Compression for Power Set Visualization using Accordion Drawing

Janek Klawe
61496980
CPSC 449

May 1, 2004

Abstract

Steerable data mining systems allow humans to monitor data mining programs and focus their efforts more effectively, but in order to do so, the human must process the large amounts of data produced by the system. The PowerSetViewer visualization system is an interface for such a system, based on ”accordion drawer” technology; however, one of the challenges in the development of PowerSetViewer was that the data returned by the mining system was in the form of power sets, which are difficult to render in real time because of their large size. My honors thesis project has been to help design and implement a compression algorithm which acts on the data structures used to represent these power sets and reduces them to a manageable size.

1 Introduction

Data mining tools, which search large bodies of data for patterns, see use in a variety of applications. However, such tools generally take a long time to return results; there is a danger that the user will run a data mining program and wait for it to finish, only to find that the system's initial settings were poorly specified. Typically the constraints are either much too tight, resulting in too few matches; or much too loose, resulting in unmanageably large numbers of matches. To remedy this, some researchers [1] suggest that data mining be approached as an iterative process, with a human monitoring the system and ”steering” it towards more favorable areas of analysis. Unfortunately, data mining applications tend to return so much data that it is difficult for a human to process the data quickly enough to steer the application in real time. However, with the aid of a good visualization system, such an arrangement becomes more realistic.

To solve this problem, Professors Munzner and Ng are developing PowerSetViewer, a visualization system based on ”accordion drawing”. PowerSetViewer uses existing accordion drawing code from previous projects, which is based on a quadtree data structure. Unfortunately, due to the size of power sets, which grow exponentially, rendering them with this code would result in very large quadtrees; the costs in memory and processing power would be unacceptable. However, in this application, the quadtrees are extremely deep and sparse, making them particularly amenable to compression. For my honors thesis, I worked with Dr. Munzner to design a compression scheme which exploits the sparseness of these quadtrees, and then implemented it.

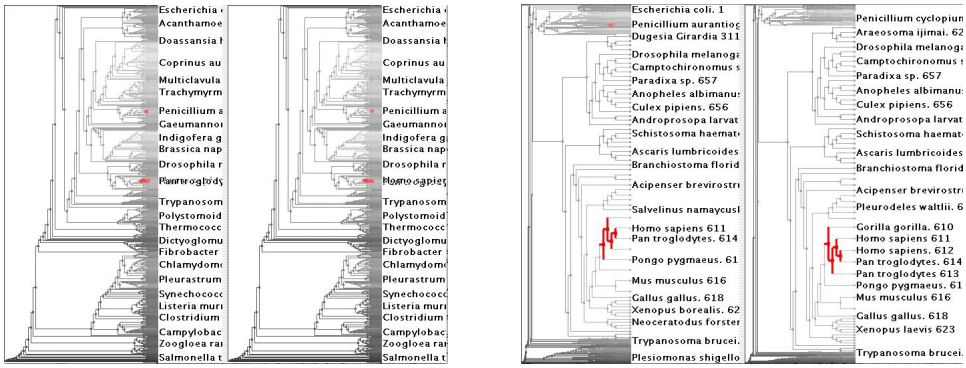

Figure 1: *Left:* TreeJuxtaposer used to compare two phylogenetic trees. Red coloring marks the areas where the trees differ. *Right:* The same tree, after using distortion-based navigation to zoom in on *homo sapiens*.

2 Background

2.1 Data Mining Background

In [2] and [1], Leung *et al.*describe a type of data mining system. The system, given a set of constraints and a database (a set of elements), acts as a filter, returning the subset of elements in the database that satisfy the constraints. The system returns the elements as it finds them, so it produces a stream of elements in real time until it has found all of them. The system is also steerable in real time, meaning the user can change the constraints while the system is running. This feature enables users to notice when the system is returning too many or too few results and adjust the constraints accordingly.

In this project, we consider the case in which the elements in the database are item sets. For example, if the database contained the record of all sales made by a retail store, each element would contain the items purchased in a single transaction, so the item sets might be ”apple, banana” or ”banana, orange, pear”. (Note that the quantity of items purchased is irrelevant; the area of interest is which items were purchased together.) Examples of constraints would be ”item sets with total price $> \$5$” or ”item sets with maximum item price $< \$4$”.

We call the set of all items the alphabet. In the above example, the aphabet would include ”apple”, ”banana”, ”orange”, and ”pear”. Obviously, the set of all possible item sets is the power set of the alphabet. The output of the data mining system will be a subset of this power set.

Since the system can be steered in real time, it is necessary that the user be able to view and understand the returned information efficiently. Because the output will be produced quickly and in great volume, the user will require a special interface to process it. Accordion drawing seems a natural fit.

2.2 Accordion Drawing

”Accordion drawing” refers to a new information visualization technique, in which the user navigates through data by distorting its visual representation, zooming in on interesting areas while keeping the rest of the data visible in the periphery. It was introduced in TreeJuxtaposer [3], a visualization system designed to facilitate the comparison of phylogenetic trees (see Figure 1). It was recently used in a second system, SequenceJuxtaposer [4], which displays molecular sequences rather than trees; enhanced versions of TreeJuxtaposer have also been developed [5].

The accordion drawing technology used in TreeJuxtaposer and SequenceJuxtaposer has several attractive

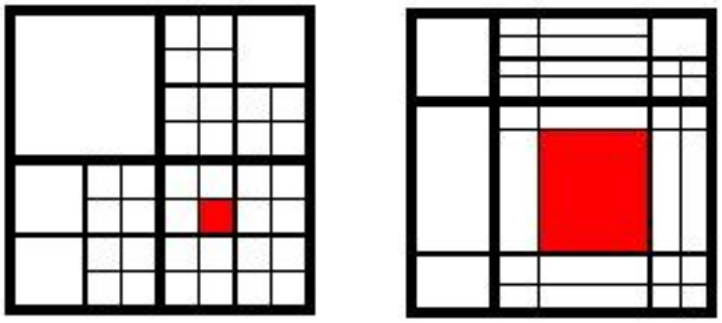

Figure 2: *Left:* A screen divided into a quadtree structure. The user is interested in the red area. Since all nodes have default SplitLines values of $\frac{1}{2}$, each node is allocated an equal amount of screen space. *Right:* By altering the SplitLines values, the screen is distorted, enlarging the area of interest.

features: *distortion-based navigation*, as described above; *guaranteed visibility*, meaning that areas marked as ”important” are always visible, even when they would normally be removed from view by distortion (*e.g.*, zoomed out to less than one pixel in size); and *guaranteed frame rate*, meaning that if the system is unable to render an entire frame within the specified time, it renders as much as possible, starting with the most ”interesting” features, and fills in the rest over time.

Accordion drawing is implemented using a quadtree data structure. The screen is divided into four quadrants, each of which is recursively subdivided into four more quadrants, and so on. The result is a tree with a branching factor of 4. Each element that needs to be rendered is assigned a leaf node at the bottom of the tree; the rest of the nodes are only used to maintain the hierarchy. This hierarchical structure is the main advantage of the quadtree: it makes it easy select to groups of nodes onscreen and scale them.

Specifically, each node in the quadtree has a certain rectangular region of screen space associated with it. This space is divided into four smaller rectangles by two lines: one vertical, one horizontal. The placement of these lines is determined by the node’s so-called *SplitLines values*. By default, these values are both equal to $\frac{1}{2}$, meaning that the rectangle is evenly split on each axis. However, these SplitLines values can be altered (by the user, in real time), resulting in some regions growing and others shrinking accordingly. Through a mouse-based interface, users can manipulate these values in order to ”zoom in” on areas of interest (See Figure 2).

3 PowerSetViewer

PowerSetViewer is an in-progress visualization system designed for viewing item sets returned by the data mining engine described above. It uses the same accordion drawing code as TreeJuxtaposer and SequenceJuxtaposer; however, instead of drawing trees or sequences, it draws item sets. It works as follows:

Upon receiving a set of constraints from the user, the interface communicates the constraints to the data mining engine, which searches the database and returns all item sets satisfying the constraints. As the engine finds them, the item sets are displayed onscreen, where the user can navigate through them using the accordion drawing interface. If user feels, based on the results so far, that the constraints are too tight or too loose, he can change them: all item sets which do not match the new constraints will be removed, and the data mining engine will restart the search using the new constraints.

Unlike trees or sequences, item sets have no obvious structure which we can use to organize them onscreen. Our approach is to first define a simple, abitrary ordering on the power set; in this case, we order the sets first by the number of elements, then by their contents. Then, based on its place in the order, each possible

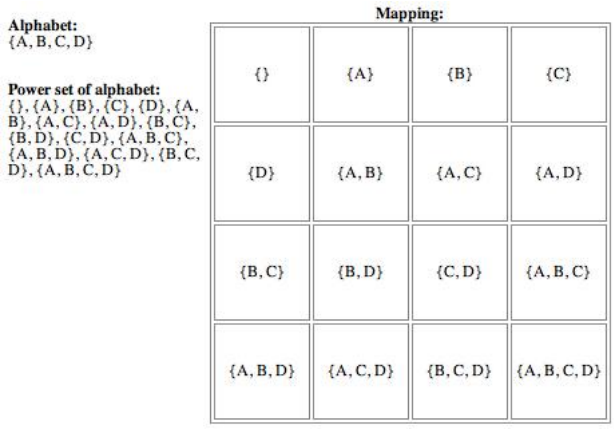

Figure 3: A simple example of positioning item sets onscreen.

item set is mapped to a location in a rectangular region displayed onscreen. (Of course, the number of item sets returned may be larger than the number of pixels onscreen; hence the need for distortion-based navigation.) Figure 3 illustrates how the power set of a simple alphabet is mapped to a two-dimensional grid.

The three students working on PowerSetViewer are Jordan Lee, Dragana Radulovic, and myself. Radulovic's work concerned the data mining engine and its interaction with the visualization client: she extended the server (the data mining engine) for two-way communication, allowing commands (such as changes in constraints) to be sent from a client. Meanwhile, Lee developed the client: extending the accordion drawing system to actually draw the item sets, and adding controls to the GUI to allow users to change the constraints and send them to the data mining server.

My role was the development of the compression system. Dr. Munzner and I designed the algorithm; then I implemented the algorithm to work with the TreeJuxtaposer code. (The PowerSetViewer rendering code had not been written at that time.)

4 Compression

One of the biggest differences between PowerSetViewer and the previous accordion drawing systems is the size of data to be displayed. Because the size of a power set grows exponentially with the size of the original set (the alphabet), the number of possible item sets will be huge for any interesting application. Since the depth of a tree increases with the log of the number of leaves (that is, the number of possible item sets), the depth of our quadtree will be proportional to the size of the alphabet. Thus, even when the power set is very sparsely populated, the tree will take up a great deal of memory and require a great deal of time to navigate.

To solve this problem, Dr. Munzner and I developed compression system which reduces the number of nodes in the tree. We noticed that since the ratio of the quadtree's depth to the number of leaves is relatively high, there will be many nodes in the tree with exactly one child. Furthermore, these nodes contain no actual data; their only purpose is to maintain the hierarchical structure of the tree. Therefore, the only interesting information possessed by this node is which of 4 possible positions its child is in; that is, 2 bits' worth of information. Thus, when a tree contains a long chain of N nodes with no branching, the chain is compressed into an array of $2N$ bits. (Actually, it is stored in two arrays, each containing N bits. When a node is compressed, one bit is used to store its child's vertical position (top or bottom), and one bit is used to store its child's horizontal position (left or right). These two bits are stored in separate arrays). With this innovation, we hope to make it possible to view relatively large sets of item sets while

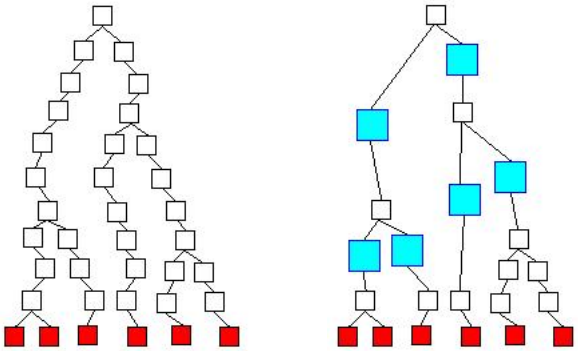

Figure 4: *Left:* A typical tree with little branching. Red nodes contain actual data. *Right:* The same tree, after compression. Blue nodes represent chains of nodes which have been compressed. (Remember that neither leaf nodes nor their parents can be compressed.)

maintaining reasonable response times. (Figure 4 shows how a tree with moderate depth and few leaves can be substantially compressed using this algorithm.)

Unfortunately, there are limits to the number of nodes we can compress. Of course, nodes with multiple children cannot be compressed, and neither can leaf nodes. Furthermore, the parents of the leaf nodes cannot be compressed either, for the following reason:

When the user attempts to distort an area onscreen, the system performs the distortion by altering the SplitLines values of certain nodes. However, if these nodes have been compressed, their SplitLines values will be fixed at $\frac{1}{2}$, making it impossible for the system to perform the correct distortion. Fortunately, for some applications (including, we believe, PowerSetViewer), we do not need every SplitLines value to be available: we can make do with a certain subset. In the case of PowerSetViewer, we require that the parents of the leaf nodes be uncompressed so that their SplitLines values remain mutable.

The compression algorithm I implemented is static, meaning once it has been applied to a quadtree, no nodes may be added or removed from the tree. (Static compression is fine for TreeJuxtaposer, where the phylogenetic trees never change, but for PowerSetViewer a dynamic algorithm will be needed.) It works by traversing the tree, identifying nodes which can be compressed, and replacing each one with a dummy node containing two empty bit vectors. If one dummy node is the parent of another, they are collapsed into a single dummy node, and a bit is added to each bit vector, depending on what position the child was in (*e.g.*, if the child was in the bottom-right position, both bits would be 1).

4.1 Benefits

Although we cannot yet test the compression system empirically, we can estimate the savings in space and time created by this system. We will consider the space savings first.

Let K be the size of the alphabet and N be the number of item sets returned. Then the number of possible item sets is 2^K, and the depth of the quadtree is $\log_4(2^K)$, or $\frac{K}{2}$. Now, since each branch of the tree increases the number of leaves by at least one, the number of branches must be at most $N-1$. The only nodes which cannot be compressed are the nodes with branches, the leaf nodes (the ones with actual content), and the parents of the leaf nodes (because they have important SplitLines information). So, we have at most $N+(N-1)+N$, or $3N-1$ nodes which cannot be compressed. In other words, the number of nodes which cannot be compressed is $O(N)$. Obviously, we cannot expect to have space complexity better than $O(N)$, so being able to compress all but $3N$ nodes into 2 bits each (plus overhead costs for the bit vector data structures and dummy cells) constitutes excellent savings in space.

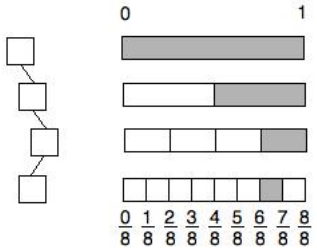

Figure 5: Each node in the tree fragment on the left corresponds to an interval on the right: the top node corresponds to the entire interval $[0, 1]$, its child corresponds to just the right half of the interval ($[\frac{1}{2}, 1]$), and so on. This fragment would be compressed with a bit vector of 110, or "right, right, left". Observe that the bottom node starts at the 6th increment, or position 110 in binary. In general, given a series of n compressed nodes, with the bit vector represented as a binary number b, the corresponding starts at $\frac{b}{2^n}$ and ends at $\frac{b+1}{2^n}$. Thus, the bounding rectangle of the bottom node can be determined by performing just 2 divisions.

Savings in time are more difficult to estimate. However, we can examine the most common operation, which is calculating the onscreen location of each item set. As described in Section 2.2, each node in the quadtree represents a rectangle of screen space; call this the node's *bounding rectangle*. The location and dimensions of this rectangle depend on those of its parent's bounding rectangle, as well as the parent's SplitLines values. In order to draw an item set, it is necessary to know the bounding rectangle of its corresponding leaf node. Thus, in order to draw every item set, the bounding rectangle of every node in the tree must be calculated.

But what happens when nodes are compressed? Since a node's bounding rectangle depends on that of its parent, we only need to calculate the bounding rectangle of the bottom node in each chain. Now, when a chain of nodes is compressed, it is assumed that the SplitLines values of each node are all equal to $\frac{1}{2}$ (the initial value for all nodes). Because of this fact, it turns out that the bounding rectangle of the bottom node can be found by interpreting the bit vectors representing the chain as two binary numbers, and performing pair of divisions on each number (See Figure 5 for an example).

Unfortunately, since it is impossible to operate on arbitrarily large binary numbers as primitives, this operation cannot quite be done in constant time. In practice, it requires a few operations for every 63 bits in each bit vector. (Because the largest primitive in Java is 64 bits, and it is easier to leave aside 1 bit to avoid dealing with the sign bit, our bit vector stores bits in groups of 63.) In other words, it can be performed in very fast linear time.

Returning to the original issue, that of calculating the location of each item set, it is clear that the time savings can be substantial if a large number of nodes are compressed. And, of course, we have already established that the number of uncompressed nodes must be less than $3N$, meaning that we can expect significant savings in time as well as space.

4.2 Future Work and Potential Problems

My implementation of the compression algorithm demonstrated the idea behind the system, but some work remains to be done before it can be used in PowerSetViewer. First, the system needs to be modified to allow elements to be added dynamically. TreeJuxtaposer, the system on which I was working, reads in all its data at once and constructs its quadtree during initialization. Consequently, it does not allow new elements to be added while it is running. However, because new data will be continously provided by the data mining engine, it is necessary that the PowerSetViewer interface be able to add new elements at any time. It will also no longer be possible to perform a one-time compression on the quadtree during initialization; instead, the compression must be performed continuously as elements are added. Additionally, there will occasionally be large numbers of deletions: when the user changes the constraints, many item sets will no

longer satisfy them and will need to be removed. My implementation does not take deletions into account at all.

The compression algorithm also needs to be integrated into the PowerSetViewer system, which may cause problems. One potential problem is the time needed to compress each node. The time taken by the current implementation seems reasonable (considering that it is not optimized), but the dynamic version of the algorithm may be faster or slower. Also, the cost of compression is proportional to the rate at which nodes are added to the tree, which depends on the speed of the data mining algorithm. Consequently, it is difficult to predict how much overhead will be imposed by this algorithm.

Another problem has to do with SplitLines. As explained in Section 4, certain nodes cannot be compressed because their SplitLines values are important. If these cells were compressed, the system would be unable to perform distortions properly. However, exactly which nodes fall into this category depends on the application. For example, in TreeJuxtaposer, every node's SplitLine values are used in some way, so the compression causes some minor inaccuracies. We believe that in PowerSetViewer, only the parents of leaf nodes will contain important SplitLine information; however, if this assumption turns out to be mistaken, we may not be able to compress as many nodes, and the savings estimates in Section 4.1 will not hold.

5 Conclusions

For my honors thesis project, I worked with Dr. Munzner to design an algorithm to compress deep, sparse quadtrees in PowerSetViewer. I then implemented the algorithm in the TreeJuxtaposer system. The algorithm appears to work well in TreeJuxtaposer; furthermore, based on the calculations of Section 4.1, we expect this algorithm to provide enough savings in space and time to allow PowerSetViewer to render the results from the data mining engine in real time.

6 Acknowledgements

I would like to thank Dr. Tamara Munzner, my thesis supervisor; Dr. Raymond Ng and Dr. Carson Kai-Sang Leung, who developed the data mining engine; and Dragana Radulovic and Jordan Lee, who worked with me on the PowerSetViewer project.

References

[1] Lakshmanan, L.; Leung, C.; Ng, R. "Efficient Dynamic Mining of Constrained Frequent Sets". *ACM SIGKDD Explorations Newsletter* Volume 4, Issue 1, pages 40-49 (June 2002).

[2] Leung, C.; Lakshmanan, L.; Ng, R. "Exploiting Succinct Constraints using FP-trees". *ACM Transactions on Database Systems* Volume 28, Issue 4, pages 337-389 (December 2003).

[3] Munzner, T.; Guimbretière, F.; Tasiran, S.; Zhang, L.; Zhou, Y. "TreeJuxtaposer: Scalable Tree Comparison using Focus+Context with Guaranteed Visibility". *SIGGRAPH 2003*.

[4] Slack, J.; Hildebrand, K.; Munzner, T.; St. John, K. "SequenceJuxtaposer: Fluid Navigation For Large-Scale Sequence Comparison In Context". (in publication)

[5] Beermann, D.; Munzner, T.; Humphreys, G. "Scalable, Robust visualization of Large Trees". (in publication)