# Ordering algorithm used in ExpressionView

Andreas Lüscher

April 16, 2015

## Contents



## 1 Introduction

Clustering genes according to their expression profiles is an important task in analyzing microarray data. To present the mutually overlapping biclusters in a visually appealing layout, the gene expression matrix has to be arranged in such a way that biclusters form contiguous rectangles. For more than two biclusters, this is in general impossible. Here, we present an algorithm that finds the arrangement that maximizes the areas of the largest contiguous parts of the biclusters.

## 2 Problem

The problem can be formulated very generally: given a matrix in which each element is part of at least one bicluster, arrange the rows and columns in such a way that the biclusters are easily recognizable on a two-dimensional plot. Since biclusters are rectangular, rows and columns can be ordered separately. The problem thus reduces to finding the optimal arrangement of a set of $n$ elements that are part of at least one of $m$ clusters. In what follows, we refer to the $n$ elements as slots.

## 3 Quantifying the order

There are several ways of quantifying a "visually appealing layout". We have chosen to define the quality of the order $Q$ as the sum over the maximal number of neighboring elements in every cluster, as illustrated in Fig. 1.

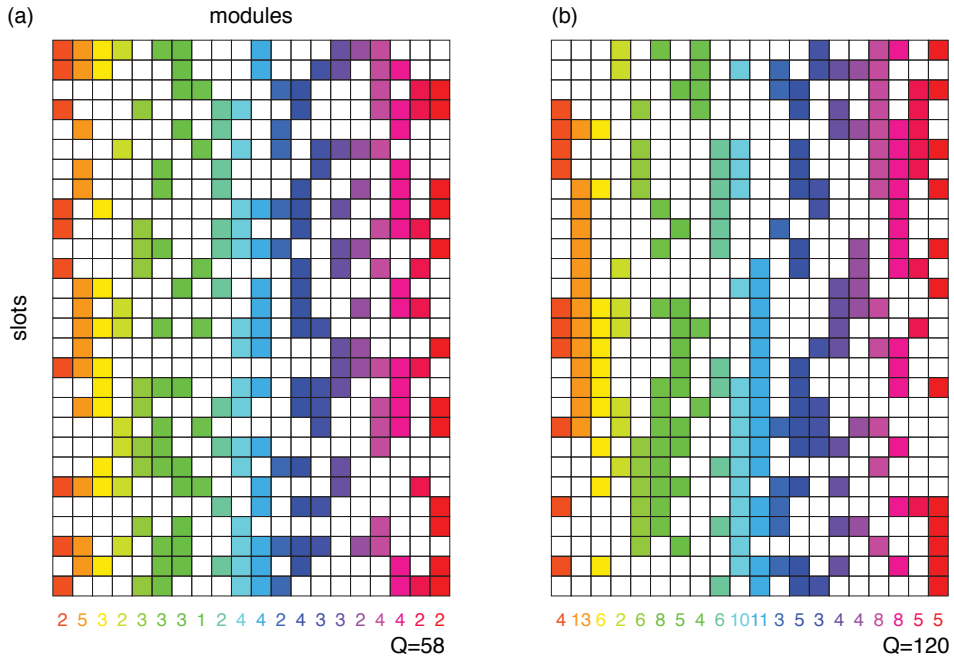


Figure 1: The initial configuration (a) with a score $Q$=58 can be rearranged by modifying the order of the slots. The optimal solution (b) maximizes $Q$.

# 4 Optimizing the order

Optimizing the order thus consists in maximizing $Q$ by changing the order of the slots. To achieve this goal we use a greedy approach that

- shifts parts of a given cluster to better positions and
- permutes elements contained in a given cluster.

Internally, the data is represented as a binary matrix (i.e., elements are either 0 or 1) with dimensions $m \times n$, where the number of rows $m$ correspond to the number of genes or conditions and the number of columns $n$ is determined by the number of clusters.
The ordering is done in four steps:

1. `simplify`: Reduces the dimensions of the problem by excluding duplicate rows. The multiplicity of each slot is taken into account when calculating the score $Q$.
2. `prearrange`: Tries to find a good starting point for the optimization by creating a new order, starting from the first row and adding the consecutive rows one after the other, placing them at the optimal position. For smaller samples, the fitness function is used, while for larger samples only the similarity (scalar product) between rows is taken into account.
3. `arrange`: Uses the greedy approach to further increase the alignment score.
4. `complexify`: Reinstates the original dimensions of the problem by inserting the omitted rows at the newly found position.

If the user specifies a time limit, the algorithm stops the alignment after the elapsed interval and returns the best alignment. Otherwise, it continues until it can no longer improve the alignment score.

The ordering iterates over the clusters and first tries to shift the rows of them to a better position. A contiguous group of rows can be repositioned at once or split up in subgroups. Empirically, it turned out that the alignment is fastest by starting the repositioning with a single row, increasing the length of the group only if no better position could be found.
After repositioning, the algorithm tries to exchanges rows contained or not contained in a given cluster. It randomly picks a number of rows (up to six right now) and generates all their possible permutations, calculates the corresponding alignment scores, and retains the best order. This realignment is repeated a number of times, so that on average every slot is part of one optimization.
These two steps are repeated until the maximal computation time is reached or no improved alignment could be found anymore.
The `C++` program can also be compiled to a standalone executable by defining `-DSTANDALONE` to exclude R-specific functions. You can also define `-DTIMING` to measure the time spent in the function that calculates $Q$ (`getfitness`).

# 5 Calculation time

To get an idea of the complexity of the algorithm, we determined the scaling of the calculation time with the dimensions of the data, i.e., the number of clusters $m$ and the slots $n$. The results, obtained as the average over at least 40 runs, are summarized in Figs. 2 and 3. We consider orderable and completely random samples. In both cases, the time increases polynomially with the number of clusters as $\mathcal{O}(m^\alpha)$, but with different exponents, see Fig. 2: For random samples, we find $\alpha \in [1.6, 2]$, almost independent of the number of elements $n$, while for orderable samples, $\alpha \in [2.3, 4.3]$. We note that despite of having a slower increase of computation time with $m$, it takes considerably more time to order random samples.
The scaling of the computation time with the number of slots $n$ is shown in Fig. 3. Here, only the random samples exhibit a polynomial dependence $\mathcal{O}(n^\beta)$, with $\beta \in [2.5, 2.7]$. The orderable batch saturates at larger $n$. This is due to the fact that by increasing the number of slots $n$ while keeping the number of clusters constant, the complexity of the problem stays roughly the same because most of the additional slots are equal to those already contained in the data.

# 6 Possible improvements

More than 90% of the calculation time is spent to recalculate the length of the longest contiguous subclusters. Optimizing this algorithm thus has a direct impact on the overall performance of the algorithm. Since the different clusters can be treated independently, it is straightforward to parallelize the calculation, maybe even on the graphic chip.

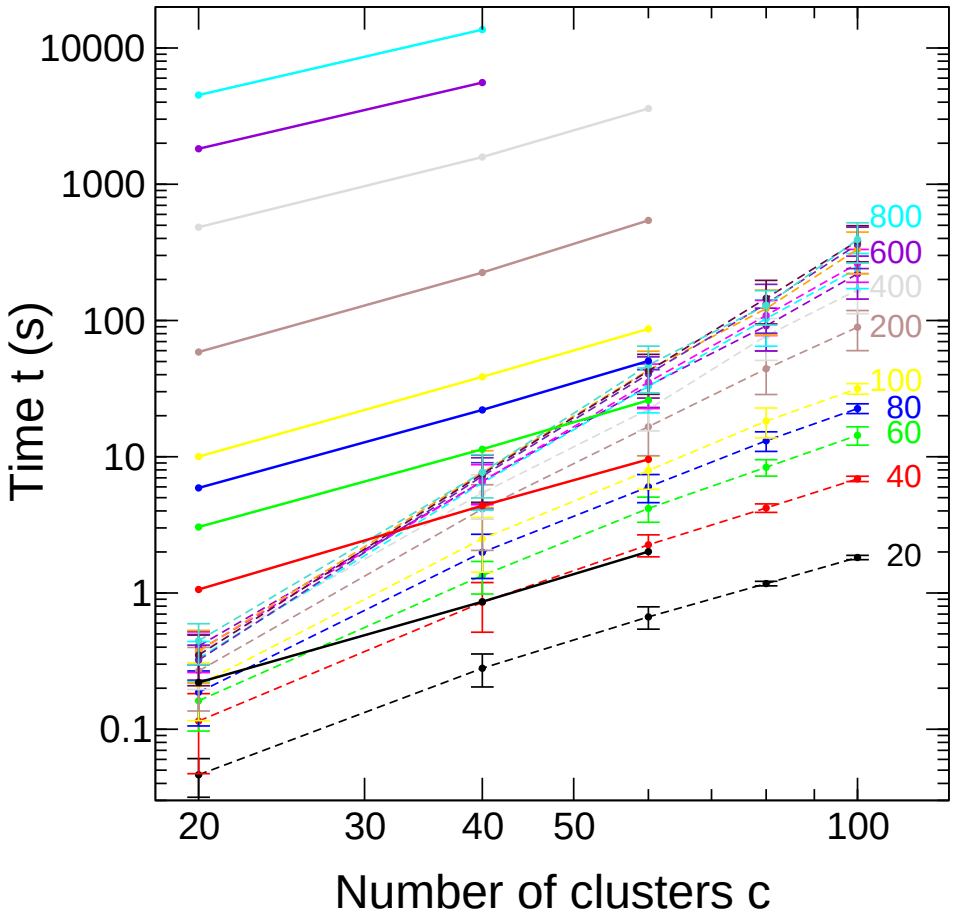


Figure 2: Scaling of the execution time as a function of the number of clusters. Data points connected by dashed lines represent results obtained from orderable samples. The second set is obtained from random samples and is thus more representative of gene expression data. Both sets scale polynomially with the number of clusters $m$ as $\mathcal{O}(m^{\alpha})$, with $\alpha \in [1.6, 2]$ for random samples and $\alpha \in [2.3, 4.2]$ in the case of orderable samples.

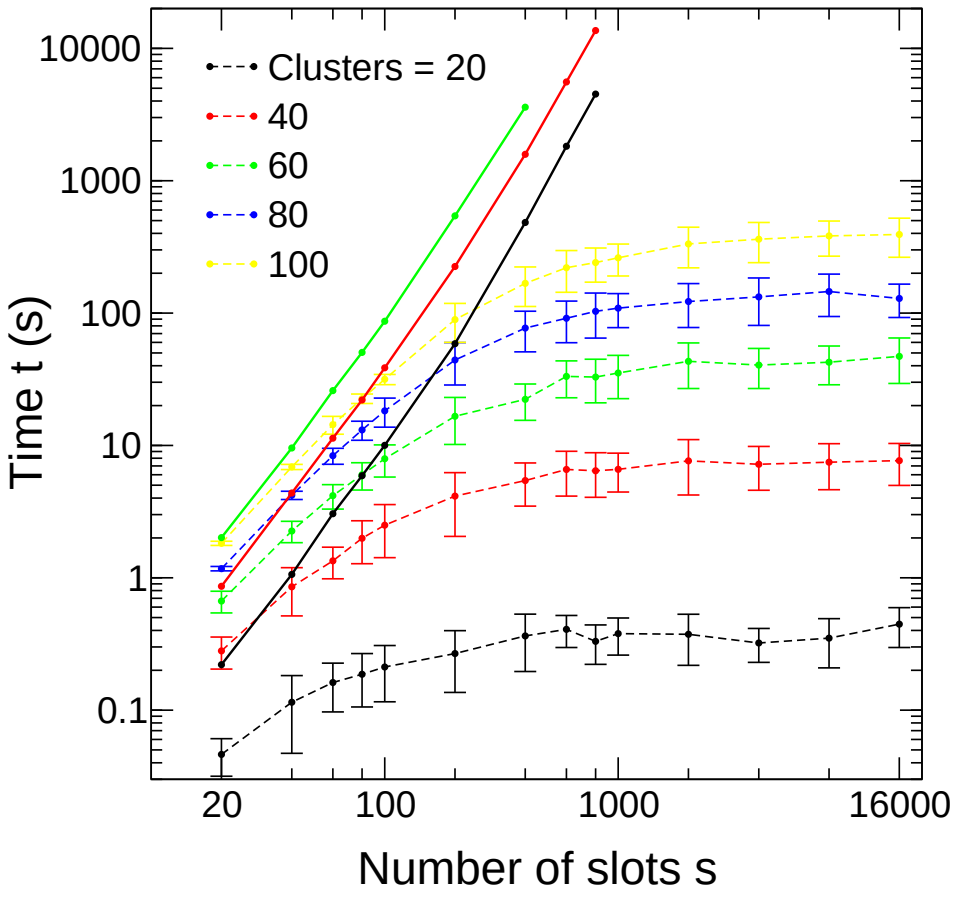


Figure 3: Scaling of the execution time as a function of the number of slots. The second set is obtained from random samples and is thus more representative of gene expression data. The random set scales polynomially with the number of slots $n$ as $\mathcal{O}(n^{\beta})$, with $\beta \in [2.5, 2.7]$. The orderable data set shows an initially polynomial increase and then saturates for larger $n$.