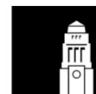
UNIVERSITY OF LEEDS

This is a repository copy of *Minimising the number of gap-zeros in binary matrices*.

White Rose Research Online URL for this paper:
http://eprints.whiterose.ac.uk/79240/

Version: Submitted Version

Article:

Glass, CA and Shakhlevich, NV (2013) Minimising the number of gap-zeros in binary matrices. European Journal of Operational Research, 229 (1). 45 - 58. ISSN 0377-2217

https://doi.org/10.1016/j.ejor.2013.01.028

Reuse
Unless indicated otherwise, fulltext items are protected by copyright with all rights reserved. The copyright exception in section 29 of the Copyright, Designs and Patents Act 1988 allows the making of a single copy solely for the purpose of non-commercial research or private study within the limits of fair dealing. The publisher or other rights-holder may allow further reproduction and re-use of this version - refer to the White Rose Research Online record for this item. Where records identify the publisher as the copyright holder, users can verify any specific terms of use on the publisher's website.

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

Minimising the Number of Gap-Zeros in Binary Matrices

Konstantin Chakhlevitch [a], Celia A. Glass [a,*], Natalia V. Shakhlevich [b]
[a] Cass Business School, City University London, London, EC1Y 8TZ, United Kingdom,
[b] School of Computing, University of Leeds, Leeds, LS2 9JT, United Kingdom.

Abstract

We study a problem of minimising the total number of zeros in the gaps between blocks of consecutive ones in the columns of a binary matrix by permuting its rows. The problem is refered to as the Consecutive One Augmentation Problem, and is known to be NP-hard. An analysis of the structure of an optimal solution allows us to focus on a restricted solution space, and to use an implicit representation for searching the space. We develop an exact solution algorithm, fixed-parameter tractable with respect to the number of columns, and two constructive heuristics to tackle instances with an arbitrary number of columns. The heuristics use a novel solution representation based upon row sequencing. In our computational study, all heuristic solutions are either optimal or close to an optimum. One of the heuristics is particularly effective, especially for problems with a large number of rows.

Keywords: Combinatorial Optimisation; Scheduling; Heuristics

1 Introduction

In this paper, we consider a Combinatorial Optimisation problem which has several practical applications. Given a binary matrix with m rows and n columns, we aim to find a permutation of its rows with the minimum total number of zero-entries situated between the blocks of consecutive ones in the columns of the permuted matrix.

This study was motivated by the following real-world problem arising in the food testing process at Premier Food UK's microbiological laboratory. Food samples each undergo a suite of food safety tests. Each of these tests requires a specific type of agar media suitable for growing the particular contaminant which is being tested for. The food samples presented to the laboratory have a broad range of required tests, as raw peanuts and pre-cooked pizza, for example, are subject to a very different combination of bacteria and moulds. The test suites,

*Corresponding author. Tel.: +44 (0)20 7040 8959; fax: +44 (0)20 7040 8572; e-mail: c.a.glass@city.ac.uk

and hence the combinations of agar requirements, thus vary greatly. Now, the laboratory produces its own agar, and is therefore responsible for the scheduling of the process. Each type of agar must be made before any test sample in which it is required can begin processing at the agar pouring stage. However, agar has a limited shelf life. The available volume of agar should, therefore, ideally be used in as many tests as possible before it expires. In order to better coordinate the production of the different types of agar with its usage, it is therefore important to sequence test samples for processing so that consumption of each agar is not unnecessarily drawn out. This can be achieved by selecting a sequence of test samples which minimises the total elapse time during which any agar is in demand.

The microbiology laboratory scheduling problem can then be formulated using a (0,1) incidence matrix whose columns correspond to agar types and rows to test samples. The matrix thus specifies the agar combination required for the suite of tests of each test sample. Changing the sequence in which test samples are processed in the laboratory, corresponds to reordering the rows in the matrix. Now the equipment pours agar onto test plates one at a time, and each test plate takes the same length of time to process. Thus, the sample's processing time is proportional to its number of test plates. Any agar not involved in the sample's test suite will remain unused while the sample is being processed. Gaps in usage of agar, correspond to 0's which occur between consecutive 1's in the corresponding column of the matrix, so called *gap-zeros*. Thus, by taking the total number of test plates associated with each sample as the corresponding row weight, the weighted sum of gap-zeros in a column, measures the length of the gap(s) in usage of the corresponding agar.

The solution approach developed in this article could therefore be applied in the microbiology laboratory context, to contribute to significant efficiency gains, of over 20%, as reported in [8]. The saving is achieved by judicious batching of agar production, following the approach to batching perishable goods described in [7]. It is only at this further stage of analysis that account is taken of the volume of the individual agar requirement of test samples. Other applications discussed in the literature include the design of the storage schemes for sparse matrices [22] and the physical mapping of chromosomes in computational biology [14].

Our problem extends the class of problems concerned with the consecutive ones property for binary matrices. A given (0,1)-matrix satisfies the *Consecutive Ones Property (for columns)* (C1P) if there exists a permutation of its rows in which the 1's in each column appear consecutively. The C1P was introduced by Fulkerson and Gross [11] in the context of recognising interval graphs. A broader survey of the C1P problem is provided by Dom [9]. Dom also demonstrates how the C1P property of input data infer polynomial solvability on both Integer Linear Programming and certain variants of Set Covering. Ruf and Schobel UMLAUT! extend the result for Set covering by showing how input matrices which almost have the C1P property, allow an efficient solution using a Branch and Bound scheme.

The problem of verifying the C1P arises in archaeology [18], file organisation [13], [19], and especially in computational biology (DNA sequencing) [1], [2], [4], [14]. A permutation of rows with the desired property can be found efficiently, if one exists. This can be done by decomposition techniques [11], [17], which can be implemented in $O(n^2)$ and $O(m+n+r)$ time, and using the PQ-tree data structure proposed in [5], [6] in $O(\max\{n,m\}+r)$ time,

recognising interval graphs in [15] in linear time. where m, n and r are the number of rows, of columns and of 1's in the matrix, respectively.

For many practical problems, the C1P is not satisfied and it is therefore important to transform a given matrix to one in which the 1-entries are consecutive in as many columns as possible. There are two versions of such a problem. The first formulation is known as the *Consecutive Ones Submatrix* problem (COS) [12]: find the largest possible subset of columns of a given matrix so that for the resulting submatrix the C1P is satisfied. The decision version of the COS problem is known to be NP-complete [16] and it remains NP-complete even for some special cases, see [4] and [21]. For matrices with a restricted number of 1's in rows and columns, approximation algorithms are developed in [21] and [10].

In the second formulation, which we call the *Consecutive Ones Blocks* (COB) problem, the objective is to find a permutation of rows of a given binary matrix for which the number of blocks of consecutive 1's in the permuted matrix is minimised, see [14], [19]. An equivalent interpretation is to minimise the number of gaps in the matrix, i.e., the number of blocks of 0's between blocks of consecutive 1's in each column of the matrix [2]. This problem is proved to be NP-hard in [19]. It is equivalent to the Hamming Distance Travelling Salesman Problem, in which the cities are the rows of the matrix together with an additional row consisting of all 0's, and the distance between two cities is the Hamming distance between the corresponding rows, see [1], [2] and [14]. Heuristics for the COB problem such as greedy algorithms and local search techniques are discussed in [2].

Our problem is essentially concerned with how close a matrix is to having the C1P, except for gap zeros. The decision version, referred to as $C1PR_k$, to decide if changing no more than k 0 entries to 1's results in a matrix with the C1P. The matrix is then said to have the k-augmentation consecutive one property. This problem is referred to as the *Consecutive One Matrix Augmentation Problem* in the literature [20]. Booth [5] proves that the problem is NP-hard, as described in [20]. Veldhorst considers the problem of efficient storage of sparse matrices which is equivalent to the *Gap0* problem with the transpose of the input matrix, in [22]. Each column (row in [22]) of the matrix is stored as a vector of the entries between the first and the last block of 0's, along with the row (column in [22]) index of the first non-zero entry and the length of the significant vector. Minimising gap-zeros provides a good mechanism for coding sparse matrices with low storage requirements. Veldhorst studies a number of two-stage algorithms which perform row (column in [22]) sorting first and on-line insertion next, and shows that no approximation bounds can be derived for this class of algorithms. A heuristic algorithm for the *Gap0* problem is suggested by Greenberg and Istrail [14]. They experiment with the so-called spectral algorithm initially developed for the C1P problem in [3], evaluating its performance mainly in biological terms in the context of genome reconstruction without assessing the quality of heuristic solutions from the combinatorial optimisation point of view.

Thus, there is no effective solution method for the *Gap0* problem in the literature, and limited performance analysis except for the negative result for on-line algorithms which provides yet another motivation for this research.

The contribution of this paper is threefold. Firstly, we identify a novel solution representation which provides an efficient solution space for search algorithms. Secondly, we develop

an exact solution algorithm for the *Gap0* problem, whose complexity may be expressed as a function of the number of columns alone, independent of the number of rows, and hence is constant time for instances with a fixed number of columns. Thirdly, we propose two heuristic algorithms which are capable of finding high quality solutions quickly. These three aspects are developed in the paper in the following manner. In Section 2, we reformulate the *Gap0* problem in terms of a so-called end-zero problem with non-duplicate weighted rows, derive its general properties, and classify problem instances as those with and without, so called, overlapping 0's. In Section 3, we introduce a new representation of solution classes specified by their common structure. Based on this representation, we develop an exact algorithm for the *Gap0* problem with a fixed number of columns. The exact algorithm is based on indirect enumeration of the solution space by considering solution classes one at a time, finding an optimal solution in each class. The best such solution found determines the global optimum. Section 4 then describes our two heuristics for the *Gap0* problem, *GRIN* and *MAZE*. These are both based on insertion of row(s) iteratively into a partial solution. Heuristic *GRIN* considers rows in a random order and selects the insertion position in a greedy fashion, while heuristic *MAZE* has a fixed position for insertion at each step and for that position it selects the most appropriate row(s) in a greedy fashion. In our computational experiments, we focus on instances with unit row weights and without duplicate rows. However, both heuristics, as well as the exact algorithm, can be generalised to handle instances with duplicate rows and arbitrary row weights, as outlined in Sections 3.3 and 4.3. The results of computational experiments with our heuristics are reported in Section 5. We show that both heuristics are very effective, especially *MAZE* which frequently finds optimal solutions for a range of problem instances. Finally, conclusions are presented in Section 6.

2 Preliminaries

In this section, we first formulate two version of the gap-zero problem, and explore some basic properties of optimal solutions. Instances are then classified into two types, in Section 2.2.

2.1 Problem Formulation

Let A be a binary matrix with m rows and n columns, i.e., $A = (a_{ij})$, $a_{ij} \in \{0, 1\}$, $i = 1, \ldots, m$, $j = 1, \ldots, n$, and $\mathbf{a}_i$ be a row-vector of length n which represents row i of matrix A, $i = 1, \ldots, m$. A 0-entry $a_{ij} = 0$ of matrix A is called a *gap-zero*, if there exist two $1'$s in column j, one above a_{ij} and another one below a_{ij} in matrix A; otherwise, a_{ij} is referred to as an *end-zero*.

Example 1 *Consider the binary matrix instance A below, with 8 rows and 4 columns. The unpermuted solution contains 12 gap-zeros (highlighted in bold) and 7 end-zeros.*

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & \mathbf{0} & 0 \\ 1 & \mathbf{0} & \mathbf{0} & 1 \\ 1 & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & 1 & 1 \\ 1 & 1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & 0 & \mathbf{0} & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}.$$

Denote by $\mu = (i_1 i_2 \dots i_m)$ a permutation of row indices of the original matrix A. Let *solution*, S or $S(\mu)$, refer to an $m \times n$ matrix which contains the rows of the original matrix A taken in the order given by μ, and let $g(S)$ denote the total number of gap-zeros in matrix S defining the *gap-zero function* g. Let Ω denote the set of solutions defined by all $m!$ possible row permutations of the original matrix A. Since solutions are equivalent up to matrix row inversion, the size of the solution space can be reduced by half to $m!/2$. The *Gap0* problem may now formally be defined as follows.

Problem Gap0*: Given a binary matrix A, find a permutation μ^* of the rows for which the corresponding solution S^* has the minimum number of gap-zeros i.e.,*

$$g(S^*) = \min_{S \in \Omega} g(S).$$

Observe that instances are equivalent up to column permutation.

The search space may be further reduced when row duplication is taken into account as we now show.

Lemma 1 *There exists an optimal solution to problem Gap0 in which rows with identical entries are adjacent.*

The lemma can be proved by considering pairs of identical non-adjacent rows and moving the one that contains more gap-zeros to be adjacent to the other one (observed also in Lemma 2.3 from [22]).

As a consequence of Lemma 1, we only need to consider solutions with adjacent duplicate rows. Suppose that z_i denotes the number of gap-zeros in row i in an optimal solution S^* and that there are w_i duplicates of row i which are adjacent to each other in S^*. Then the total contribution of the w_i duplicate rows to the objective function $g(S^*)$ is $w_i z_i$. Thus, an instance A of the original unweighted problem *Gap0* with duplicate rows, can be simplified by replacing each set of identical rows by a single row with the corresponding weight. This leads us to another equivalent formulation, with row weights and a weighted g function. Observe that the above argument may also be applied to the broader class of matrices which already have row weights, to remove row duplication.

Problem WGap0*: Given a binary a matrix A with row weights w_i and no duplicate rows, find a permutation of the rows μ^* which minimises the weighted sum $g(S^*)$ of gap-zeros.*

This formulation has the advantage that the solution space is limited in size depending only on the number of columns of matrix A, however many rows it has. Thus, for an arbitrary *Gap0* instance, we can perform pre-processing to achieve a *reduced instance* in which there are no duplicate rows, or zero rows. The upper limit on the number of different rows, m, is thus effectively $2^n - 1$.

We now explore the nature of the objective function, and transform the problem to one which is more amenable to algorithmic approaches. We define an *end-zero function* $f(S)$ representing the total number of end-zeros in a solution S. Denote by z the total number (weighted sum) of 0-entries in matrix A (and, therefore, in any solution S). The next result follows immediately from the fact that $g(S) = z - f(S)$.

Lemma 2 *Maximising the total number (weighted sum) of end-zeros is equivalent to minimising the total number (weighted sum) of gap-zeros.*

As we show in Section 3, any solution may be characterised by the structure of end-zeros in its top and bottom parts and, therefore, function $f(S)$ is preferable for algorithm design purposes. On the other hand, function $g(S)$, which should be as close to 0 as possible in an optimal solution, is more convenient for evaluating the accuracy of heuristics. For ease of exposition, and to simplify the evaluation of heuristics, we focus on the unweighted case. However, each algorithm which we develop is easily extended to the weighted case, and the mechanism for doing so is given alongside.

2.2 Instance Classification Related to Overlapping End-Zeros

Consider a solution S obtained from the initial matrix A by applying some permutation of rows. We split the set of end-zeros in matrix S into *top-zeros*, i.e., zeros situated above the first 1-entry in a column, and *bottom-zeros* which are below the last 1-entry in a column.

Example 2 *Given matrix A of Example 1, compare solutions S and S' shown below. Solution S has 9 top-zeros and 6 bottom-zeros which are separated by the borders. Solution S' is obtained from solution S by swapping row 4 (1010) and row 5 (0110). It has 9 top-zeros and 7 bottom-zeros, one more than S. thus, $f(S') = f(S) + 1$ matrix S', and S' is a better solution than S.*

$$S = \left(\begin{array}{lcccc} \scriptstyle 1) & 1 & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \scriptstyle 2) & 0 & 1 & \mathbf{0} & \mathbf{0} \\ \scriptstyle 3) & 1 & 1 & \mathbf{0} & \mathbf{0} \\ \scriptstyle 4) & 1 & 0 & 1 & \mathbf{0} \\ \scriptstyle 5) & 0 & 1 & 1 & \mathbf{0} \\ \scriptstyle 6) & 1 & \mathbf{0} & 0 & 1 \\ \scriptstyle 7) & \mathbf{0} & \mathbf{0} & 1 & 1 \\ \scriptstyle 8) & \mathbf{0} & \mathbf{0} & \mathbf{0} & 1 \end{array}\right), \qquad S' = \left(\begin{array}{lcccc} \scriptstyle 1) & 1 & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \scriptstyle 2) & 0 & 1 & \mathbf{0} & \mathbf{0} \\ \scriptstyle 3) & 1 & 1 & \mathbf{0} & \mathbf{0} \\ \scriptstyle 4) & 0 & 1 & 1 & \mathbf{0} \\ \scriptstyle 5) & 1 & \mathbf{0} & 1 & \mathbf{0} \\ \scriptstyle 6) & 1 & \mathbf{0} & 0 & 1 \\ \scriptstyle 7) & \mathbf{0} & \mathbf{0} & 1 & 1 \\ \scriptstyle 8) & \mathbf{0} & \mathbf{0} & \mathbf{0} & 1 \end{array}\right).$$

Top-zeros and bottom-zeros in the same row of a solution S are said to be *overlapping end-zeros*. It appears that solutions without overlapping end-zeros are easier to specify and

to handle. For algorithmic purposes we now develop the means of identifying instances in which overlapping end-zeros cannot occur. We refer to an instance as *overlapping* if there exists a solution with overlapping zeros, and as *non-overlapping* otherwise. Observe that an instance is non-overlapping if it contains a row of all 1's.

Theorem 1 *An instance of the* Gap0 *problem given by an* $m \times n$ *matrix* A *is non-overlapping if and only if for any pair of* 0*-entries in columns* j *and* k *of row* i*, i.e.* $a_{ij} = a_{ik} = 0$*, there exists a pair of* 1*-entries in another row* i' *in the same columns* j *and* k*, i.e.* $a_{i'j} = a_{i'k} = 1$*.*

Proof. Suppose first that conditions of the theorem hold for all pairs of columns of matrix A which contain 0-entries in some rows. Then for any pair j and k of such columns with 0-entries in row i, the presence of 1 simultaneously in both columns in another row prohibits entries a_{ij} and a_{ik} from being overlapping end-zeros.

Suppose that the condition in the Theorem does not hold. Then there is a two, i, and two columns, j and k say, for which $a_{ij} = a_{i'k} = 0$, and $a_{ij} + a_{i'k} \leq 1$ for all $i' = i$. Consider the solution constructed from top to bottom as follows: rows i' with $a_{i'j} = 0$ in any order; then row i; followed by rows i' with $a_{i'k} = 0$ in any order. This solution has overlapping end-zeros in row i, and hence A is overlapping, as claimed in the Theorem. ■

Theorem 1 implies that recognising overlapping or non-overlapping instances requires $O(mn^2)$ time, since there are $n(n-1)/2$ pairs of columns to test and for each pair of columns, $m-1$ pairs of entries should be scanned once.

3 Exact Algorithm

In this section we formulate an exact algorithm for finding an optimal solution for the *Gap0* problem and then generalise it for *WGap0*. The algorithm is applicable to instances with a fixed number of columns n. Observe that the number of rows m can be as large as $2^n - 1$ for reduced instances and even larger for arbitrary *Gap0* instances with duplicate rows. Therefore the straightforward enumeration of all $m!$ row permutations is impractical. Our objective is to develop an algorithm which is polynomial with respect to the number of rows m.

The algorithm is based on splitting the solution space into classes and finding an optimal solution within each class. We define solution classes for instances with and without overlaps in Sections 3.1-3.2 and explain how an optimal solution can be found within each class. Finally, the generalisation of the exact algorithm for the weighted case is discussed in Section 3.3.

3.1 The Algorithm for Non-Overlapping Instances

Consider the case of a non-overlapping problem instance introduced in Section 2.2. Each row in a solution can only contribute to either top-zeros or bottom-zeros. The middle rows (if any) contain insignificant zeros which do not contribute to $f(S)$ and their sequence is of no consequence. We may therefore focus upon the rows at the top and at the bottom of the solution matrix separately when evaluating the end-zero function $f(S)$.

The main idea of the approach we propose is based on splitting the solution space into solution classes, characterised by the shapes of top-zeros and bottom-zeros. In each class the algorithm either finds a solution with the maximum number of end-zeros or it finds a solution which does not belong to the class but has more end-zeros than any solution in that class. Before giving the required definitions and algorithm description, we provide an illustrative example.

Example 3 *Consider the input matrix A which satisfies non-overlapping properties identified in Theorem 1, and solution S with the same row sequence as matrix A.*

$$A = S = \begin{pmatrix} 1 & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ 1 & 1 & \mathbf{0} & \mathbf{0} \\ 1 & 0 & 1 & \mathbf{0} \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & \mathbf{0} \\ \mathbf{0} & 1 & 1 & \mathbf{0} \\ \mathbf{0} & 1 & \mathbf{0} & \mathbf{0} \end{pmatrix}$$

The top-zeros of S are characterised by permutation $\pi = (1234)$ ranking the columns in increasing number of top-zeros and the bottom-zeros of solution S are characterised by permutation $\sigma = (2314)$ of column numbers, similarly.

Given a pair of permutations π and σ, we can introduce the associated sequence of *row patterns* $\mathcal{P}_{\pi,\sigma}$ (or simply 'patterns') which emulate the shapes of end-zeros corresponding to π and σ. For example, for $\pi = (1234)$ and $\sigma = (2314)$ we have

$$\mathcal{P}_{\pi,\sigma} = \begin{pmatrix} * & 0 & 0 & 0 \\ * & * & 0 & 0 \\ * & * & * & 0 \\ * & * & * & * \\ * & * & * & 0 \\ 0 & * & * & 0 \\ 0 & * & 0 & 0 \end{pmatrix}. \tag{1}$$

Each pattern consists of 0's which contribute to the end-zero function $f(S)$ and $*$ corresponding to an arbitrary 0/1-element whose actual value is insignificant for this purpose. We refer to the end zeros in a row pattern as *significant* 0's. Observe that the zero row vector will not arise as a row pattern, since we have restricted attention to reduced instances which exclude matrices with a zero row. The solution space may thus be partitioned into *classes* $\mathcal{C}(\pi, \sigma)$ of solutions whose top end-zero columns have length ranked in the order given by π, and the bottom end-zero columns satisfy σ, for all pairs of column permutations (π, σ). We therefore refer to the sequence of patterns $\mathcal{P}_{\pi,\sigma}$ as a *description* of the class $\mathcal{C}(\pi, \sigma)$.

We have illustrated how a solution gives rise to an end-zero row-pattern sequence. We now describe how to construct a solution with the maximum number of significant zeros within a given solution class $\mathcal{C}(\pi,\sigma)$. We say that a row i of matrix A *matches* a pattern within sequence $\mathcal{P}_{\pi,\sigma}$, if that row has 0's in all significant positions of the pattern. Thus, allocating a matching row i to a pattern with k significant 0's ensures a contribution of k to the end-zero function $f(S)$. If row i has several matching patterns, it is therefore beneficial to allocate it to a matching pattern with the maximum number of significant 0's. Having performed allocation of rows to patterns, a solution S can be constructed by placing the rows allocated to each pattern in the order the patterns appear in $\mathcal{P}_{\pi,\sigma}$; the rows allocated to the same pattern can be sequenced arbitrarily without losing the significant end-zeros related to $\mathcal{P}_{\pi,\sigma}$. The formal description of this procedure is captured in algorithm *ClassOpt* presented at the end of the section.

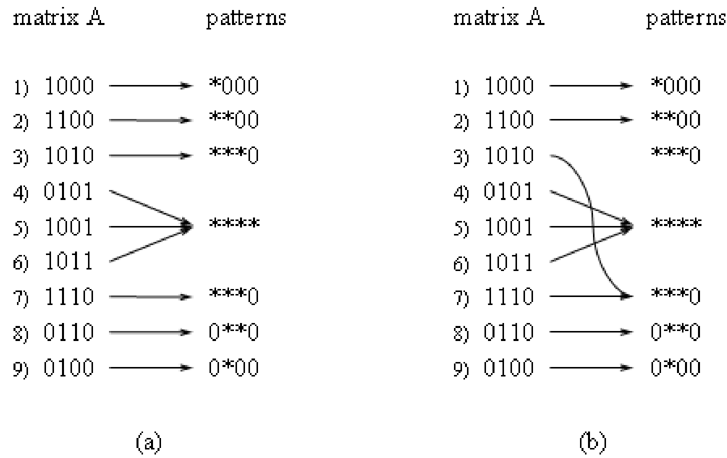

Figure 1: Two possible allocations of the rows of matrix A to the patterns of class $\mathcal{C}(\pi,\sigma)$

It is worth noting that this procedure may result in a solution with some additional end-zeros which lay outside the initial description $\mathcal{P}_{\pi,\sigma}$. For example, there are two options for allocating row 1010 in Example 3, each resulting in one significant 0 in $\mathcal{P}_{\pi,\sigma}$, see Fig. 1 (a) and (b). The first allocation represented in Fig. 1 (a) results in solution S; the second allocation represented in Fig. 1 (b) results in solution S', which has two additional top-zeros (marked by double lines) not in $\mathcal{P}_{\pi,\sigma}$:

$$S=\begin{pmatrix} 1 & \mathbf{0} & \mathbf{0} & \mathbf{0}\\ 1 & 1 & \mathbf{0} & \mathbf{0}\\ 1 & 0 & 1 & \mathbf{0}\\ 0 & 1 & 0 & 1\\ 1 & 0 & 0 & 1\\ 1 & 0 & 1 & 1\\ 1 & 1 & 1 & \mathbf{0}\\ \mathbf{0} & 1 & 1 & \mathbf{0}\\ \mathbf{0} & 1 & \mathbf{0} & \mathbf{0} \end{pmatrix} \qquad S'=\begin{pmatrix} 1 & \mathbf{0} & \mathbf{0} & \mathbf{0}\\ 1 & 1 & \mathbf{0} & \mathbf{0}\\ 0 & 1 & \mathbf{0} & 1\\ 1 & 0 & \mathbf{0} & 1\\ 1 & 0 & 1 & 1\\ 1 & 0 & 1 & \mathbf{0}\\ 1 & 1 & 1 & \mathbf{0}\\ \mathbf{0} & 1 & 1 & \mathbf{0}\\ \mathbf{0} & 1 & \mathbf{0} & \mathbf{0} \end{pmatrix}.$$

Thus, the solution S' constructed for the description $\mathcal{P}_{\pi,\sigma}$ actually belongs to a different class, namely $\mathcal{C}(\pi'', \sigma)$ where $\pi'' = (1243)$.

To handle cases similar to the one described above, we introduce an auxiliary function $f_{\pi,\sigma}(S)$ defined as the number of significant 0's of solution S with respect to pattern $\mathcal{P}_{\pi,\sigma}$. Thus,

$$\begin{aligned} f(S) &\geq f_{\pi,\sigma}(S) \text{ for any solution } S, \text{ and} \qquad (2) \\ f(S) &= f_{\pi,\sigma}(S) \text{ if solution } S \text{ belongs to class } \mathcal{C}(\pi,\sigma). \qquad (3) \end{aligned}$$

In particular, for the above example we have

$$\begin{aligned} f_{\pi,\sigma}(S) = 12, \quad f(S) = 12, \\ f_{\pi,\sigma}(S') = 12, \quad f(S') = 14. \end{aligned}$$

We then denote the maximum value of $f_{\pi,\sigma}(S)$ over all solutions from the class $\mathcal{C}(\pi,\sigma)$ by $f^*_{\pi,\sigma}$, and refer to a solution S as being *optimal in the class* $\mathcal{C}(\pi,\sigma)$ if

$$f(S) = f_{\pi,\sigma}(S) = f^*_{\pi,\sigma}. \qquad (4)$$

We now establish key properties of an optimal solution in a given solution class $\mathcal{C}(\pi,\sigma)$.

Theorem 2 *For a given pair of permutations π and σ and a solution S^* optimal in class $\mathcal{C}(\pi,\sigma)$, each row of S^* is positioned to correspond with a matching pattern from the class description $\mathcal{P}_{\pi,\sigma}$ which has the maximum number of significant 0's amongst all matching patterns.*

Proof. Consider a schedule S^* which is optimal in class $\mathcal{C}(\pi,\sigma)$ but does not satisfy the theorem. Then there exists a row, say i, which matches another pattern in the description $\mathcal{P}_{\pi,\sigma}$ with a larger number of significant 0's. Moving row i to the position corresponding to the pattern with more significant 0's increases the value of the end-zero function $f_{\pi,\sigma}(S)$, which contradicts the optimality of S^* within class $\mathcal{C}(\pi,\sigma)$. ■

Corollary 1 *If the solution class $\mathcal{C}(\pi,\sigma)$ is not empty, then allocating the rows of matrix A each to a matching pattern with the maximum number of significant 0's amongst those in $\mathcal{P}_{\pi,\sigma}$, produces a solution S with $f_{\pi,\sigma}(S) = f^*_{\pi,\sigma}$.*

Proof. Consider an optimal solution S^* in class $\mathcal{C}(\pi,\sigma)$ and also a solution S obtained by allocating the rows of matrix A to matching patterns with the maximum number of significant 0's. From Theorem 2, each row i of matrix A contributes the same number of significant 0's to function $f_{\pi,\sigma}$ in both solutions S and S^*, so that $f_{\pi,\sigma}(S) = f_{\pi,\sigma}(S^*) = f^*_{\pi,\sigma}$. ■

Thus, for a given pair of permutations, allocating the rows of the original matrix A to the matching patterns, within $\mathcal{P}_{\pi,\sigma}$, with the maximum number of significant 0's, gives rise to a solution S with $f_{\pi,\sigma}(S) = f^*_{\pi,\sigma}$. Now, if $\mathcal{P}_{\pi^*,\sigma^*}$ is the description corresponding to a globally optimal solution, S^*, then S^* is optimal in class $\mathcal{C}(\pi^*,\sigma^*)$, and $f(S^*) = f_{\pi^*,\sigma^*}(S^*) = f^*_{\pi^*,\sigma^*}$.

A globally optimal solution to the *Gap0* problem can therefore be identified by enumerating all classes $\mathcal{C}(\pi, \sigma)$, finding a solution which has the optimal value, $f^*_{\pi,\sigma}$, for each class, according to Corollary 1. In each class, a solution is constructed by allocating each row of the original matrix A to a matching pattern with the maximum number of significant 0's and the value of function $f_{\pi,\sigma}$ is recorded as described in procedure $ClassOpt(A, \pi, \sigma)$. Observe that if for some class $\mathcal{C}(\pi, \sigma)$ a solution S is found which does not belong to that class, then $f(S) > f_{\pi,\sigma}(S)$ and the global optimal solution will be found in a different class. A formal description of algorithm *ClassOpt* is given below.

Algorithm ClassOpt(A, π, σ)
`Input:` $m \times n$ `binary matrix` A `and permutations` π, σ
`Output:` `solution matrix` S `(at least as good as the optimal solution in class` $\mathcal{C}(\pi, \sigma)$`) and the value of the function` $f_{\pi,\sigma}(S)$

1. `Initialisation: create a sequence of row-patterns` $\mathcal{P}_{\pi,\sigma}$ `for class` $\mathcal{C}(\pi, \sigma)$ `and set` $f_{\pi,\sigma}(S) \leftarrow 0$.
2. `FOR each row of matrix` A
   2.1. `Select a matching pattern among the rows of` $\mathcal{P}_{\pi,\sigma}$, `with the maximum number of significant` 0`'s (break ties arbitrarily)`
   2.2. `Allocate the row to the selected pattern and update` $f_{\pi,\sigma}(S)$ `by adding the number of significant zeros matching the current row-pattern`

   `ENDFOR`
3. `Construct a solution` S `by placing the rows in the order of the class description` $\mathcal{P}_{\pi,\sigma}$ `from top to bottom. If there are several rows allocated to the same pattern, place them in an arbitrary order.`
4. `Return` S and $f^*_{\pi,\sigma} = f_{\pi,\sigma}(S)$

Algorithm $ClassOpt(A, \pi, \sigma)$ can be implemented in $O(mn)$ time, as shown in Appendix A. Since there are $(n!)^2$ different classes $\mathcal{C}(\pi, \sigma)$ and for each class there exists an equivalent counterpart with a reverse order of patterns, it is sufficient to consider $(n!)^2/2$ solution classes, so that the overall time complexity of finding the globally optimal solution is $O(mn(n!)^2)$.

3.2 The Algorithm for Overlapping Instances

We introduce a permutation ρ of $2n$ elements $\{1^-, 1^+, 2^-, 2^+, \ldots, n^-, n^+\}$ which we call the *end-zero sequence* of solution S. Permutation ρ defines the order in which the top-zero columns terminate and bottom-zero columns start if the solution matrix is scanned from top to bottom. Element j^- (j^+) relates to the top-zero (bottom-zero) portion of

column j and its position in the permutation ρ indicates when top-zeros (bottom-zeros) in column j terminate (start) relative to end-zeros in other columns. For example, end-zero sequences for solutions S and S' in Example 2 are given by $\rho = (1^- 2^- 3^- 4^- 2^+ 1^+ 3^+ 4^+)$ and $\rho' = (1^- 2^- 3^- 2^+ 4^- 1^+ 3^+ 4^+)$, respectively. The overlap in columns 2 and 4 of solution S' is indicated by entry 2^+ appearing before 4^-. On the other hand, the end-zero sequence ρ corresponds to a solution without overlaps, since all entries $1^-, 2^-, 3^-, 4^-$ for top-zeros precede to entries $2^+, 1^+, 3^+, 4^+$ for bottom-zeros.

Each end-zero sequence ρ defines a class of solutions which we denote by $\mathcal{C}(\rho)$. It can be described by a sequence of patterns $\mathcal{P}(\rho)$ in a similar way as for non-overlapping instances. For example, the class of solutions $\mathcal{C}(\rho')$ is described by the sequence of patterns

$$\mathcal{P}_{\rho'} = \begin{pmatrix} * & 0 & 0 & 0 \\ * & * & 0 & 0 \\ * & * & * & 0 \\ * & 0 & * & 0 \\ * & 0 & * & * \\ 0 & 0 & * & * \\ 0 & 0 & 0 & * \end{pmatrix}.$$

It is easy to check that Theorem 2 and Corollary 1 hold for overlapping instances and therefore algorithm *ClassOpt*(A, π, σ) can be converted into a corresponding algorithm *Class-Opt*(A, ρ). Step 1 is modified so that the sequence of patterns is created for the class $\mathcal{C}(\rho)$ rather than for $\mathcal{C}(\pi, \sigma)$, and Step 2.1 is extended to take care of the situation in which a row of matrix A does not have a matching pattern, and hence no feasible solution exists in class $\mathcal{C}(\rho)$. Recall that, by contrast, any class description of non-overlapping instances contains the 'universal' pattern $(* * \ldots * *)$, and hence a matching pattern always exists.

2.1. `Select a matching pattern with the maximum number of significant 0's,`
`if one exists; (break ties arbitrarily).`
`If no such matching exists, return 'no solution in class` $\mathcal{C}(\rho)$`' and STOP.`

In order to implement efficiently algorithm *ClassOpt* for overlapping instances, the ideas of Section 3.1 can be adopted so that the same time complexity of $O(mn)$ can be achieved, see Apendix B. To estimate the number of permutations ρ (classes $\mathcal{C}(\rho)$) which should be enumerated, observe that there are $(2n)!$ permutations ρ consisting of $2n$ elements j^-, j^+, $j = 1, \ldots, n$. Note that if a permutation contains an entry j^+ appearing before j^-, then this permutation is infeasible. Hence, we can discard all but one of the 2^n combinations of pairwise orientations of j^+ preceding j^- for each $j = 1, \ldots, n$. Taking into account that for each class defined by a permutation ρ, there exists an equivalent counterpart with a reverse order of patterns, the overall number of classes which need to be enumerated by the algorithm *ClassOpt*(A, ρ) is therefore $(2n)!/2^{n+1}$. Thus the time complexity of finding the globally optimal solution is $O(mn(2n)!/2^{n+1})$. Notice that the number of classes for overlapping instances is much larger than that for non-overlapping instances and therefore it is inefficient to use the class description $\mathcal{C}(\rho)$ for non-overlapping instances.

3.3 An Exact Algorithm for the Weighted Case

The above approach is equally applicable to the general problem in which rows may be duplicated. Moreover, it can be easily generalised to tackle problem instances with arbitrary weights of the rows. The only change required is related to calculating the objective value: instead of counting the number of end-zeros in a solution, the weighted number of end-zeros should be calculated in Step 2.1 of algorithms $ClassOpt(A, \pi, \sigma)$ or $ClassOpt(A, \rho)$. Conversely, given a *Gap0* instance, preprocessing to obtain the equivalent non-duplicate weighted instance of *WGap0* and then applying the generalised algorithm, may be more efficient than applying the *ClassOpt* algorithm directly. To be precise, if the original instance has m rows, m' of which are distinct, $m' \leq m$, then $m' \leq 2^n$ and the computational complexity of solving the instance to optimality reduces to $O((2^n)n(n!))$ or $O(n(2n)!)$. The problem is thus fixed-parameter tractable with respect to the parameter n.

4 Heuristics

In this section, we present two constructive heuristics for our problem *Gap0*.

4.1 A Greedy Insertion Heuristic

Our first heuristic, *GRIN* (for GReedy INsertion), employs a greedy strategy for constructing the sequence of rows. The rows are taken one at a time in some pre-specified order. At each iteration, a next currently unscheduled row, which we call a *target* row, is inserted into the partial sequence at the best possible position. This is achieved by considering all possible positions for row insertion and selecting the one for which the number of additional end-zeros created by the insertion is maximum. A formal description of heuristic *GRIN* appears in Appendix B; as we show there, its time complexity is $O(m^2 + mn)$.

We carried out initial experiments to establish suitable choices for pre-ordering the rows of the matrix A and for selecting between solutions of apparently similar quality. The experiments revealed that results produced by heuristic *GRIN* are highly dependent on the order of insertion of rows, and to a lesser extent on the tie-breaking rule. However, no specific ordering of rows, e.g. by the number of zero-entries, gives consistently good results across a range of problem instances. These observations are in line with Veldhorst's proposition [22] which states that no approximation bound can be derived for this class of greedy insertion algorithms. Since it is not clear in advance which order of row insertion should be selected to obtain the best outcome, we opted for running the heuristic *GRIN* on randomly generated orderings, multiple times, to get a range of solutions. In each separate run, the initial ordering of the rows of matrix A is generated randomly.

4.2 A Top-Bottom Construction Heuristic

In our second heuristic, *MAZE* (for MAximising the number of end-ZEros), we separately construct two parts of a solution matrix, namely the upper and the lower parts, aiming to

maximise the total number of end-zeros at both ends. The algorithm starts with allocation of two rows with the maximum number of zero-entries as the first and the last row, thus initialising the solution. The positions of zero-entries in these two rows define initial patterns $\overline{\mathbf{p}}$ and $\underline{\mathbf{p}}$ of end-zeros for the upper and lower parts of the solution.

At each subsequent iteration, the remaining empty rows of the solution matrix are filled in by unallocated rows of matrix A in two directions, by inserting rows in turn at the bottom of the upper part and then at the top of the lower part, thus moving towards the middle of the solution matrix. First an attempt is made to allocate row(s) to the upper part. We check for unallocated rows which match significant 0's of pattern $\overline{\mathbf{p}}$ in all positions except for one position. If no such row exists, one arbitrarily selected 0 in $\overline{\mathbf{p}}$ is declared insignificant (replaced by $*$). Otherwise the largest subset of unallocated rows is selected, all of which match the same pattern corresponding to $\overline{\mathbf{p}}$ with one 0 ignored, and these rows are inserted in an arbitrary order below the upper part of the solution; pattern $\overline{\mathbf{p}}$ is modified accordingly by replacing one significant 0 by $*$.

The similar (symmetric) procedure is then applied to the bottom part of the solution which is characterised by the pattern $\underline{\mathbf{p}}$. As a result, either the lower part remains unchanged and pattern $\underline{\mathbf{p}}$ is modified by replacing one significant 0 with $*$ arbitrarily, or the largest possible set of rows, which match significant 0's of pattern $\underline{\mathbf{p}}$ in all positions except for one position, is added just before the lower part of the solution.

Proceeding in this manner, we decrement by one the number of significant 0's in patterns $\overline{\mathbf{p}}$ and $\underline{\mathbf{p}}$ for both parts of the solution at each iteration. The algorithm stops when all rows of the solution matrix S are filled in with rows of matrix A. A formal description of the algorithm is presented in Appendix C; its time complexity is $O(mn^3)$.

Note that there may be ties related to the choice of rows for insertion into the upper and the lower parts of the solution matrix. Our experiments show that such a choice may have an impact on the quality of a solution to the problem. For that reason, we run heuristic *MAZE* multiple times, breaking ties arbitrarily, in order to obtain a range of solutions.

4.3 Generalisation for the Weighted Case

Both of the heuristics described above can be generalised to handle the gap-zero problem with row weights, *WGap0*. The only modification needed for heuristic *GRIN* is related to calculating the contribution of allocating a target row to a selected position in the current partial solution. As far as heuristic *MAZE* is concerned, two additional modifications are needed. The first one relates to the initial allocation of the rows to the upper and the lower parts of the constructed solution matrix. The following criterion can be used: select two rows with the largest weighted numbers of zeros. However, such a criterion may lead to a very limited choice of the two initial rows and, therefore, restrict the variety of solutions obtained in different runs of heuristic *MAZE*. To overcome this drawback, a wider choice of initial rows may be allowed and some randomness in their selection may be introduced. The second important modification concerns the selection of the subsets of matching rows. Instead of the largest subset in the original version of heuristic *MAZE*, we can select the subset with the maximum cumulative weight of significant zeros, thus ensuring the maximum

weighted contribution to the end-zero objective function at each iteration.

5 Computational Experiments

In this section, we evaluate the performance of our heuristics *GRIN* and *MAZE*, on a wide range of problem instances, while testing out our optimisation algorithm, *GlobalOpt*, on a set of small instances. For small instances the heuristics are benchmarked against the optimal solution values. As no suitable comparator heuristic is available from the literature for larger instances, we use the heuristic which takes a random permutation of rows of the original matrix, A, referred to as *RAND*. The results are presented for the original minimisation version of the problem with function g representing the total number of gap-zeros.

5.1 Experimental Design

The heuristics were evaluated systematically on one main test set, of small instances, and a subsidiary set of large instances. The first set consists of instances with up to 10 columns, n taking values 4, 5, 6, 7, 8, and 10, to be precise. For a given value of n, there are $2^n - 1$ possible different rows, excluding the all-zero-row. Therefore, in order to maintain consistency for different values of n, instances are classified by the number of possible rows, m, as a proportion of the maximum possible number of rows, i.e., $m = \lceil \alpha(2^n - 1) \rceil$, $0 < \alpha \leq 1$. For each value of n, we select $\alpha = 0.1$, 0.3, 0.5, 0.7, 0.9, 1, corresponding to 10%, 30%, 50%, 70%, 90% and 100% of all possible rows, respectively. Note that we do not consider $\alpha = 0.1$ for $n = 4$ since it corresponds to trivial instances with only 2 rows. For all combinations of parameters n and α, we randomly generate 50 problem instances. Note that when $\alpha = 1$ each instance contains the same (whole) set of possible rows, and the instances differ from each other only in their row order, for each value of n. The second test set consisted of five much larger instances, with 25 columns and 20000 rows. The rows of a matrix, are generated by randomly selecting a decimal number from the range $[1, 2^n - 1]$ and then convert it into the corresponding binary string of length n.

Our exact algorithm *GlobalOpt* was executed for the full range of instances with $n = 4, 5, 6$, and for a selection of instances with $n = 7$ and 8. For $n = 7$, algorithm *GlobalOpt* is applied to all 50 instances with $\alpha = 0.5$ and to only a single, randomly chosen, instance for each other value of α. For $n = 8$, algorithm *GlobalOpt* takes too long to be practical (over 6 days) and only a single run for $\alpha = 0.5$ has been performed. The test set of larger instances could certainly not be evaluated with *GlobalOpt*. Recall that both of our heuristics have a stochastic element: *GRIN* in the order for row insertion and tie breaking, and *MAZE* in breaking ties at Step 2. We, therefore, opt to run our heuristics 50 times for each problem instance, using different random seeds.

For a better evaluation of results of our heuristics, we also run a simple heuristic *RAND* in which a solution is generated as a random permutation of rows of the original matrix. Similar to *GRIN* and *MAZE*, heuristic *RAND* is executed 50 times for each problem instance, i.e., we evaluate 50 randomly chosen row permutations. In total, we perform 2500 runs of each heuristic for every combination of n and α. All algorithms have been programmed in the

C++ language and tested on a PC with AMD XP-M 2.2GHz CPU and 1024 MB RAM running under Microsoft Windows XP.

5.2 Comparison between Heuristics

First, we compare the performance of our two heuristics, *GRIN* and *MAZE*, against heuristic *RAND* which constructs random permutations. The results for instances with 6 columns are summarised in Fig. 2, which shows average percentages of gap-zeros in heuristic solutions relative to the total number of 0's in problem instances. The averages are taken over 50 problem instances generated for each value of parameter α. The graphs for other values of n are quite similar and we therefore do not present them here. The results demonstrate a significant superiority of solutions constructed by algorithms *GRIN* and *MAZE* over those obtained by random permutations of rows of the matrices, with *RAND* performing, on average, over 50% worse in all cases. This observation is confirmed by the results for the range of n values but fixed $\alpha = 0.5$ shown in Fig. 3. We compared the quality of the solutions found to random permutations produced by heuristic *RAND*. On average, both heuristics *GRIN* and *MAZE* outperformed *RAND* with approximately 14% improvement in the value of the gap-zero function g.

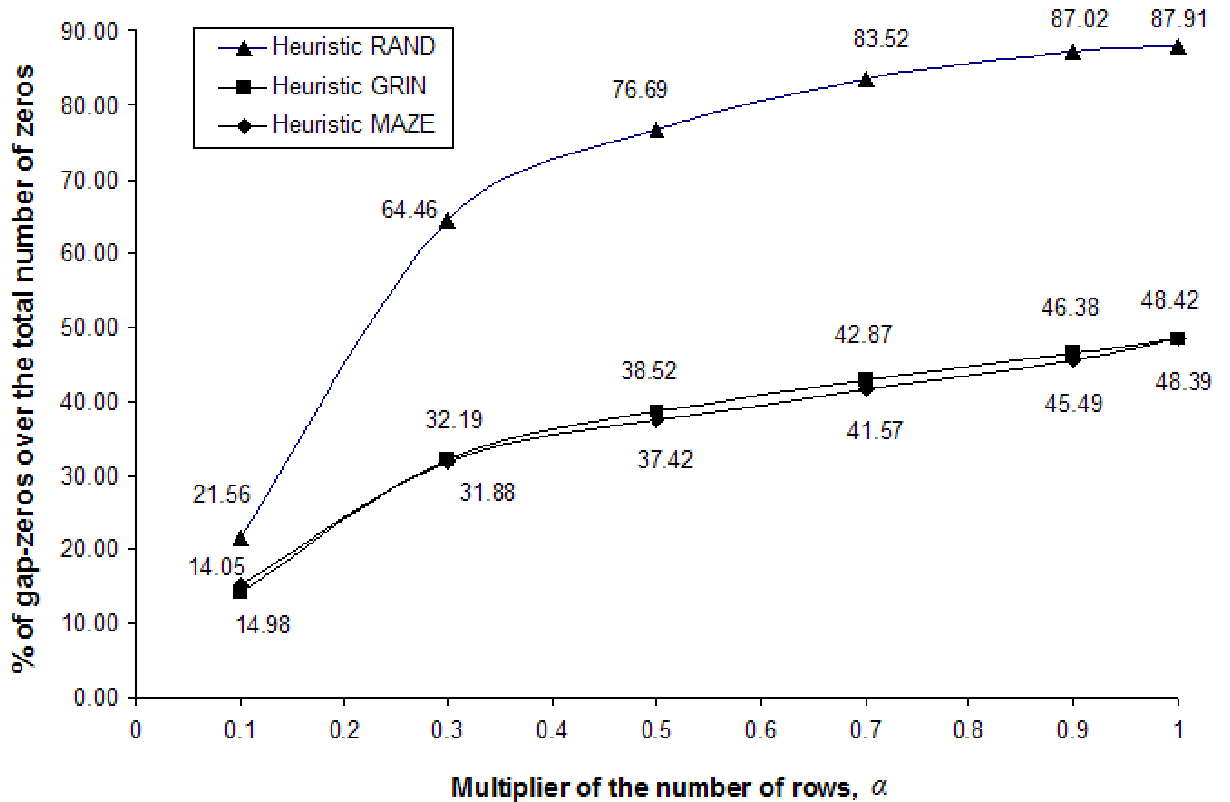

Figure 2: Average proportion of gap-zeros over 50 runs of each heuristic for matrices with 6 columns

Another observation is that heuristic *MAZE* outperforms heuristic *GRIN* for all values of α except for $\alpha = 0.1$. A possible explanation of the advantage of *GRIN* over *MAZE* for small instances, is that rerunning heuristic *MAZE* for such instances gives very little variation in solutions, as the choice of the initial pair of top and bottom rows may be highly restricted due to the limited number of rows with the largest number of 0's. On the other hand, it seems that 50 runs of heuristic *GRIN* cover most of the possible options for insertion of each row, if the number of rows is small. It is therefore better to run *GRIN* as an alternative to *MAZE* for small instances. However, *MAZE* could be modified to search the solution space more widely, by introducing some randomness in the initial selection of top and bottom rows.

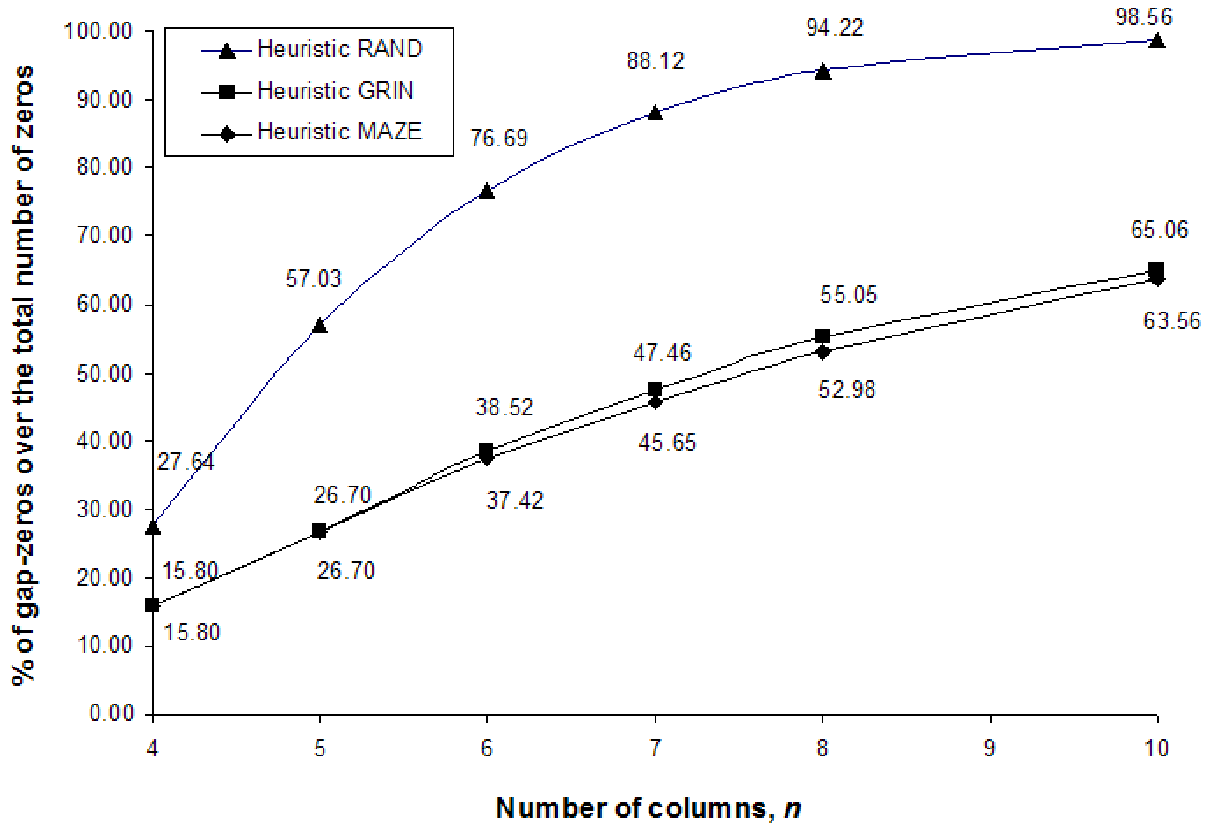

Figure 3: Average proportion of gap-zeros over 50 runs of each heuristic for matrices with different number of columns and $\alpha = 0.5$

Further evidence of the stronger performance of heuristic *MAZE* over *GRIN* for large problems is demonstrated for $\alpha = 0.5$ and a range of values for n in Fig 3. Both heuristics produce similar results for smaller values of n and heuristic *MAZE* outperforms *GRIN* for $n \geq 6$, although by only a small margin. Moreover, for instances with $n = 8$ and 10, heuristic *MAZE* outperforms heuristic *GRIN* for every value of α, including $\alpha = 0.1$.

The experiments conducted on large instances also confirmed the advantage of heuristic

MAZE over *GRIN*: the former heuristic produced better results than the latter one for 4 instances out of 5, with the difference in values of the gap-zero function g ranging from 300 to 3000, and averaging 1070, or 0.5% of the g function value. Moreover, computation time of *MAZE* remained quite small (37 sec for 50 runs on average), while *GRIN* was significantly slower (about 1.5 hours for 50 runs).

5.3 Comparison to an Optimal Solution

Having identified the superiority of heuristic *MAZE* over *GRIN* on many occasions, we now discuss the quality of heuristic solutions by comparing their results to optimal values obtained by algorithm *GlobalOpt*. For instances with 6 columns and various numbers of rows, the number of optimal solutions (out of 50) found by each heuristic, is shown in column N_{opt} in Table 1. A complementary measure of the solutions quality is the proximity of heuristic solutions to the optimal ones, in those cases when heuristics fail to hit the optimum, calculated as the difference between the number of gap-zeros of a heuristic solution and that of an optimum, $g^H - g^*$, and denoted by $D_{(H-opt)}$.

	N_{opt}		$D_{(H-opt)}$	
α	GRIN	MAZE	GRIN	MAZE
0.1	50	42	0	1.25
0.3	30	39	1.1	1.18
0.5	9	45	1.39	1
0.7	4	49	1.93	1
0.9	3	50	1.6	0
1	47	50	1	0

Table 1: Comparative performance of heuristics *GRIN* (G) and *MAZE* (M) to optimal, for instances with $n = 6$

Analysing the figures for N_{opt} and $D_{(H-opt)}$, we observe that heuristic *MAZE* constructs optimal solutions for the majority of problem instances for all values of α and comes very close to the optimum for the remaining instances, with the average deviation from the optimum of around only one gap-zero. Heuristic *GRIN* works particularly well for $\alpha \leq 0.3$ and $\alpha = 1$, but rarely finds optimal solutions for instances with $\alpha = 0.5, 0.7$, and 0.9. Note that for easier problems with $n = 4$ and 5 both heuristics often produce more optimal solutions than for $n = 6$.

We now briefly discuss results of our algorithms for $n = 7, 8$ and 10. For $n = 7$, $\alpha = 0.5$ 50 instances were taken. Heuristic *MAZE* constructed optimal solutions for 43 instances out of 50 with the average deviation from the optimum $D_{(H-opt)}$ of just 1.7 gap-zeros. For for each other values of α, a single instance generated *MAZE* constructed optimal solutions in each case. By contrast, heuristic *GRIN* was able to solve to optimality only a few instances corresponding to $\alpha = 0.1$ and $\alpha = 1$, with $D_{(H-opt)} = 5.0$ for $\alpha = 0.5$. For $n = 8$, only a single instance with $\alpha = 0.5$ was solved by algorithm *GlobalOpt*, as it took 6 days. Heuristic

MAZE constructed an optimal solution for that instance while solution of *GRIN* contained 15 gap-zeros more than the optimal one. The advantage of heuristic *MAZE* over *GRIN* for $n = 8$ and 10 is demonstrated in Fig. 3 for $\alpha = 0.5$. Similar results were obtained for other values of α. All these results suggest that we can expect a very strong performance of heuristic *MAZE* for large size problems where our exact algorithm is inefficient.

Since the run time of algorithm *MAZE* is linear in the number of rows m (its time complexity is $O(mn^3)$) and the number of columns is relatively small in our experiments, it is very fast for all problem instances discussed above. The total CPU time required for 50 runs of *MAZE* is always under a second. Heuristic *GRIN* works slower as the number of rows grows (its running time is of the order $O(m^2n)$). It takes less than 1 second to perform all 50 runs of *GRIN* for all problem instances with $n \leq 6$ and 4, 37 and 332 seconds on average for largest instances ($\alpha = 1$) with $n = 7, 8$, and 10, respectively. Finally, algorithm *GlobalOpt* requires 0.2, 4, and 120 seconds for problems with 4, 5, and 6 columns, respectively, and nearly 2 hours in case of 7 columns. It becomes impractical for $n = 8$ taking around 6 days to find an optimal solution to a single problem.

Finally, we tested our heuristics on larger instances, for which optimal solutions were known in advance. We took matrices with $n = 25, 50, 100$ and 500 columns and $m = 2n - 1$ rows without duplication with the C1P (Consecutive Ones Property), since the optimal value is then known in advance. The value $2n-1$ corresponds to the maximum number of different rows in a C1P matrix, a greater number leads to a repetition of rows. Each column of such a matrix contains exactly n ones and $n - 1$ zeros. As before, 50 runs of each heuristic were performed. The heuristic *RAND* consistently produced very poor solutions in which 99% of zeros appeared as gap-zeros. We found that heuristic *GRIN* achieved optimum in its best runs for each instance. For example, for the instances with $n = 500$ and $m = 999$, 36 out of 50 runs resulted in optimal solutions without any gap-zeros. Heuristic *MAZE* was even more successful constructing an optimal solution in every single run for all four instances.

6 Conclusions

In this paper we have developed three algorithms for tackling the NP-hard problem of minimising the total number of gap-zeros in columns of a binary matrix by permuting its rows. We have proposed algorithms which between them provide optimal, or close to optimal solutions for instances of all sizes: an exact algorithm for smaller instances with no more than 7 columns, and a heuristic, *MAZE*, for larger instances. These approaches accommodate instances expressed in terms of weights on the rows of the binary matrix.

Our exact algorithm is based on indirect enumeration of the solution space by means of classes of solutions with a common structure. The algorithm can be efficiently applied to matrices with a small number of columns, even when the number of rows is quite large. In order to tackle larger problems, we have developed two heuristic approaches. One, *GRIN*, is based on the greedy insertion of randomly pre-sorted rows of the original matrix into a partial solution. The other, *MAZE*, constructs a solution by allocating matching rows to the upper and the lower parts of the solution matrix. Both heuristics involve multiple runs. Extensive testing was carried out on instances with a relatively small number of columns ($n \leq 10$), and

on large instances. The first heuristic, *GRIN*, worked particularly well for smaller problems, while the second one was able to produce optimal or near-optimal outcomes for a wider range of generated problems. The good performance of the first, multiple run, heuristic, *GRIN*, contrasts starkly with the known result that no performance bound exists for a single run of any such greedy heuristic employing any given type of pre-ordering of row inputs with on-line row selection [22]. Moreover, the second heuristic, *MAZE*, proved to be particularly effective and solves all of the test instances with known optimal value, to optimality, including very large instances. In all cases where the optimal value is known, both of our heuristics produced solutions within 3% of optimal.

An additional contribution of the article is a deeper understanding of the structure of the *Gap0* problem. A new representation of the solution space, in terms of the ranking of the lengths of columns of end-zeros, is offered, and used successfully. We also highlight the phenomenon of overlapping end-zeros and associated complications. An open challenge is that of estimating the optimality gap related to ignoring overlapping end-zeros. Some evidence to support the restriction to non-overlapping zeros, is given by the computational investigations, reported in the Appendix, which indicate that instances with overlapping end-zeros become rare for larger sizes of problem.

Many real world problems may be abstracted as a *Gap0* problem. In particular, we have successfully applied the approach developed in this paper in the context of microbiological food testing. The methodology is used to resequence food samples, and is combined with an optimisation approach for scheduling agar production presented in [7], to take account of the limited shelf life of agar. This approach provides a methodology for increasing efficiency of food testing in the laboratory by 20%.

Acknowledgements

This research was supported by EPSRC grant number EP/D079713/1 with Premier Analytical Services (UK).

References

[1] F. Alizadeh, R. M. Karp, L. A. Newberg, and D. K. Weisser, *Physical mapping of chromosomes: a combinatorial problem in molecular biology*, Algorithmica **13** (1995), 52–76.

[2] F. Alizadeh, R. M. Karp, D. K. Weisser, and G. Zweig, *Physical mapping of chromosomes using unique probes*, Journal of Computational Biology **2** (1995), 159–184.

[3] J. E. Atkins, E. G. Boman, and B. Hendrickson, *A spectral algorithm for seriation and the consecutive ones problem*, SIAM Journal on Computing **28** (1998), 297–310.

[4] J. E. Atkins and M. Middendorf, *On physical mapping and the consecutive ones property for sparse matrices*, Discrete Applied Mathematics **71** (1996), 23–40.

[5] K. S. Booth, *PQ-tree algorithms*, Ph.D. thesis, University of California, Berkeley, US, 1975.

[6] K. S. Booth and G. S. Lueker, *Test for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms*, Journal of Computer and System Sciences **13** (1976), 335–379.

[7] K. Chakhlevitch, C. A. Glass, and H. Kellerer, *Batch machine production with perishability time windows and limited batch size*, European Journal of Operational Research **210** (2011), 39–47.

[8] K. Chakhlevitch, C. A. Glass, and P. A. Sadd, *Applying efficient logistics in a microbiology laboratory*, Journal of Food Engineering **103** (2011), 377–387.

[9] M. Dom, *Algorithmic aspects of the Consecutive-Ones Property*, EATCS Bulletin **98** (2008), 27–59.

[10] M. Dom, J. Guo, and R. Niedermeier, *Approximation and fixed-parameter algorithms for consecutive ones submatrix problems*, Journal of Computer and System Sciences **76** (2010), 204–221.

[11] D. R. Fulkerson and O. A. Gross, *Incidence matrices and interval graphs*, Pacific Journal of Mathematics **15** (1965), 835–855.

[12] M. R. Garey and D. S. Johnson, *Computers and intractability: A guide to the theory of np-completeness*, W. H. Freeman, San Francisco, 1979.

[13] S. P. Ghosh, *File organization: the consecutive retrieval property*, Communications of ASM **15** (1972), 802–808.

[14] D. S. Greenberg and S. Istrail, *Physical mapping by STS hybridization: algorithmic strategies and the challenge of software evaluation*, Journal of Computational Biology **2** (1995), 219–273.

[15] M. T. Haibi, R. McConnell, C. Paul, and L. Viennot, *Lex-BFS and partition refinement, with applications to the transitive orientation, interval graph recognition and consecutive ones testing*, Theoretical Computer Science **234** (2000), 59–84.

[16] M. T. Hajiaghayi and Y. Ganjali, *A note on the consecutive ones submatrix problem*, Information Processing Letters **83** (2002), 163–166.

[17] W.-L. Hsu, *A simple test for the consecutive ones property*, Journal of Algorithms **43** (2002), 1–16.

[18] D. G. Kendall, *Incidence matrices, interval graphs and seriation in archaeology*, Pacific Journal of Mathematics **28** (1969), 565–570.

[19] L. T. Kou, *Polynomial complete consecutive information retrieval problems*, SIAM Journal on Computing **6** (1977), 67–75.

[20] M. Oswald, *Weighted consecutive one problem*, Ph.D. thesis, University of Heidelberg, Germany, 2003.

[21] J. Tan and L. Zhang, *The consecutive ones submatrix problem for sparse matrices*, Algorithmica **48** (2007), 287–299.

[22] M. Veldhorst, *Approximation of the consecutive ones matrix augmentation problem*, SIAM Journal on Computing **14** (1985), 709–729.

Appendix A: Complexity Analysis of Algorithm *ClassOpt*

Overlapping Instances

The time complexity of algorithm $ClassOpt(A, \pi, \sigma)$ is defined by Step 2. The number of patterns which characterise a class $\mathcal{C}(\pi, \sigma)$ is $2n-1$: there are $n-1$ patterns with significant top-zeros, $n-1$ patterns with significant bottom-zeros and one additional middle pattern $(**\ldots**)$ without significant 0's. There are up to $n-1$ zero-entries in each row which should be checked against significant 0's of each pattern in order to establish a match.

In order to implement Step 1 efficiently, we consider the entries of the current row of A in reverse order of π noting when the first 1-entry is identified. This provides the maximum number of significant 0's if the current row is allocated to the top patterns of $\mathcal{P}_{\pi,\sigma}$. In a similar way we calculate the maximum number of significant 0's if the current row is allocated to the bottom patterns of $\mathcal{P}_{\pi,\sigma}$. The best pattern out of the two is selected as the output of Step 2.1, so that the time complexity of this step is $O(n)$. Thus finding the best matches for all m rows of matrix A incurs $O(mn)$ time.

Non-overlapping Instances

First we perform pre-processing by constructing patterns which characterise class $\mathcal{C}(\rho)$, counting the number of significant zeros for each pattern. The number of patterns is $2n-1$ since there are $n-1$ zeros in the top pattern and each subsequent pattern differs from the previous one by one 0 removed or one 0 added until the bottom row with $n-1$ zeros is reached. Thus the pre-processing can be done in $O(mn)$ time.

Suppose the row-patterns in class $\mathcal{C}(\rho)$, which contain all top-zeros, have numbers 1, 2, $\ldots$, t, and those which contain all bottom-zeros have numbers b, $b+1$, $\ldots$, $2n-1$, where $1 < b < t < 2n-1$ for overlapping instances. The matching patterns can be identified in the same way as in the case of non-overlapping instances; one just needs to introduce permutations π and σ associated with ρ to characterise top-zeros and bottom-zeros, respectively. The following three cases may occur.

- Both sets of matching top-zeros and bottom-zeros are empty: then no matching patterns exist and there is no solution in the class $\mathcal{C}(\rho)$.
- One of the sets is empty and the other one is non-empty: then the row with the maximum number of significant zeros is selected in the same way as for non-overlapping instances.
- Both sets are non-empty and they are given by row-patterns t',$t'+1$, $\ldots$, t, $1 \le t' \le t$ for top-zeros and patterns b,$b+1$, $\ldots$, b', $b \le b' \le 2n-1$ for bottom-zeros: if $b' < t'$, then no matching patterns exist; otherwise select a row-pattern from the range t', $t'+1$, $\ldots$, b' which has the maximum number of significant zeros.

Thus Step 2.1 of algorithm *ClassOpt* can be implemented in $O(n)$ time. Therefore finding the best matches for all m rows of matrix A incurs $O(mn)$ time in Step 2. We conclude that

the time complexity of algorithm *ClassOpt* is the same as in the non-overlapping case, i.e., $O(mn)$.

Appendix B: Formal Description of Heuristic *GRIN* and its Analysis

Algorithm GRIN (for GReedy INsertion)

```
Input:  rows of matrix A in a pre-specified order
Output: solution matrix S and objective value f(S)
```

1. `Insert the first row of` A `into a new empty solution` S
2. `REPEAT until all rows of` A `are inserted into partial solution` S
   - 2.1. `Select the next row of matrix` A `respecting the given ordering`
   - 2.2. `For each position in the partial solution calculate the number of end-zeros after each insertion of the target row`
   - 2.3. `Insert the target row into a position in the partial solution` S `which corresponds to the option with the maximum value of the objective function` f `among those considered in Step 2.2. Break ties by applying Rules 1-4 described in the Appendix`

   `END`
3. `Return the constructed solution matrix` S `and objective value` $f(S)$

If there are several equally "good" options for target row insertion in Step 2.3 of algorithm *GRIN*, we use the following rules, which provide consistency, in breaking ties. Note that the rules with the choice of options opposite to those used in Rules 1, 2, and 4 can also be applied, which leads to the reverse sequence of rows in the resulting solution.

Rule 1. In case of a choice between inserting the target row immediately before or immediately after some selected row in the partial solution, put the target row before the selected one if the former contains more zero-entries than the latter, otherwise insert it after the selected row.

Rule 2. In case of a choice between inserting the target row into the very first or into the very last position in the partial solution, make the target row first if it contains more zero-entries than the current first row in the solution, and make the target row last otherwise.

Rule 3. In case of a choice between inserting the target row in between two rows of the partial solution or into the very first (into the very last) position, always select the former option.

Rule 4. In case of more than two possible options for inserting the target row, first apply Rule 3 to eliminate insertions at the ends of the partial solution (if such options are present). Then apply Rule 1 (if there are appropriate pairs of options) to further restrict the choice. If there are still multiple positions for insertion, place the target row into the lowest position among them, i.e. into the position which is the closest to the beginning of the partial solution.

The selection rules in *GRIN*, are designed to imitate the structure of no-gap solutions for matrices which satisfy the consecutive ones property. In such solutions the rows with greater number of 1's are usually concentrated in the middle of the solution matrix while the rows mostly consisting of 0 entries are distributed towards the end rows of the matrix.

We analyse the time complexity of heuristic *GRIN*. There are $O(m)$ repetitions of Step 2. For each target row, there are $O(m)$ different options for insertion. The straightforward evaluation of $f(S)$ for each option, which is done in Step 2.2, requires $O(mn)$ time: one can insert the target row in S temporarily and count the number of end-zeros in the resulting matrix. A more efficient approach is to maintain the value $f(S)$ for a current partial solution S together with additional markers for each element of S which indicate whether an element is a top-zero, a bottom-zero or an insignificant element. Then the evaluation of $f(S)$, after a possible insertion of the target row, can be implemented in $O(n)$ time by comparing the target row with its two neighbours. Finally, the actual insertion of the target row in S in Step 2.3 requires $O(mn)$ time. Therefore, with efficient implementation of Step 2.2, the inner loop consisting of Steps 2.1-2.3 can be implemented in $O(mn)$ time, leading to the $O(m^2n)$ time complexity of heuristic *GRIN*.

Appendix C: Formal Description of Heuristic MAZE and its Analysis

Algorithm MAZE

`Input:` rows of matrix A

`Output:` solution matrix S and objective value $f(S)$

1. Initialise S by allocating two rows of matrix A with the maximum number of 0's to positions $i' \leftarrow 1$, $i'' \leftarrow m$ of S; break ties arbitrarily

   For row i', define pattern $\overline{\mathbf{p}}$ of significant 0's and calculate $\overline{z}$, the number of its 0's

   For row i'', define pattern $\underline{\mathbf{p}}$ of significant 0's and calculate $\underline{z}$, the number of its 0's

   Assign unallocated rows to the set R

2. REPEAT until $R = \emptyset$

   2.1 Consider the upper part of S

   FOR $k = 1, \ldots, \overline{z}$

   Construct pattern $\overline{\mathbf{p}}_k$ by replacing the kth 0 in $\overline{\mathbf{p}}$ by $*$, and the set of rows $R_k \subseteq R$ which match pattern $\overline{\mathbf{p}}_k$

   ENDFOR

   IF $R_1 = \cdots = R_{\overline{z}} = \emptyset$

   Replace pattern $\overline{\mathbf{p}}$ by a pattern $\overline{\mathbf{p}}_k$ with one significant 0 less (k can be selected arbitrarily)

   $\overline{z} \leftarrow \overline{z} - 1$

   ELSE

   Select the set R_ℓ corresponding to the largest number of matching rows; break ties arbitrarily

   Allocate all rows from R_ℓ after row i' in S

   Update the set of unallocated rows: $R \leftarrow R \backslash R_\ell$

   Set $i' \leftarrow i' + |R_\ell|$, $\overline{\mathbf{p}} \leftarrow \overline{\mathbf{p}}_\ell$ and $\overline{z} \leftarrow \overline{z} - 1$

   2.2 Repeat the actions similar to those of step 2.1 for the bottom part of S, updating R, i'', $\underline{\mathbf{p}}$ and $\underline{z}$ accordingly

   END

3. Return the constructed solution matrix S and its objective value $f(S)$

We analyse the time complexity of Algorithm *MAZE*. It is determined by Step 2. Step 2.1 is repeated $O(n)$ times since pattern $\overline{\mathbf{p}}$ contains no more than $n-1$ significant 0s, and at each iteration one significant 0 is replaced by $*$. Similarly, Step 2.2 is repeated $O(n)$ times. At each iteration, $O(m)$ rows from R are checked for matching against each of the patterns $\overline{\mathbf{p}}_1, \ldots, \overline{\mathbf{p}}_z$, $z < n$, in order to form subsets R_k, and each matching test requires $O(n)$ comparisons of individual entries. Assigning selected rows R_l and updating variables takes $O(mn)$ time. Therefore, each iteration of Step 2 requires $O(mn^2)$ time and the overall time complexity of algorithm *MAZE* is $O(mn^3)$.