the rendering hardware and cause a slow-down.

We aim for a minimum of 20 frames per second (FPS). To accomplish this, the entities to be drawn are broken down into small chunks that are fed to the rendering engine as time permits. If we run out of time before a frame is complete, we draw it as is and continue drawing in the next frame.

2.2.4 Layout

To assign sets to cells, we chose a lexicographic layout strategy. We enumerate the power set in increasing cardinality with singletons first and doubletons next and so forth. Users have access to a toggle that colours the background grid with areas of similar cardinality coloured together. Four light pastel colours are used to differentiate between differing cardinalities. We chose unobtrusive colours since the main focus of the application is the visualization of the interesting sets and not the visualization of the entire enumerated power set. (See Figure 2).

2.3 Navigation and Zooming

An integral function of this application is the ability to “drill down”, manipulate and investigate sets and trends. Because we have chosen to display the entire power set at the same time, we do not support panning. Additionally, we found no benefits in supporting rotations. Therefore, the main emphasis is on zooming.

In a traditional 1024x768 pixel display[4], we can draw less than a million pixels. Thus, we can display a maximum of a million distinct cells (default cell size of 1 pixel) or a dataset with alphabet size 20. In order to work past this limitation, we take advantage of the AccordionDrawer's infrastructure and zooming capability. The user is able to grab and manipulate the position of grid lines to enlarge or decrease areas. Limitations are set to prevent the user from

[3] developed by T. Munzner, F. Guimbretiere, S. Tasiran, Li. Zhang, and Y. Zhou for TreeJuxtaposer

[4] Gathered from several random popular websites.

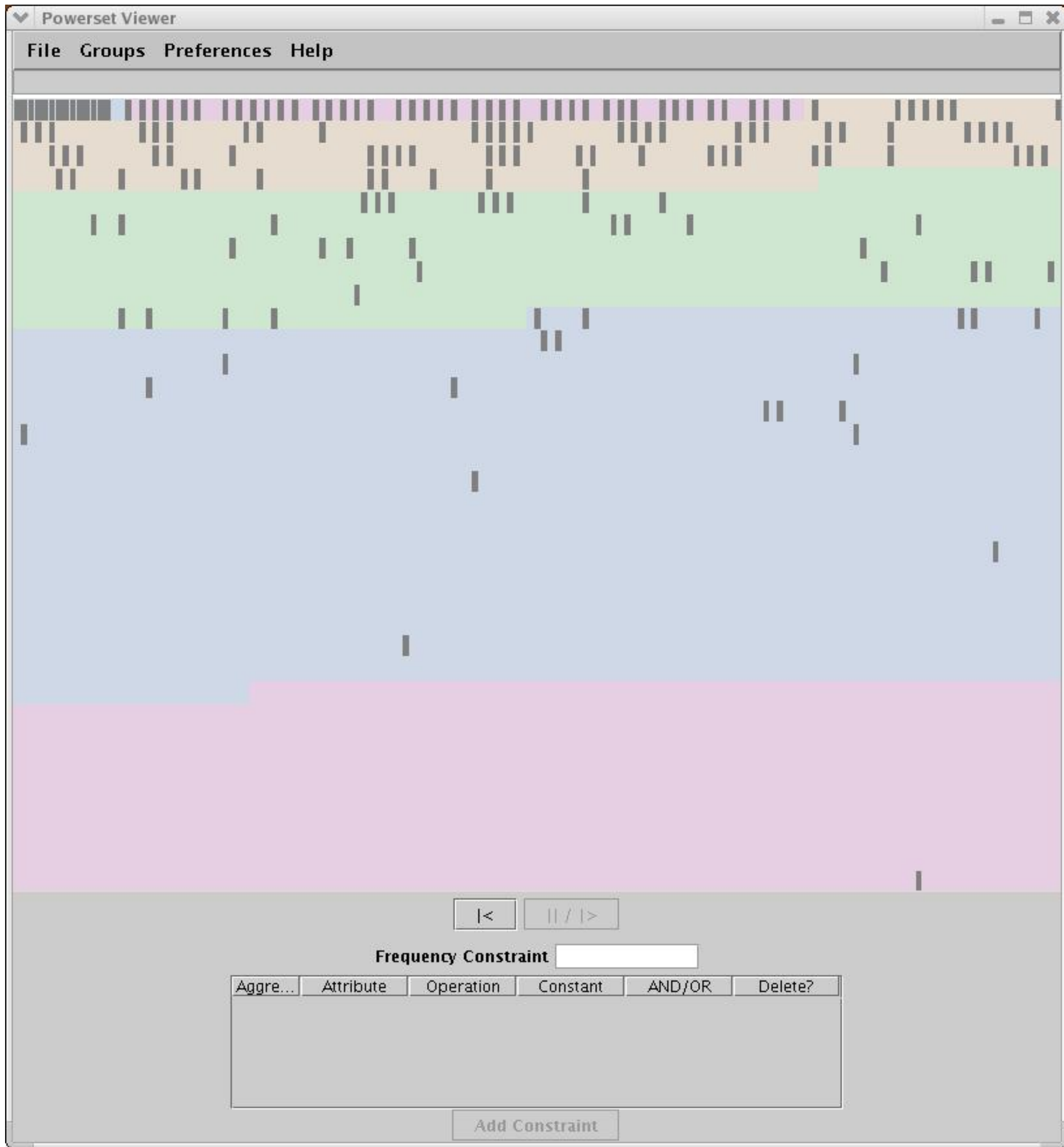

Figure 2: Lexicographic Ordering with Background Colours

dragging edges to the absolute window borders and “losing” information. A built-in safety region is set up on the edges.

In practical applications, we found that the interesting sets exist only in areas of low cardinality. Our choice of layout results in a clustering of sets in the upper areas of the grid. To maintain a high information density, we enable automatic zooming. We begin with a small grid. As sets are returned from the data-mining engine, they are added to the grid. If we occupy a cell outside of the current visible range, we automatically zoom out to encompass the new interesting set.

We found that until the grid is sufficiently large enough, virtually all new sets occur outside of the visible range. Thus, the grid was constantly growing, and user interaction was impossible and useless. At a cost of adding a delay between the data-mining engine and the visualization engine, we introduced a minimum time between re-zooms. A delay under 1 second caused confusion while a delay over 2 seconds felt like the server was responding too slowly. A

decision was made to use a minimum of 1 second between re-zooms.

2.4 Animation

We require animation when zooming to ensure that the user does not lose context of the navigation. As discussed above, there are two main zooming scenarios: user-controlled zoom and automatic resize zoom.

For the user-controlled zoom, we recalculate and redraw the display as long as the mouse button is held down. This animation gives the user important feedback regarding his actions. When rendering large datasets, we are unable to draw the entire dataset in $1/20^{th}$ of a second to maintain 20 FPS. Therefore, we use the AccordionDrawer progressive rendering technique to ensure that only the areas of interest are drawn and, time permitting, the surrounding areas.

The second instance of zooming occurs with the application's automatic zooming and re-sizing. Initially we designed a jump-cut style re-zooming. However, we found that it was disorienting and the user spent time re-assimilating the location and layout of the grid. We introduced animation to this re-zooming process with no slow-down or speed-up techniques. Steps are linearly interpolated according to a set number of animation steps.

2.5 Groups and Marking

We allow the user to create 3 types of groups. A group is a collection of sets that are assigned a colour and can be resized separately. There are a maximum of 6 groups and a set can belong to any number of groups. If a set belongs to multiple groups, it adopts its colour according to group priority which is specified by the user.

Three methods of adding sets to a group are possible:

a) User-defined constraints

Since querying a data-mining engine for results can be a lengthy process and depends on the size of the dataset, we realize that it is more convenient to encompass low cost queries in the client application. The interface is identical to the initial server query interface and is analogous to a GUI style SQL query. The result of Figure 3's query can be found as Figure 5.

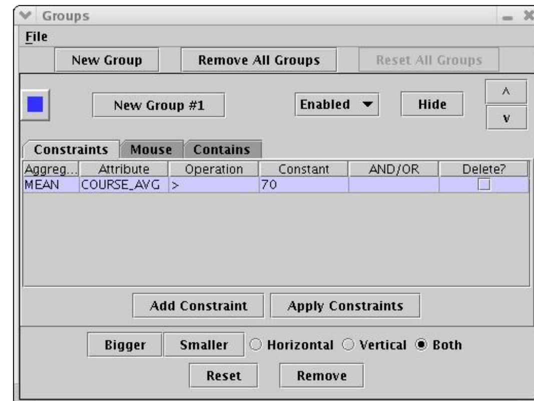

Figure 3: Constraint Selection

b) Mouse selection

For groups that do not conform to logical patterns in the data, we allow the user to manually select sets. Two types of selection are available: individual picking and batch selection. Initially, we had defined the batch selection algorithm to select all sets contained between a rectangle defined with opposite corners at the start and end drag points. However, we found that there was no correlation between sets across rows. So, we changed the selection algorithm to select all sets between the start and end sets going along rows.

c) Filter

In instances where we want to focus on a data field that contains categorical data, we can not use SQL type queries. We list all the possible values in a list and allow the user to select all interesting values. All sets that contain one or more items that have an interesting value is selected. For example, if we are interested in filtering over departments in the student record scenario, we can select all student records that have students enrolled in at least one CPSC class and at least one MATH class.

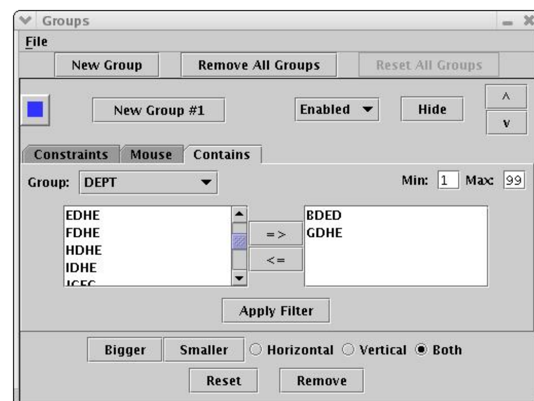

Figure 4: Filter Selection

2.6 Implementation

PowersetViewer is implemented in Java. The base visualization package, AccordionDrawer, was developed by Dr. Munzner. We extended this package and developed the AccordionPowersetDrawer package. Inspiration for this package came from the AccordionSequenceDrawer package. Our package describes the basic components of the power set including the basic visualization components, PowersetGridCells and SetNodes, and the datastructure to hold these components, Powerset. This package contains 3600 lines of commented code.

The PowersetGridCell objects are the components of the AccordionDrawer's QuadTree class. They are responsible for keeping track of the components that do the actual drawing. It stores its location within the grid and a path to the root. These objects populate the QuadTree on all levels from the root to the leaves. And, they know their exact position on the screen to help in picking.

The SetNode objects are the visual components that are attached to a PowersetGridCell. Each SetNode keeps track of its set, its position in the grid (row and column) as well as its cached colour. It is also responsible for drawing itself given a reference to the AccordionDrawer's drawing method.

The Powerset object contains the collection of sets as well as methods to add and remove sets on the fly. Because we are attempting to store an enumerated power set, we need to use a collection object that is not indexed by integers. Integers have a maximum value of 2^32 and would effectively limit us to a power set of alphabet size 30 as discussed later. Additionally, we know that we require a collection that will use an amount of memory proportional to the number of interesting sets and not the size of the power set. Finally, we expect to have a sparsely filled power set. Thus, we determined that the best collection would be Java's HashMap[5]. We use Java's BigInteger[6], an arbitrarily large number, as a key to the map. We noticed an extreme slow down when we transitioned to from integers to BigIntegers. To counteract this slow-down, we store and pass integers and use these integers to lookup the BigInteger keys to the main HashMap. This bridge also allows us to use the default parameters in AccordionDrawer as it passes around integer parameters.

All GUI widgets, interfaces and visualization features are implemented in the application package, PowersetViewer, with nearly 10000 lines of commented code.

3. SAMPLE SCENARIOS

We choose to use sample scenarios from a student record database. We have access to old student records from the University of British Columbia. We are yet unable to support an alphabet size as large as the university's course list. Therefore, we generated a smaller database with the same table setup on a smaller scale.

3.1 Finding popular courses combinations

We begin with a simple database query scenario. We setup the server constraint to search all courses but with a frequency constraint of 0.1%. That is, we want to display all sets of courses that are taken by at least 0.1% of the student population.

As sets are returned to the visualization engine, the user can determine if this query is returning too many or not enough sets. He can then pause the query, relax or tighten the frequency constraint and resume the query.

Suppose the user would like to confirm a hunch that students who take Computer Science courses prefer to take Philosophy over Anthropology courses. He can set up one group to display students that take both CPSC and PHIL. And, another group to display students that take CPSC and ANTH. By hiding all other sets to reduce clutter, he could prove or disprove his results. See Figure 6 for results.

By hovering over the remaining sets, he could perhaps determine other popular courses that students take in conjunction with the above two combinations. Thus, possibly leading him in other interesting directions.

3.2 Finding course combinations (3 courses) that result in high averages

A different approach must be used to solve this query than the one used above. We must query the data-mining engine with a low frequency constraint and constraints on the course averages. The low frequency constraint informs the server that we are interested in all combinations of courses taken. The constraint on the course average ensures that we receive only sets of courses with a combined high average.

The data-mining engine pays no attention to the size of each set. So, the user must specify the desired set size on the visualization side. By hiding or fading out all sets of cardinality not equal to 3, we have narrowed down our query to our desired specifications. See Figure 7.

4. EVALUATION

[5] http://java.sun.com/j2se/1.4.2/docs/api/java/util/HashMap.html

[6] http://java.sun.com/j2se/1.4.2/docs/api/java/math/BigInteger.html

PowersetViewer was designed with scalability in mind. We have the intent of being able to support datasets with alphabet size of 50,000 items. However, with the assumption that the max set size is significantly lower than the alphabet size. A max set size of 50 seems feasible, albeit unprecedented. At this stage, we have completed the tools and features of the application and are now concentrating on scalability.

4.1 Strengths

We believe that we have accomplished the majority of our design goals. The majority of visualization techniques employed are highly scalable. Our rendering time is limited not by the size of the dataset but by the size of the display. The quad tree structure ensures that we only render cells that are guaranteed not to be culled. Additionally, the progressive rendering technique allows us to maintain a pleasing 20 FPS[7] to ensure adequate feedback to the user.

Our zooming and navigation techniques coupled with guaranteed visibility ensure that all information available is displayed. We avoid occlusion and information loss by using a focus+context style navigation in 2D. And, our automatic zooming ensures high information density without altering the layout or risking confusing the user.

4.2 Weaknesses

Although we have successfully implemented the prototype, we are yet unable to display a power set with an alphabet size larger than 30. Since we index all possible sets within the power set, our limitation lies with the maximum size of an integer, 2^32. To address this problem, we reduced the max possible set cardinality to 10. This resulted in an increase of the alphabet by more than 50%. We are now able to support set sizes of 45 with a max set size of 10 items.

We attempted to use Java's BigInteger class to remove the max integer constraint. However, we are unable to use primitive arrays since we can only address locations with the integer primitive. Therefore, we had to re-implement our array data structures with structures that were not limited to integers. This resulted in an unacceptable slow-down of the application and so a new approach will need to be devised.

4.3 Benchmarks

Our preliminary benchmarks demonstrate a nearly logarithmic growth on datasets generated from the lowest cardinality sets. This results in all sets clustering at the top and filling downwards. We notice the initial memory cost to be unusually high compared with the amount of memory required to store the sets.

MEMORY (MB)	NUMBER OF SETS
74	10M
75	1M
74	100,000
73	10,000
58	1,000

Table 1: Low Cardinality First (Artificial) – Alphabet size 15

In efforts to have a more realistic scattering of sets, we benchmarked PowersetViewer with smaller sample sizes but with sets that are scattered across the dataset again with significant clustering across the lower cardinalities. We notice the expected linear growth with respect to the set size.

MEMORY (MB)	NUMBER OF SETS
72	263
73	244
71	168
72	160
70	127
70	45
72	30
71	10
64	0

Table 2: Random Generated (Artificial) – Alphabet size 15

5. LESSONS LEARNED

Over the span of this course and project, we learned that:

- Designing a user interface that is intuitive and functional is no easy task. We went through several iterations of various GUI widgets before the current design was settled on.
- In order to implement a highly scalable application, every design step must take scalability into consideration. Unfortunately, we did not foresee the limitations that the base visualization engine would introduce into the application.
- In comparison with other high-level programming languages, Java uses extreme amounts of memory. However, it did allow for a quicker implementation time.

6. FUTURE WORK

As noted in section 4.2 Weaknesses, we encountered certain areas that need extensive redesign in order to achieve our goal of a highly scalable application. Most importantly, we have begun to redesign the visualization engine in conjunction with Dr. Munzner and James Slack. We plan to generate the QuadTree object on the fly as opposed to pre-generating. This will result in a data structure that takes advantage of

[7] http://www.compuphase.com/animtips.htm

the sparseness of our grid as well as the delay between the data-mining engine and the client.

Our current animation algorithms linearly interpolate the intermediate positions based on a set number of changes. In the future, it may be beneficial to alter animation sequences based on the number of changes, number of perceptual changes and the distance of the change. Additionally, slow-in/slow-out animations may aid the user in maintaining context.

The current layout of sets on the grid is analogous to the way words are laid out on a page. This setup results in no correlation between rows. Research will have to be done on whether there are any generic layouts that are more useful. Perhaps layouts will need to be designed by field experts.

Our current set selection method for groups does not allow a hybrid of selection algorithms. For example, it is not possible to filter with an SQL type query and a set contents query. The computation behind the queries are simple, however, an intuitive widget will need to be designed.

We currently supply an initial set of colours to be used for the grid. However, we do not suggest any colours for the marking groups and it is up to the user to choose useful colours. A complete colour pass will need to be executed.

7. CONCLUSION

PowersetViewer was developed as a tool to visualize power sets. It can be applied to many fields and scenarios. Most interesting is the field of data-mining. We tailored the tool to analyze transaction logs as a method to analyze and detect trends.

Several visualization techniques were employed to emphasize precognitive abilities and reduce cognitive overhead. In particular, we used a focus+context technique to navigate and zoom around the dataset. Additionally, we utilized animation to help maintain position and context when navigating.

We believe that with continued development, this tool will be able to visualize and aid in other fields such as AI-searching by displaying and allowing human intervention.

8. REFERENCES

[1] T. Munzner, F. Guimbretiere, S. Tasiran, Li. Zhang, and Y. Zhou. TreeJuxtaposer: Scalable Tree Comparison using Focus+Context with Guaranteed Visibility. *SIGGRAPH 2003*.

[2] J. Slack, K. Hildebrand, T. Munzner, and K. St. John. Fluid Navigation For Large-Scale Sequence Comparison.

[3] J. Han, M. Kamber, and A. K. H. Tung. Spacial Clustering Methods in Data Mining: A Survey. *H. Miller and J.Han (eds.)*, Atlanta , GA, Nov. 2001.

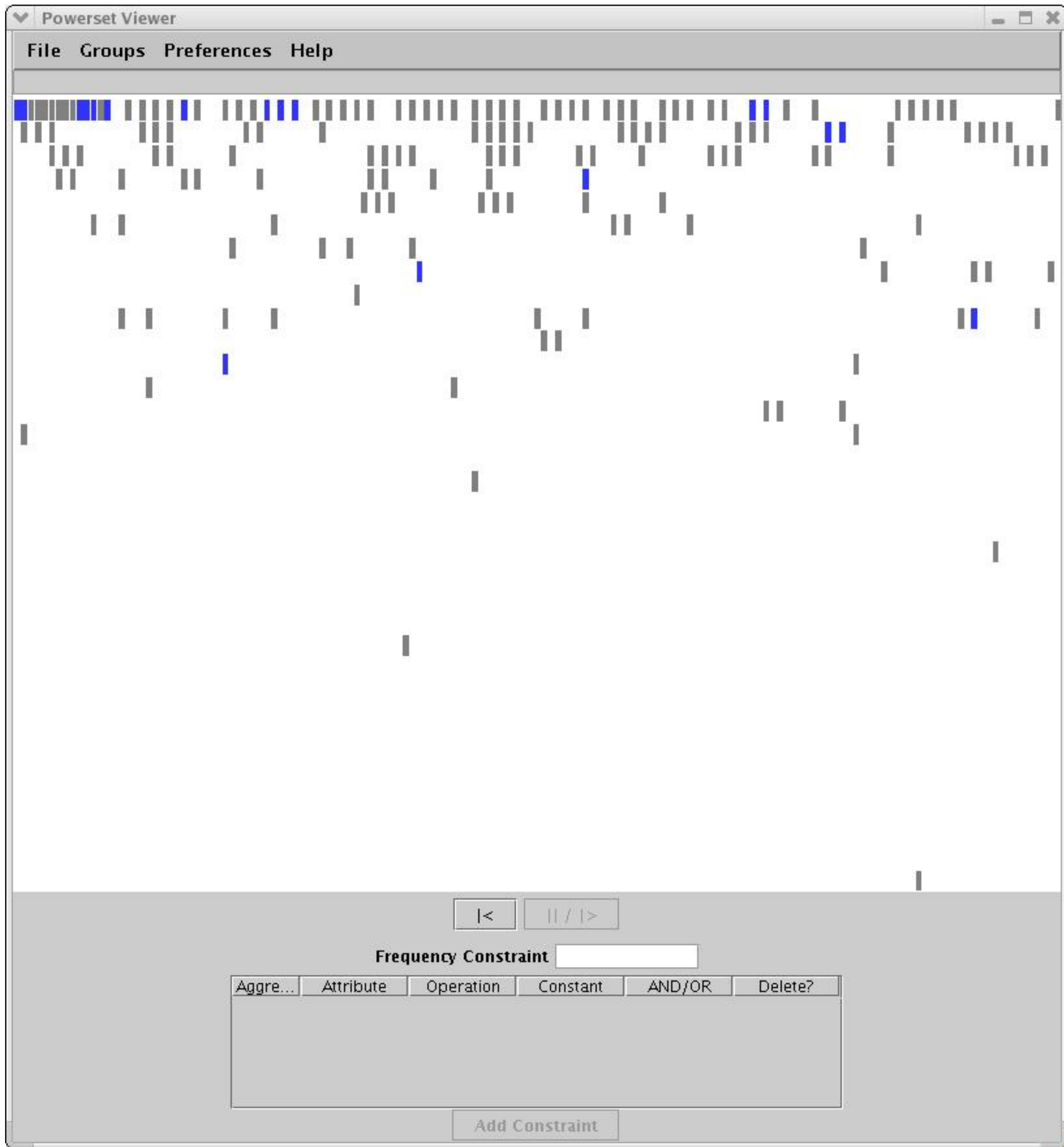

Figure 5: Results after constraint selection in Figure 3

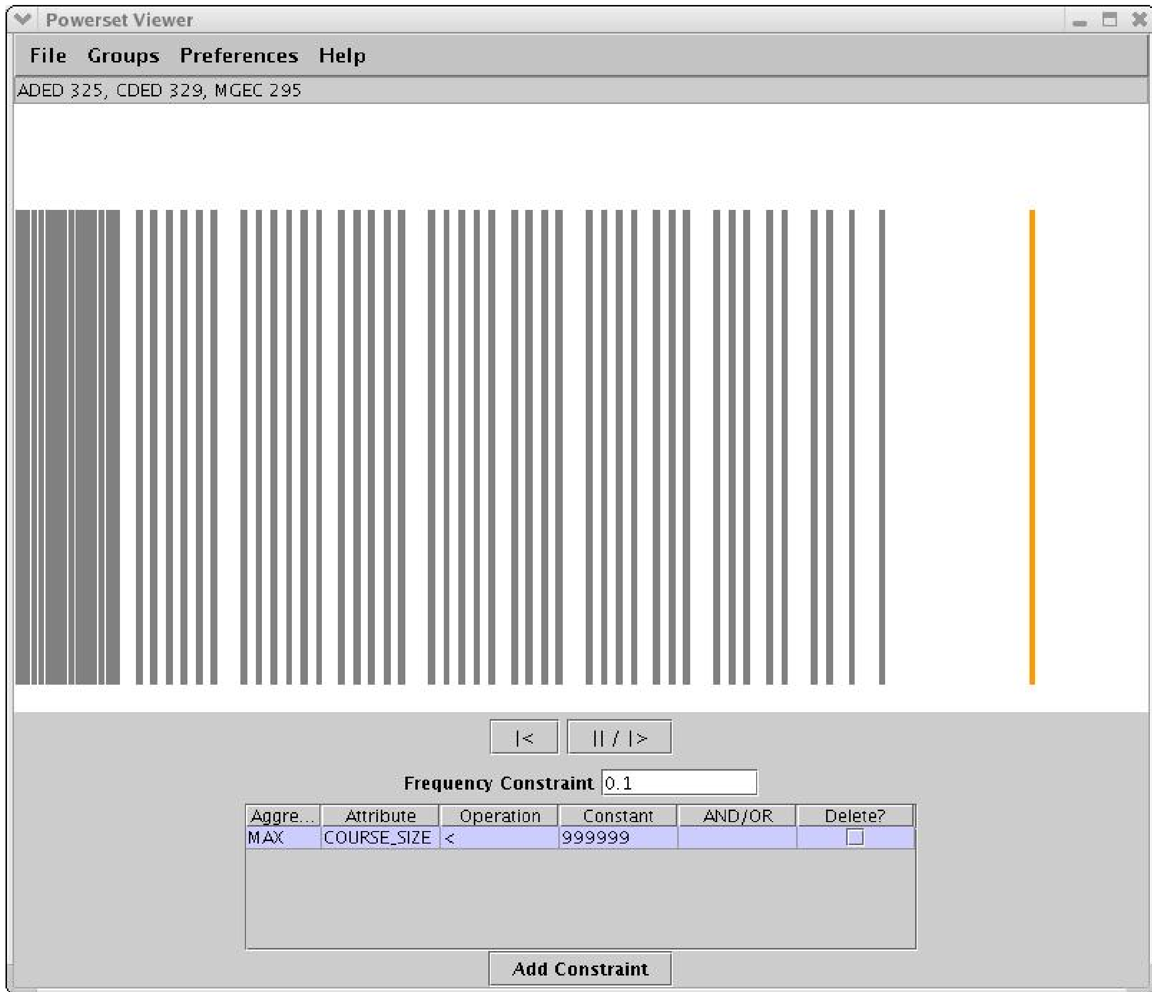

Figure 6: Results after Scenario 1.

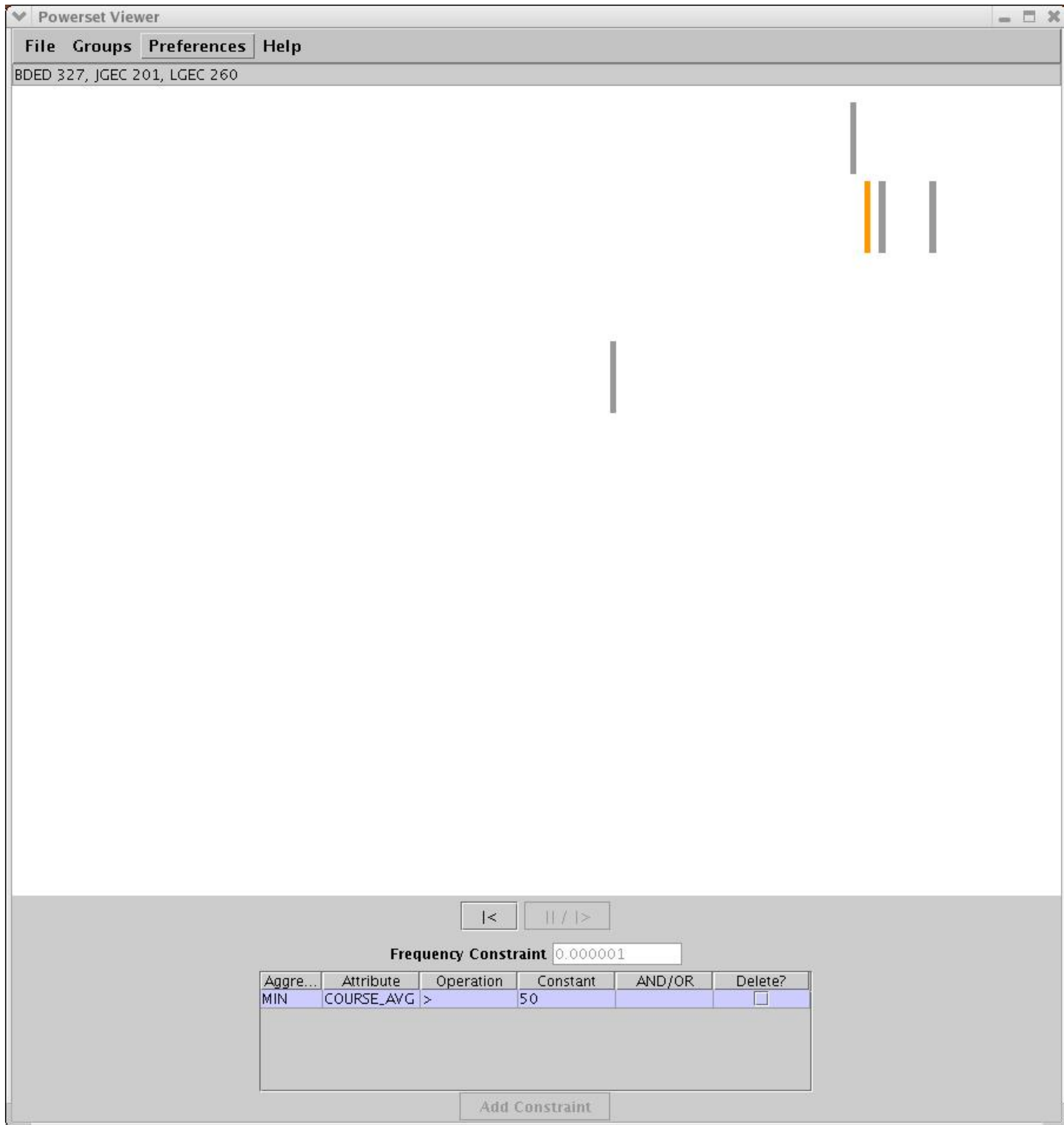

Figure 7: Results after Scenario 2.