CHAPTER TWELVE

Turning a Table into a Tree: Growing Parallel Sets into a Purposeful Project

Robert Kosara

ACADEMIC SOFTWARE PROJECTS TEND TO GROW ORGANICALLY from an initial idea into something complex and unwieldy that is novel enough to publish a paper about. Features often get added at the last minute so they can be included in the paper, without much thought about how to integrate them well or how to adapt the program's underlying architecture to make them fit.

The result is that many of these programs are hacked together, buggy, and frankly embarrassing. Consequently, they do not get released together with the paper, which leads to a fundamental problem in visualization: reproducibility is possible in theory, but in practice rarely happens. Many programs and new techniques are also built from scratch rather than based on existing ones.

The optimal model would be to release the software right away, then come back to it later to refine and rearchitect it so that it reflects the overall design goals of the project. This is seldom done, though, because there is no academic value in a reimplementation (or thorough refactoring). Instead, people move on to the next project.

The original prototype implementation of Parallel Sets (*http://eagereyes.org/parallel-sets*) was no different, but we decided that in order to get the idea out of academia into actual use, we would need a working program. So we set out to rethink and redesign it, based on a better understanding of the necessary internal structures that we had gained over time. In the process, we not only re-engineered the program, but also revised its generated visualization to clarify its underlying idea.

Categorical Data

Hundreds of visualization techniques are described in the literature (with more added every year), but only a few specifically work with *categorical data*. Such data consists of only a few values that have special meanings (as opposed to continuous numerical data, where the numbers stand for themselves). Examples include typical census data, like values for sex (male or female), ethnicity, type of building, heating fuel used, etc. In fact, categorical data is crucial for many real-world analysis tasks. The data we originally designed our technique for was a massive customer survey consisting of 99 multiple-choice questions with almost 100,000 respondents. People were asked questions about consumer goods, like detergents and other household items, as well as demographic questions about household income, number of kids, ages of kids, etc. Even in cases where it would have been possible to gather precise information (like age), the survey combined the values into groups that would be useful for later analysis. That made all the dimensions strictly categorical, and almost impossible to visualize using traditional means.

The dataset we will use to illustrate Parallel Sets in this chapter describes the people on board the *Titanic*. As shown in Table 12-1, we know each passenger's travel class (first, second, or third passenger class, or crew), sex, age (adult or child), and whether they survived or not.

Table 12-1. *The Titanic dataset*

Dimension	Values
Class	First, Second, Third, Crew
Sex	Female, Male
Age	Child, Adult
Survived	Yes, No

There are really only three visualization techniques that work particularly well for categorical data: treemaps (Shneiderman 2001), mosaic plots (Theus 2002), and Parallel Sets. The reason for this is that there is a mismatch between the discrete domain of the data and the continuous domain of most visual variables (position, length, etc.). Treating categorical data as if it were numerical is acceptable when all but a few dimensions are continuous, but becomes entirely useless when all of them are categorical (Figure 12-1). While the natural distribution of data in most numerical datasets makes it possible to glean the rough distribution of at least the number of values, this becomes entirely impossible when there are only a few different values that are exactly the same between data points.

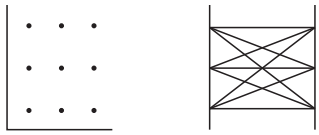

Figure 12-1. *Using classical visualization techniques for categorical data: Scatterplot (left) and Parallel Coordinates (right) lead to massive overplotting and do not provide much information even when tricks (such as jittering the data points) are used*

Parallel Sets

Parallel Sets, or ParSets (Bendix 2005, Kosara 2006), is a visualization technique that was designed specifically for interpreting categorical data. When talking to the experts analyzing the customer survey data, we realized that most of the questions they were asking were not based on individual survey responses, but on classes of answers, or sets and set intersections. How many people with more than three children under five years of age buy brand-name detergent? Or, put differently, how many members of Set A are also in Set B? How many first-class passengers on the *Titanic* survived (i.e., how many were in category *first class* on the *class* dimension, and in the *yes* category on *survived*)? How many of them were women (i.e., how many also had the value *female* in the *sex* dimension)?

This approach means that instead of plotting thousands of individual points, we only need to show the possible sets and subsets that exist in the data, as well as their sizes. If the numbers and relative sizes of those sets stayed the same, we reasoned, we could even show that the technique was independent of the actual dataset size.

In addition to the idea of showing the data as sets, ParSets was heavily influenced by Parallel Coordinates (Inselberg 2009), a popular visualization technique for high-dimensional numerical data. The parallel layout of axes makes them easier to read and compare than the nested structures of treemaps and mosaic plots, especially as the number of dimensions increases. It is also easier to design effective interactions for this kind of layout.

The first version of Parallel Sets (see Figure 12-2) was based on the categories first, then on the intersections. For each axis, we showed each category as a box, with its size corresponding to the fraction of all the data points that each category represented. In terms of statistics, this is called the *marginal distribution* (or marginal probability). Each axis is essentially a bar chart, with the bars tipped over rather than standing next to each other.

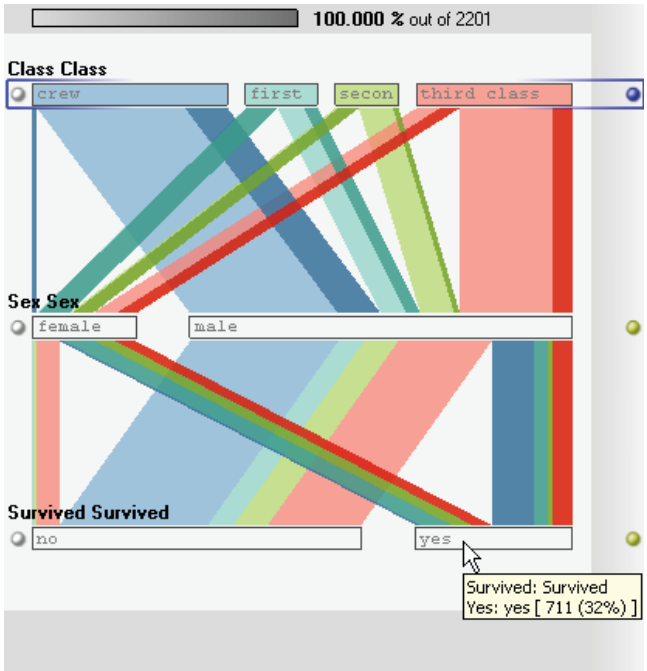

Figure 12-2. *The original Parallel Sets design*

Reading just the bars in Figure 12-2, it is easy to see that the crew was the largest class of people on the *Titanic*, with the third class close behind. The first class was much smaller than the third class, but was actually larger than the second class. It is also quite obvious that there was a majority of men (almost 80%) on the ship, and that only roughly one-third of all people on board survived.

Ribbons connect categories that occur together, showing how often, for example, *first class* and *female* intersect, thus making it possible to tell what proportion of the passengers in first class were women. The ribbons are what makes Parallel Sets more than a bunch of bar charts: being able to see distributions on several axes at the same time allows the user to identify and compare patterns that would otherwise be difficult to spot.

In the case of the *Titanic*, there was clearly an uneven distribution of women among the different classes. While the first class was close to 50% female, the second and third classes had progressively larger majorities of men. The crew consisted of over 95% men.

While the ribbons are clearly useful, they also pose some challenges. They must be sorted and the wider ones drawn first, so that the smaller ones end up on top and are not hidden. Also, when there are many categories there tend to be a lot of ribbons, resulting in a very busy display that is difficult to read and interact with.

Interaction is an important aspect of ParSets. The user can mouse over the display to see actual numbers, and can reorder categories and dimensions and add dimensions to

(and remove them from) the display. There are also means of sorting categories on an axis by their size, as well as combining categories into larger ones (e.g., to add up all the passenger classes to better compare them with the crew).

Visual Redesign

One aspect of ParSets that required us to experiment quite a bit was the question of how to order the ribbons going from one axis to the next. We came up with two different orderings that seemed to make good sense, which we called *standard* and *bundled*. Standard mode ordered ribbons only by the category on top, which led to a branching structure but resulted in a rather visually busy display when large numbers of dimensions and categories were included. Bundled mode kept ribbons as parallel as possible by grouping them by both the top and bottom categories, which meant detaching parts of the ribbons from one another vertically.

It was only when we started to reimplement the technique a while later and were looking for a good representation of the visual structures that we realized that we had been looking at a tree structure all along (and that standard mode was the way to go). The entire set of data points is the root node of the tree, and each axis subdivides it into the categories on that axis (Figure 12-3). The ribbons display the tree; the nodes just look different than expected because we collect them on each axis to form the bars.

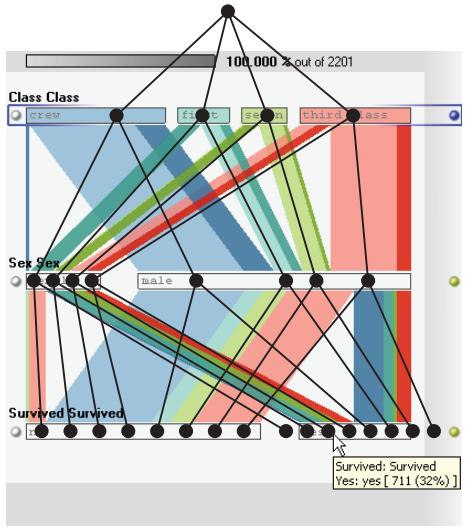

Figure 12-3. *The tree structure in Parallel Sets: nodes on each level are collected into bars, and the ribbons are the connections between the nodes*

We went ahead with our reimplementation without making any major changes to the visual display, but the idea of the tree stuck in my head. So one day, I asked myself: what if we reduced the bars and focused on the ribbons? And lo and behold, I was looking at a much clearer tree structure (Figure 12-4).

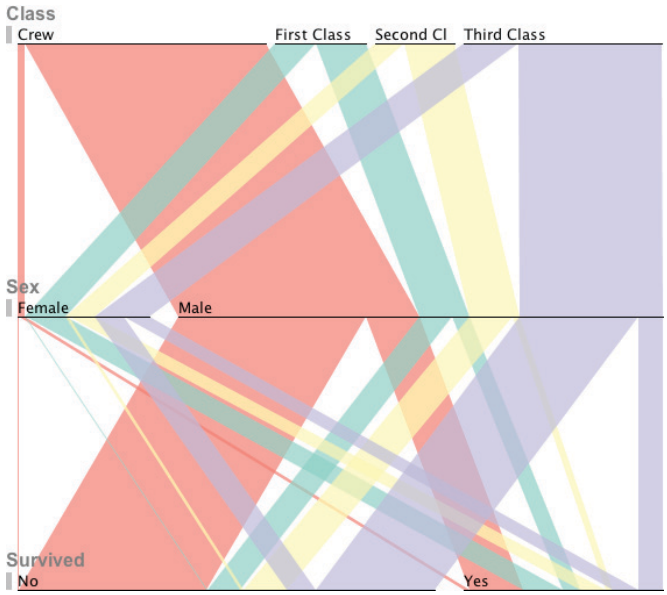

Figure 12-4. *The new Parallel Sets design, showing the tree structure much more clearly*

A simple change had shifted the focus from the category boxes to the ribbon tree. In the new design, the boxes still appear when the user mouses over the lines (to suggest to the user that she can interact with them), but they are only a means to an end. The key information we are interested in is really the decomposition into subsets.

In addition to improved structural clarity, the new design also makes much better use of typography to communicate the hierarchy of dimension and category labels and is much more pleasant to look at.

Looking at data in terms of aggregation and sets is not a new idea. Polaris (Stolte, Tang, and Hanrahan 2002) and, by extension, Tableau* were built on a similar idea: aggregation of individual values and decomposition into subsets. The use of treemaps for non-hierarchical data (which is what treemaps are mostly used for today) is based on the same transformation. Creating a tree of subsets from the data enables one to use any hierarchy visualization to show that data. The treemap, with its emphasis on node size rather than tree structure, is a natural choice for this.

The initial design change required only a few small changes in the program, but it was clear from this point on (and from the rather lackluster performance of our reimplementation) that the perceived need for a visual change had just been a symptom of a fundamental design issue with the program's data model.

* See *http://www.tableausoftware.com*.

A New Data Model

In the original program, the data had been stored the way it came in: as one big table. We later added the ability to create additional dimensions from the data, but the principle did not change. With every change to the display, the program had to work its way through the entire dataset and count the combinations of categories. With larger datasets, this became quite slow and required a lot of memory.

The big advantage of looking at data in terms of sets is that the individual data points are really of no interest; what counts are the subsets. So, the natural next step was to look at all possible aggregations of the data into sets, which could then be used to compute any subsets the user was interested in.

In statistics, this is called a *cross-tabulation* or pivot table. In the case of two dimensions, the result is a table with the categories of one dimension becoming the columns, and the other becoming the rows (Figure 12-5).

Class	Sex					
	female		male			
first	145	44.6%	180	55.4%	325	
	30.8%	6.6%	10.4%	8.2%		14.8%
second	106	37.2%	179	62.8%	285	
	22.6%	4.8%	10.4%	8.1%		12.9%
third	196	27.8%	510	72.2%	706	
	41.7%	8.9%	29.5%	23.2%		32.1%
crew	23	2.6%	862	97.4%	885	
	4.9%	1.1%	49.8%	39.1%		40.2%
	470		1731		2201	
		21.4%		78.6%		100%

Figure 12-5. *A cross-tabulation of the class and sex dimensions of the Titanic dataset*

There are two kinds of numbers in this table: counts and percentages. Each cell contains the count of people for its combination of criteria at the top left, and the percentage that number is of the entire dataset at the lower right. That latter percentage is called the *a priori percentage* (or probability). What is generally of more interest, though, are the *conditional percentages* (or probabilities), which tell us the composition of the different classes. In the top-right corner of each cell is the chance of finding the column's criterion given that we know the row (e.g., how many of the passengers in first class were women); at the lower left is the percentage likelihood of finding the row criterion given the column (e.g., what percent of women were in first class).

Because the data is purely categorical, the cross-tabulation contains all the information about it and is all we need to store. If we wanted to recreate the original data from it, we could do that by simply generating as many rows with each combination of categories as are given by the cell. The only case where additional data is needed is when the dataset also contains numerical columns.

A cross-tabulation for more than two dimensions is a bit more involved, but follows basically the same principle. A high-dimensional array is constructed that has as many dimensions as the dataset, with each cell in the array holding the count of how often that combination of values occurred.

Unfortunately, the number of possible combinations gets rather large quite quickly, and is actually much larger than the number of rows in most datasets. In the case of the census data, for example, taking only the dimensions *owned or rented*, *building size*, *building type*, *year built*, *year moved in*, *number of rooms*, *heating fuel*, *property value*, *household/family type*, and *household language* (out of over 100 dimensions) would result in 462,000,000 combinations, while the 1% microdata census sample has only 1,236,883 values for the entire U.S.!

The key here is that in the high-dimensional case, most combinations never actually occur in the data. So, it makes sense to only count those that do and store only their information. This is done in our current implementation by simply using an array of integers to hold all the values for each row, and using that as the key for a hash table. In almost all cases, that hash table takes up less space than the original data.

The Database Model

The database is essentially a direct mapping of the hash table that contains the counts for each combination of categories. Each dataset is stored in a separate table, with a column for each dimension in the dataset. Each row contains the values for the categories that describe the cell in the cross-tabulation, as well as the count of how often that combination occurs. There is an additional field, called the *key*, which is unique for each row and is used for joining the table when looking at numerical data.

Aggregating the data is done with a SQL query that simply selects the dimensions the user is interested in plus the total counts, and groups the results by those same dimensions (Table 12-2):

```
select class, sex, survived, sum(count) from titanic_dims
group by class, sex, survived;
```

The database thus aggregates the counts and returns a lower-dimensional cross-tabulation containing only the values needed for the visualization.

Table 12-2. *The result of querying the Titanic dataset to include only the dimensions class, sex, and survived*

Class	Sex	Survived	Count
First	Male	Yes	62
First	Male	No	118
First	Female	Yes	141
First	Female	No	4
Second	Male	Yes	25
Second	Male	No	154
Second	Female	Yes	93
Second	Female	No	13
Third	Male	Yes	88
Third	Male	No	422
Third	Female	Yes	90
Third	Female	No	106
Crew	Male	Yes	192
Crew	Male	No	670
Crew	Female	Yes	20
Crew	Female	No	3

This model is very similar in principle to data warehousing and Online Analytical Processing (OLAP). Most databases have a special *cube* or *rollup* keyword to create an aggregation from a regular table. This has the advantage that no special processing is needed beforehand, but the disadvantage of being slower and requiring more disk space to store all the original values. Structuring the data specifically for fast read and aggregation performance (as is done in data warehouses and our database schema) considerably speeds up the most common operation at the expense of more processing being required when new data is added.

While the ParSets program does not currently show numerical dimensions, it does store them in the database. They are stored in a separate table, containing the key of the row the values correspond to and one column per numerical dimension. Instead of using the count, a simple join query can therefore be used to aggregate any numerical dimension by the cross-tabulation cells. Any standard SQL aggregation (`sum`, `avg`, `min`, `max`) can be used for this purpose. Eventually, the program will allow the user to select a numerical dimension to use to scale the bars and ribbons, and to also select the aggregation used.

The current version of Parallel Sets stores its data in a local SQLite database. SQLite is a very interesting open source database that operates on a single file. It is used in many embedded applications and is extremely resilient against data corruption (such devices

have to expect power failure at any moment). While it does not have all the features of commercial databases, it is small, fast, and does not require any setup. This makes it a perfect data store that has a query language as an added benefit.

Growing the Tree

The cross-tabulation that the database stores and that can be retrieved is only a part of the story, though. To show the user the Parallel Sets display, we need a tree. Whenever the user changes the dimensions or reorders them, the program queries the database to retrieve the new cross-tabulation. It then walks through the resulting data to build the tree. If you look closely, you can actually already see it in Table 12-2. Whenever the same value appears several times in the same column, we're looking at the same node of the tree, and only the nodes to the right of it change, as shown in Table 12-3.

Table 12-3. *The tree structure inherent in the query result in Table 12-2*

Class	Sex	Survived	Count
First	Male	Yes	62
		No	118
	Female	Yes	141
		No	4
Second	Male	Yes	25
		No	154
	Female	Yes	93
		No	13
Third	Male	Yes	88
		No	422
	Female	Yes	90
		No	106
Crew	Male	Yes	192
		No	670
	Female	Yes	20
		No	3

All the program needs to do is go through the result set line by line and build the tree by following the existing nodes from left to right until it encounters a node that does not exist yet. That node is added and its count is taken from the database row.

The database contains only the counts for the tree's leaves, though, not its internal nodes (other databases, such as Oracle, have queries that also return internal nodes when performing cube queries). However, it is easy enough to calculate those by simply summing the values of each node's child nodes recursively, from the leaves to the root node.

The counts themselves are also only the raw material of the fractions, which are calculated in the same step once all counts for a node are known. To actually display the bars and ribbons, percentages are used: the *a priori* percentage of each category becomes the length of the bar, by using it as the fraction of the total width, and the conditional percentages (the lower category on a ribbon given the upper category) are used to determine the width of the ribbon as a fraction of the category bar length (Figure 12-6).

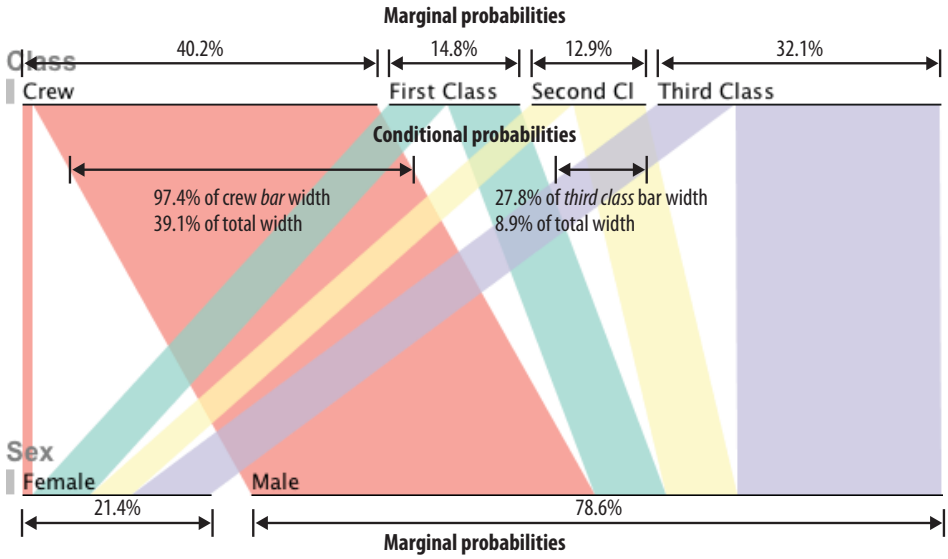

Figure 12-6. *The width of each ribbon represents its marginal probability (proportional fraction) of the total data set, and also its conditional probability within each category*

Parallel Sets in the Real World

Since the program was released in June 2009, it has been downloaded over 750 times (as of January 2010). We have heard from many users who have had success using it with their own data. We even won a prize at VisWeek 2010's Discovery Exhibition (*http://discoveryexhibition.org*) for our entry talking about three case studies using the program. This was written together with Joe Mako (Mako Metrics), Jonathan Miles (Gloucestershire City Council, UK), and Kam Tin Seong (Singapore Management University).

Joe Mako's use of the program was especially interesting, because he used it to show a kind of data flow through many processing stages. Putting the last stage on top meant that the ribbons were colored by final result, which let him easily see where problems occurred. There actually is a visualization technique that is visually (though not conceptually) similar to Parallel Sets that is used for flows, called a Sankey diagram. ParSets can emulate these diagrams for flows that move strictly in one direction and only split up (but never merge). Jonathan Miles and Kam Tin Seong's uses were closer to the original aim of the program, providing interesting insights into survey results and bank customers, respectively.

Conclusion

Academia values novelty, but there is clearly a case to be made for letting ideas develop over time, so they become clearer and more refined. The result is not just a better understanding of the issues and techniques, but better tools that are easier to understand and provide more insights to the user.

Redesigning Parallel Sets illustrated how visual representation and data representation (as well as database design) go hand in hand. Understanding the underlying model of our own technique led to a better visual design, which in turn led to a much-improved database and program model.

References

Bendix, Fabian, Robert Kosara, and Helwig Hauser. 2005. "Parallel Sets: Visual analysis of categorical data." In *Proceedings of the IEEE Symposium on Information Visualization*, 133–140. Los Alamitos, CA: IEEE Press.

Inselberg, Alfred. 2009. *Parallel Coordinates: Visual Multidimensional Geometry and Its Applications*. New York: Springer.

Kosara, Robert, Fabian Bendix, and Helwig Hauser. 2006. "Parallel Sets: Interactive exploration and visual analysis of categorical data." *IEEE Transactions on Visualization and Computer Graphics* 12, no. 4: 558–568.

Shneiderman, Ben, and Martin Wattenberg. 2001. "Ordered treemap layouts." In *Proceedings of the IEEE Symposium on Information Visualization*, 73–78. Los Alamitos, CA: IEEE Press.

Stolte, Chris, Diane Tang, and Pat Hanrahan. 2002. "Polaris: A system for query, analysis, and visualization of multidimensional relational databases." *IEEE Transactions on Visualization and Computer Graphics* 8, no. 1: 52–65.

Theus, Martin. 2002. "Interactive data visualization using Mondrian." *Journal of Statistical Software* 7, no. 11: 1–9. *http://www.theusrus.de/Mondrian/*.