CS346 Advanced Databases – Report

Column Data Organisation

Bartholomew Joyce
ID: u1517739

Department of Computer Science, University of Warwick
email: B.Joyce@warwick.ac.uk

December 5, 2017

1 Introduction

The aim of this project was to design and implement data pre-processing methods that prepare data sets of a particular format before they get compressed as to maximise compressibility of the data sets. The sample data sets provided are binary tables with hundreds of columns and arbitrarily many rows. The compression algorithm takes these data sets and compresses along the columns, compressing runs of 0s or 1s down to a count of their run-length.

The compressibility of individual columns can be maximised very easily by sorting the rows: group all rows of 0s together, followed by all rows of 1s. However, maximing a data set's compressibility overall is more difficult. When the rows of the columns are sorted, they can't be sorted independently. Consequently, prioritising the compression of one column will lead to another column being out of order. The challenge is thus to try and improve the compression of all columns simultaneously.

The approach taken to achieve good compression on all columns was to investigate the structure of the data sets and to intelligently prioritise particular columns. Columns are rarely ever identically and independently distributed. By discovering the relationships that exist between columns, assumptions can be made about the data that will allow for optimal sorting. In particular, all the columns are grouped with their related counterparts, and then the sorting algorithm prioritises the sorting of the larger groups over the smaller groups.

Though in theory the sorting method was very intriguing, the results were unfortunately not very good. In terms of performance, the sorting method took longer than the naive lexicographic sorting method, primarily due to it needing to calculate the relationships between columns. In terms of output, it wasn't able to achieve a compression ratio signicantly better than naive lexicographic ordering.

2 Research and Development

2.1 Finding structure in randomness

In the initial stages of carrying out the coursework the data sets were converted to bitmap images. The reasoning here was that key observations about the data sets can be made by representing the data in a visual manner; a pattern in the data will reveal itself more directly if the pattern manifests itself graphically.

The first pattern discovered was in the file `uniform.txt`. As you can see from figure 1, when sorting `uniform.txt` by the first column it becomes apparent that whenever the first column contains a 1 bit, the following ten columns are always 0.

Naturally, one ordering optimisation became clear: if columns 2-10 are always 0 when a 1 appears in column 1, then those columns won't need to be sorted when there is a 1 in column 1, because they will already be sorted.

In order to generalise this observation code was written to build a profile for a given data set. The profile should quantify the degree to which columns are correlated with one another. In the spirit of visualisations this profile was also represented as bitmaps. In figure 2 you can see the correlation profile of `uniform.txt`, depicting how every column interacts with every other column. From the diagram it is clear that columns 0-9 are mutually exclusive: any column being 1 implies that all other columns will be 0 (for any particular row).

It is clear from `uniform.txt`'s statistical profile that its data is very structured in some places and uniformly distributed in others. What about data from the real-world? In figure 3 you can see the profile of `hep.txt`. Clearly, the data is a lot more unstructured, consisting of high density columns, low density columns, and vague correlations between the remaining columns. Although the data is more unstructured, it still contains many instances of mutually exclusive groups.

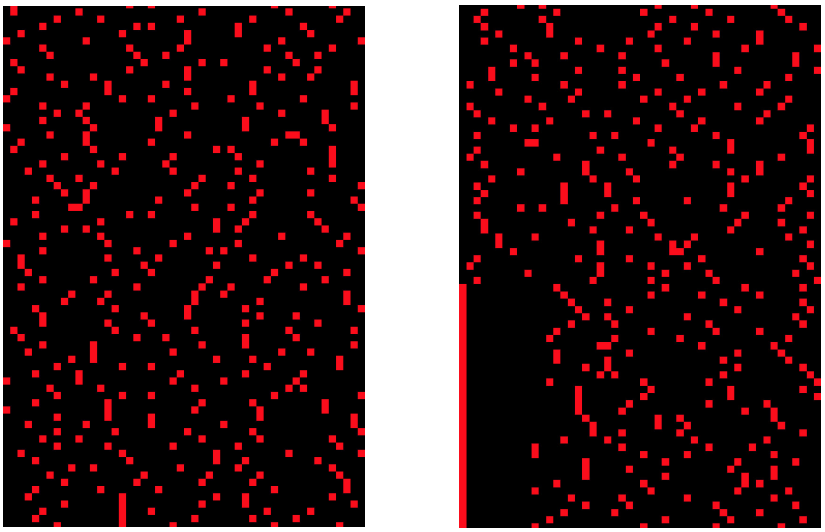

Figure 1: Bitmap images of `uniform.txt`. On the left: an unsorted section. On the right: the result of sorting the first column.

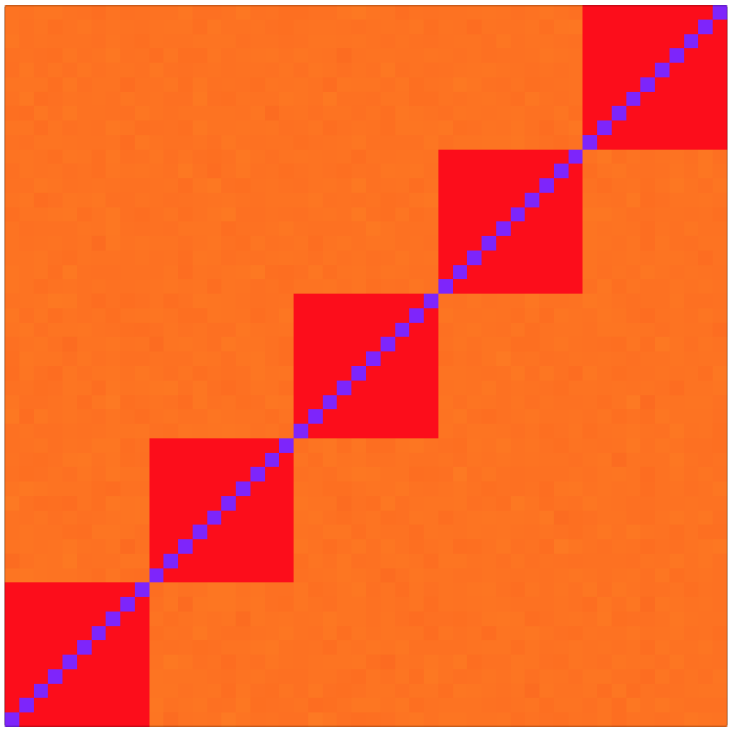

Figure 2: Statistical profile of `uniform.txt`. From left to right, x varies from 0 to 49. From bottom to top, y varies from 0 to 49. Each pixel represents the probability: $P(C_y = 1|C_x = 1)$. A violet colour signifies $P(C_y = 1|C_x = 1) = 1$, i.e. column x being 1 implies that y is 1 on every row. In contrast, the red colour signifies $P(C_y = 1|C_x = 1) = 0$, i.e. column x being 1 implies that y is 0 on every row. The orange colour signifies $P(C_y = 1|C_x = 1) \approx 0.1$, i.e. the columns are independent.

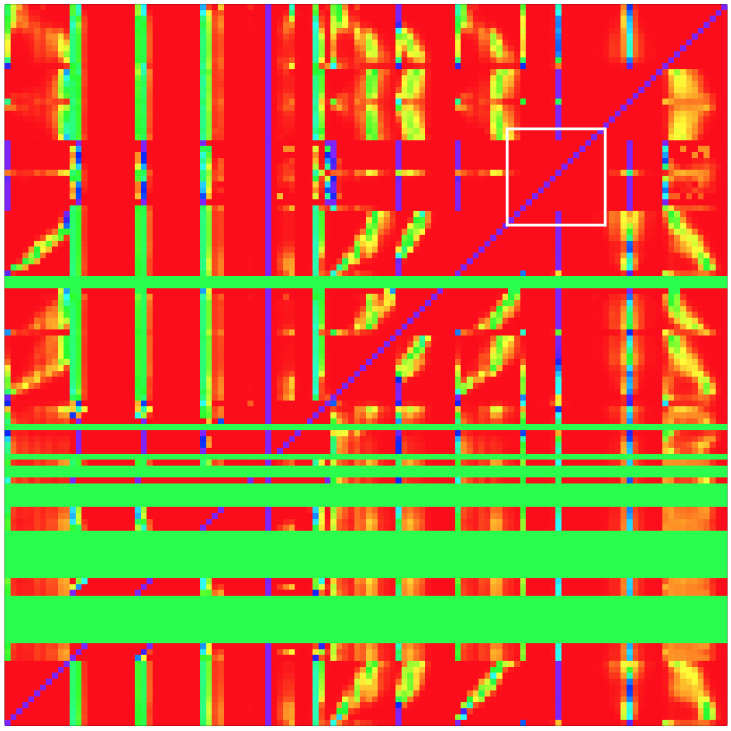

Figure 3: Correlation profile of `hep.txt`. From left to right, x varies from 0 to 121. From bottom to top, y varies from 0 to 121. The bright green horizontal lines are columns that are always 0. The violet vertical line is a column that is always 1. Inside the white square is a group of mutually exclusive columns: a violet diagonal running through a solid red area.

2.2 Categorising and clustering columns

At this stage a lot is known about the structure of the data sets through their statistical profiles. The next step is to turn continuous statistical information into discrete decisions for clustering the columns.

First, high and low density columns should be identified and excluded. This is pretty simple, during the building of a profile of the data all the rows are summed along their columns. Low density columns can be identified by having less than 0.01% of their rows filled. Conversely, high density columns have more than 99.99% of their rows filled.

Once high and low density columns have been excluded, all the remaining columns need to be grouped into mutually exclusive groups. In order to find the right algorithm for this problem, mutual exclusion must be understood. What kind of relation is mutual exclusion? It's symmetric: $M(a,b) \iff M(b,a)$, non-reflexive: $M(a,b) \iff a \neq b$, and non-transitive: $M(a,b) \wedge M(b,c) \not\Longrightarrow M(a,c)$. In graph theory terms, mutually exclusive groups are cliques.[1] A clique is a set of nodes in a graph whose induced subgraph is complete. In order to cluster the columns into the fewest number of groups it is necessary to find the minimal clique cover for the graph. See figure 4 for an example of this.

Unfortunately, finding a minimal clique cover is an NP-hard problem.[2] Because of this a greedy algorithm has been used instead that doesn't necessarily find a minimal clique cover.

In figure 4 you can see the column groups that this method produces when applied to the `hep.txt` data set.

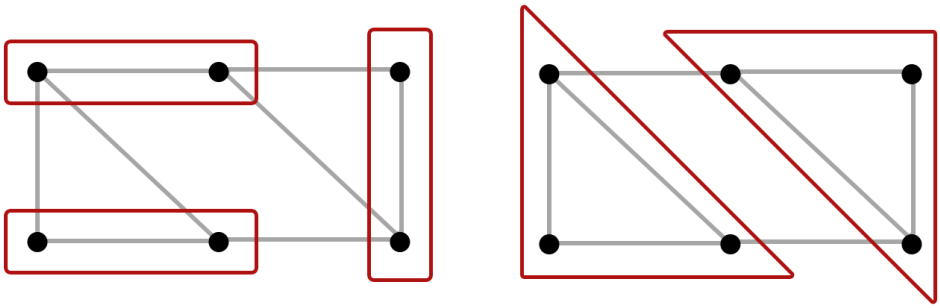

Figure 4: Two clique covers of the same graph. On the left is a cover that a greedy algorithm might end up finding. On the right is a minimum cover.

```
Low Density Columns: C14, C15, C16, C17, C18, C19, C20, C21, C25, C26, C27,
 C28, C29, C30, C31, C32, C37, C38, C39, C40, C41, C42, C43, C45, C49, C50,
 C51, C72, C73, C74, C75, C88, C89, C90, C91, C94, C95, C96, C97, C98, C99,
 C100, C108, C109, C110, C120, C121

High Density Columns: C44

Mutually Exclusive Column Groups:
Group 0: C0, C1, C2, C3, C4, C5, C6, C7, C8, C9, C10
Group 1: C11, C12, C13
Group 2: C14, C15, C16
Group 3: C17, C18, C19, C20
Group 4: C21, C22, C23, C24, C25, C26
Group 5: C27, C28, C29, C30, C31, C32, C33, C34, C35, C36, C37
Group 6: C38, C39, C40, C41, C42, C43
Group 7: C44, C45, C46, C47, C48, C49, C50, C51, C52, C53, C54
Group 8: C55, C56, C57
Group 9: C58, C59, C60, C61, C62, C63, C64
Group 10: C65, C66, C67, C68, C69, C70, C71, C72, C73

Program ended with exit code: 0
```

Figure 5: The standard output from running `./analyse` on `hep.txt`.

Data set	Number of rows	`reorder` time	`reorder-lexico` time	Evaluation
`uniform.txt`	200000	0.65s	0.35s	**0.30s slower**
`zipf1.txt`	200000	0.62s	0.38s	**0.24s slower**
`zipf2.txt`	200000	0.57s	0.36s	**0.21s slower**
`hepdistinct.txt`	353889	2.86s	2.39s	**0.47s slower**
`hep.txt`	2173762	6.16s	13.47s	**7.31s faster**

Figure 6: Performance of `reorder` compared to `reorder-lexico`. All tests were carried out on the DCS lab machines.

Data set	Baseline ratio	`reorder` ratio	`reorder-lexico` ratio	Evaluation
`uniform.txt`	103.08%	29.47%	30.37%	**97.04% of lexico**
`zipf1.txt`	99.43%	19.80%	21.62%	**91.58% of lexico**
`zipf2.txt`	73.56%	5.69%	6.32%	**90.03% of lexico**
`hepdistinct.txt`	43.16%	19.47%	21.29%	**91.45% of lexico**
`hep.txt`	38.38%	13.00%	6.95%	**187.05% of lexico**

Figure 7: Compression ratio of `reorder` compared to `reorder-lexico` and the unsorted file. The ratios are calculated by the `eval` program.

3 The Algorithm

In short, this is the proposed algorithm:

1. Read bit file.
2. Sum by columns.
3. Separate columns up into: low, high, and normal.
4. Calculate cross sums for normal columns.
5. Greedily cluster columns into mutually exclusive groups.
6. Sort file by mutually exclusive groups, prioritising large groups over small groups.
7. Save bit file.

4 Results

Figures 6 and 7 show the performance of the reordering program and the compression efficiency respectively. It appears that the `reorder` program is slightly slower than `reorder-lexico` on small input sizes ($\approx$ 200000 rows), however it is significantly faster on `hep.txt`, the largest data set. This must be due to the statistical profiling that `reorder` must do before it starts reordering any file. For small data sets, the time saved by intelligently sorting groups of columns together is lost on the profiling necessary for building the groups. On larger data sets the time saved makes up for the time spent profiling.

In terms of efficiency, the results are not very impressive. On the smaller data sets `reorder` produces orderings that compress to $\approx 90\%$ of the size produced by `reorder-lexicographic`.

However, for the largest data set, `hep.txt`, the results are very poor: the ordering is almost twice as large as the lexicographic ordering.

5 Conclusion

To conclude, the efficiency is not quite good enough. Although the speed of the algorithm is fairly good, the inefficient ordering on `hep.txt` makes the speed-gain pointless.

To reflect, the idea to search for structure within the data sets seems promising. There will inevitably be some pattern in the data that will make sorting easier and compression more efficient. However, the method by which the profile was translated into column clusters might be flawed. Trying to identify mutually exclusive groups may not be the right structure to search for. An alternative could have been to look for columns that are similar to one another.

All in all, this has been an incredibly interesting project to work on, in spite of the results not meeting my personal expectations.

6 Appendix: Source code

6.1 Compiling

All the executables can be compiled from source with CMake. In the source code directory run "`cmake .`" followed by "`make`" to compile all executables.

6.2 Pre-compiled Binaries

In case the C++ compiler is not up-to-date, or CMake is not up-to-date, pre-compiled binaries have been included as well in the `binaries/` directory. These have been compiled on the DCS lab machines.

6.3 Executable: `eval`

Due to issues with the Java `eval` program that was provided with the project, I decided to write my own version in C++.

Usage:

```
$ ./eval path/to/bit/file.txt
```

`eval` will compress the given file along its columns using WAH run-length encoding with words of size 31. It will print a comma-separated list of data to `stdout`, breaking down the quality of compression for each column as well as showing the result for the entire file.

6.4 Executable: `visualise`

The `visualise` executable will be able to produce the correlation profile bitmaps included in this report.

Usage:

```
$ ./visualise path/to/bit/file.txt profile1.bmp profile2.bmp profile3.bmp
```

`visualise` will output three different bitmaps: the first represents $A := P(C_y = 1|C_x = 1)$, the second represents $B := P(C_y = 1|C_x = 0)$, and the third represents $\frac{A-B}{P(C_y=1)}$.

6.5 Executable: `analyse`

The `analyse` executable will be able to produce the listing of mutually exclusive groups for any data set.

Usage:

```
$ ./analyse path/to/bit/file.txt
```

6.6 Executable: `reorder`

The `reorder` executable will reorder a given bit file following the algorithm described in this report.

Usage:

```
$ ./reorder path/to/bit/file.txt path/to/bit/file_out.txt
```

6.7 Executable: `reorder-lexico`

The `reorder-lexico` executable will reorder a given bit file lexicographically. This program acts as the baseline for measuring results.

Usage:

```
$ ./reorder-lexico path/to/bit/file.txt path/to/bit/file_out.txt
```

References

[1] en.wikipedia.org (2017). *Clique (graph theory)* [online] Available at: `https://en.wikipedia.org/wiki/Clique_(graph_theory)` [Accessed 4 Dec. 2017]

[2] en.wikipedia.org (2017). *Clique cover – Computational complexity* [online] Available at: `https://en.wikipedia.org/wiki/Clique_cover#Computational_complexity` [Accessed 4 Dec. 2017]