Extracting Pure Network Submatrices in Linear Programs Using Signed Graphs

N. GÜLPINAR[*] G. GUTIN[†] G. MITRA[‡] A. ZVEROVITCH[§]

Abstract

It is shown that the problem of detecting a maximum embedded network in a linear program is related to balancing of subgraphs of signed graphs. This approach leads to a simple efficient heuristic to extract an embedded network. The proposed heuristic also determines whether a given linear program is a (reflected) network itself. Some complexity results are obtained and computational results are also reported.

1 Introduction, terminology and notation

Large-scale linear programming (LP) models which arise in applications usually have sparse coefficient matrices with special structure. If a special structure can be recognized, it can often be used to considerably speed up the process of solving the LP problem. A well-known family of such special structures is networks; a number of heuristics to extract (reflected) networks in LP problems have been developed and analyzed, see, e.g., [1, 3, 4, 6, 9, 15] (a formal definition of a reflected network is given below). From the computational point of view, it is worthwhile extracting a reflected network only if the LP problem under consideration contains a relatively large reflected network.

In this paper, we consider a new approach based on application of signed graphs to detect reflected networks. We show that the problem of detecting a reflected network structure in an LP problem can be transformed into the problem of finding an independent set in some signed graph. This approach leads to a simple algorithm to find reflected networks embedded within LP problems that, unlike most known methods utilizing only

[*]Department of Computing, Imperial College of Science, Technology and Medicine, University of London, 180 Queen's Gate, London, SW7 2BZ, UK

[†]Department of Computer Science, Royal Holloway, University of London, Egham, Surrey TW20 0EX, UK

[‡]Department of Mathematical Sciences, Brunel University, Uxbridge, Middlesex, UB8 3PH, UK

[§]Department of Computer Science, Royal Holloway, University of London, Egham, Surrey TW20 0EX, UK

row reflection, determines whether a given LP problem is a reflected network problem. (We are aware only of one other method having this important property: the method in [15].) Our computational experiments show that our algorithm performs well for many practical LP problems, which contain relatively large embedded reflected networks. The algorithm is time efficient which is of importance as the extraction of a reflected network is only part of the whole costly process of solving a large-scale LP problem. Our computational experiments show that the new method is a promising approach that deserves further computational investigation of itself as well as its modifications.

We consider an LP problem in the standard form stated as

$$\text{Minimize } \{p^T x;\ \text{subject to } Ax = b,\ x \geq 0\}.$$

LP problems have a number of equivalent, in a sense, forms that can be obtained from each other by various operations. Often scaling operations, that is multiplications of rows and columns of the matrix A of constraints by non-zero constants, are applied, see, e.g., [1, 3, 6, 9]. In the sequel, we assume that certain scaling operations on A have been carried out and will not be applied again apart from row reflections defined below.

A matrix B is a *network (matrix)* if B is a $(0, \pm 1)$-*matrix* (that is, entries of B belong to the set $\{1, 0, -1\}$) and every column of B has at most one entry equal to 1 and at most one entry equal to -1. The operation of *reflection* of a row of a matrix B changes the signs of all non-zero entries of this row. A matrix B is a *reflected network (matrix)* if there is a sequence of row reflections that transforms B into a network matrix. The *problem of detecting a maximum embedded reflected network* (DMERN) is to find the maximum number of rows that form a submatrix B of A such that B is a reflected network. This number is denoted by $\nu(A)$. The DMERN problem is known to be NP-hard [2].

In the rest of this paper, we assume, for simplicity, that A is a $(0, \pm 1)$-matrix itself (since all rows containing entries not from the set $\{-1, 0, +1\}$ cannot be part of a reflected network). For a graph $G = (V, E)$ and $W \subseteq V$, $G[W]$ stands for the subgraph of G induced by W; $\overline{W} = V \backslash W$.

In this paper, graphs are allowed to have parallel edges, but no loops. A graph $G = (V, E)$ along with a function $s:\ E \rightarrow \{-, +\}$ is called a *signed graph.* We assume that signed graphs have no parallel edges of the same sign. An edge is *positive* (*negative*) if it is assigned plus (minus). Let $G = (V, E, s)$ be a signed graph. For a non-empty subset W of V, the W-*switch* of G is the signed graph G^W obtained from G by changing the signs of the edges between W and $\overline{W}$. For a subgraph H of G, $\mu(H)$ is the number of negative edges of H.

Signed graphs have been studied by many researchers, see, e.g., [11, 12, 13, 14, 22, 23]; some of these papers implicitly considered connections between signed graphs and matrices, see, e.g., [14]; a bibliography and glossary on signed graphs are provided in [20, 21].

2 Embedded networks and signed graphs

For a $(0,\pm 1)$-matrix $A=[a_{ik}]$ with n rows, we construct a signed graph $G(A)$ as follows: the vertex set of $G(A)$ is $\{1,2,...,n\}$; $G(A)$ has a positive (negative) edge ij if and only if $a_{ik}=-a_{jk}\neq 0$ ($a_{ik}=a_{jk}\neq 0$) for some k. $G(A)$ may have parallel edges of opposite signs. For a signed graph $G=(V,E,s)$, G^- (G^+) denotes the subgraph of G whose vertices are the same as in G and whose edges are the negative (positive) edges in G. A set I of vertices of a graph H is *independent* if no two vertices in I are adjacent; $\alpha(H)$ denotes the cardinality of a maximum independent set of vertices in H, the *independence number* of H.

The importance of the following parameter can be seen from Theorem 2.1. For a signed graph $G=(V,E,s)$, define

$$\tilde{\alpha}(G)=\max\{\alpha((G^W)^-):\ W\subseteq V\}.$$

Theorem 2.1. *For a $(0,\pm 1)$-matrix A, we have $\nu(A)=\tilde{\alpha}(G(A))$.*

This theorem follows from the next two lemmas. A signed graph $G=(V,E,s)$ is *balanced* if there exists a subset W of V (W may coincide with V) such that all edges in the subgraphs of G induced by W and $\overline{W}$ are positive and all edges between W and $\overline{W}$ are negative; observe that, in this case, $\mu(G^W)=0$. Let $\eta(G)$ be the largest order of a balanced induced subgraph of G.

Lemma 2.2. *For a signed graph $G=(V,E,s)$, $\eta(G)=\tilde{\alpha}(G)$.*

Proof: Let $H=(U,F)$ be a balanced induced subgraph of G of maximum order, i.e. $|U|=\eta(G)$. Clearly,

$$\eta(G)=|U|\leq\max\{\alpha((G^W)^-):\ W\subseteq V\}=\tilde{\alpha}(G).$$

Let W' be a subset of V and let N be a maximum independent set in $(G^{W'})^-$ such that $|N|=\tilde{\alpha}(G)$. Clearly $G[N]$ is balanced, and, thus, $\eta(G)\geq\tilde{\alpha}(G)$. $\square$

Lemma 2.3. *For a $(0,\pm 1)$-matrix A, we have $\nu(A)=\eta(G(A))$.*

Proof: Let A be an $n\times m$ matrix and let B be a reflected network submatrix of A formed by the maximum number of rows, thus B is a $\nu(A)\times m$ matrix. Let W be the indices of rows in B whose reflection transforms B into a network matrix. Observe that $\mu(G(B)^W)=0$. Hence, $G(B)$ is a balanced induced subgraph of $G(A)$, and, thus, $\nu(A)\leq\eta(G(A))$.

Let $G=G(A)$ and let $G[X]$ be a balanced induced subgraph of G of order $\eta(G)$. This means that there exists $W\subseteq X$ such that $\mu(G[X]^W)=0$. Observe that, if we apply reflection to the rows with indices W, then the submatrix of A whose rows have indices from X transforms into a network matrix. Hence, $\nu(A) \geq \eta(G(A))$. $\square$

Theorem 2.1 allows us to study $\tilde{\alpha}(G)$ for signed graphs G rather than $\nu(A)$ for $(0, \pm 1)$-matrices A. However, we cannot restrict ourselves to any special class of signed graphs due to the following fact:

Theorem 2.4. *For a signed graph H, there exists a $(0, \pm 1)$-matrix A such that $G(A)$ is isomorphic to H.*

Proof: Let $\{1, 2, ..., n\}$ and $\{e_1, ..., e_m\}$ be the vertex set and the edge set, respectively, of H. Construct an $n \times 2m$-matrix A as follows. For an index $j \in \{1, ..., m\}$: if $e_j = ik$ has no parallel edge and is positive (negative), then columns $2j-1$ and $2j$ consist of zero entries apart from $a_{i,2j} = 1$, $a_{k,2j} = -1$ ($a_{i,2j} = a_{k,2j} = 1$); if $e_j = ik$ has a parallel edge, then columns $2j-1$ and $2j$ consist of zero entries apart from $a_{i,2j} = 1$, $a_{k,2j} = -1$, $a_{i,2j-1} = a_{k,2j-1} = 1$. It is easy to check that $H = G(A)$. $\square$

We give another proof that the DMERN problem is NP-hard. It is well known that the problem of computing the independence number of a graph is NP-hard. The last problem can be polynomially reduced to the DMERN problem as follows. Transform a graph H into a signed graph H' by doubling its edges and assigning opposite signs to parallel edges. Clearly, $\alpha(H) = \tilde{\alpha}(H')$ and it remains to apply Theorems 2.1 and 2.4 to complete the reduction.

Consider signed graphs G with no parallel edges. When all edges in G are positive, the problem of computing $\tilde{\alpha}(G)$ becomes trivial; the case in which all edges are negative is treated in the following theorem.

Theorem 2.5. *The problem of computing $\tilde{\alpha}(G)$ for signed graphs G whose edges are all negative is NP-hard.*

Proof: Let H be a graph and H' be a vertex-disjoint copy of H. To obtain the graph $L = L(H)$ add to $H \cup H'$, the disjoint union of H and H', all edges between H and H'. Assign minus to every edge of L. The theorem follows from the fact that $\tilde{\alpha}(L) = 2\alpha(H)$, which we now prove.

Let W be a set of vertices in G; let $W_0 = W \cap V(H)$, $W_1 = W \cap V(H')$. Observe that no independent set of $(L^W)^-$ can have two vertices from $H[W_0]$ and $H'[W_1]$ or from $H - W_0$ and $H' - W_1$, while any two vertices taken from $G[W]$ and $G - W$ are non-adjacent in $(L^W)^-$. Hence,

$$\alpha((L^W)^-) = \max\{\alpha(H[W_0]), \alpha(H'[W_1])\} + \max\{\alpha(H - W_0), \alpha(H' - W_1)\} \leq 2\alpha(H).$$

Thus, $\tilde{\alpha}(L) \leq 2\alpha(H)$. We have $\tilde{\alpha}(L) \geq 2\alpha(H)$ since $\alpha((L^{V(H)})^-) = 2\alpha(H)$. $\square$

In the construction in the proof of the last theorem, let us add positive edges between the pairs of non-adjacent vertices in L. We have obtained a signed complete graph K.

Since K is a spanning supergraph of L, $\tilde{\alpha}(K) \leq \tilde{\alpha}(L)$. However, $\alpha((K^{V(H)})^-) = 2\alpha(H)$. Thus, $\tilde{\alpha}(K) = 2\alpha(H)$. We have derived the following:

Corollary 2.6. *The problem of computing $\tilde{\alpha}(K)$ for signed complete graphs K is NP-hard.*

Now we formulate characterizations of signed graphs H for which there exist reflected network matrices A such that $H = G(A)$. We call such graphs *network graphs*. By Theorem 2.1, a signed graph H is a network graph if and only if there exists a set W of vertices in H such that $\mu(H^W) = 0$. Clearly, a network graph has no parallel edges. Hence, in the rest of this section we consider only connected graphs with no parallel edges. (Disconnected graphs can be treated by studying their components one by one.)

Lemma 2.7 and Theorem 2.8 are not original results. Still, we provide their proofs for the sake of completeness and to show some simple yet efficient ideas used in the theory of signed graphs; the proof of Lemma 2.7 is useful from an algorithmic point of view and illustrates the standard proof by switching (e.g., implicit in [13]).

Lemma 2.7. *Every signed tree T is a network graph.*

Proof: We prove the lemma by induction on the number of edges in T. The lemma is true when the number of edges is one. Let x be a vertex of T of degree one. By the induction hypothesis, there is a set $W \subseteq V(T) - x$ such that $\mu((T - x)^W) = 0$. In T^W the edge e incident to x is positive or negative. In the first case, let $W' = W$ and the second case, let $W' = W \cup \{x\}$. Then, $\mu(T^{W'}) = 0$. □

The equivalence of (b), (c) and (d) in the following theorem is shown in [22].

Theorem 2.8. *For a connected signed graph G with no parallel edges and a spanning tree T of G, the following assertions are equivalent: (a) G is a network graph; (b) G is a balanced graph; (c) G does not have a cycle with odd number of negative edges; (d) if $W \subseteq V(G)$ is such that $\mu(T^W) = 0$, then $\mu(G^W) = 0$.*

Proof: The equivalence of (a) and (b) is obvious and follows from the definitions of network graphs and balanced signed graphs. By the latter definition, (b) implies (c). By considering the chords of T one by one, we see that (d) follows from (c). Clearly, (d) and Lemma 2.7 imply (a). □

3 Network extraction algorithm

Theorem 2.8 leads to the following algorithm that can be used to extract reflected networks embedded in linear programs with matrix A of constraints. As above, we assume that A is a $(0, \pm 1)$-matrix.

Algorithm (SGA):

Step 1: Construct signed graph $G = G(A) = (V, E, s)$.

Step 2: Find a spanning forest T in G;

Step 3: Using a recursive algorithm based on the proof of Lemma 2.7, compute $W \subseteq V$ such that $\mu(T^W) = 0$.

Step 4: Using some algorithm, find a maximal independent set I in the graph $(G^W)^-$.

The vertices of I correspond to rows of A that form a network.

Clearly, if G is a network graph, our algorithm's output will always be $I = V$. In other cases, the output may well depend on the choice of T.

We implemented Step 1 of the algorithm using the following procedure. By $s(jk) = 0$ we mean that G has double edges between j and k.

Step 1 Expanded
Arguments:
$A = [a_{ji}]$: $n \times m$ $(0, \pm 1)$-matrix of constraints;
n: number of rows in A;
m: number of columns in A;
Result:
The signed graph $G = (V, E, s)$.

begin
  $V := \{1, \ldots, n\}$; $E := \emptyset$

  for $i := 1$ **to** m **do**
    for all pairs (j, k), such that $1 \leq j < k \leq n$ and $a_{ji} \neq 0$ and $a_{ki} \neq 0$ **do**
      if $a_{ji} = a_{ki}$ **then**
        $w := -$
      else
        $w := +$
      end if
      if $jk \notin E$ **then**
        $E := E \cup \{jk\}$; $s(jk) := w$
      else if $s(jk) \neq w$ **then**
        $s(jk) := 0$
      end if
    end for
  end for
end

4 Implementation details

Procedure for the Step 1 of the algorithm that is described in the previous section is based on column-wise scan of A. For every column, all pairs of non-zero elements of the column are considered, and the corresponding edges of the graph are updated accordingly. The complexity of this procedure depends on the complexity of performing the following three operations on the graph: insertion of an edge between two unconnected vertices, checking whether there is an edge connecting two vertices j and k, and adding the second edge between j and k. For efficiency reasons, our implementation of the column-wise scan procedure utilizes a single data structure for representing both the set of graph edges E and the sign function s as used by the algorithm. This data structure is an adjacency matrix. It allows for each of the three required operations to be carried out in constant time. Clearly, the overall time complexity of our implementation of the column-wise scan procedure is therefore $O(mn^2)$. The adjacency matrix requires $O(n^2)$ memory. Each element of the adjacency matrix can be in one of four possible states (no edge, positive/negative edge, double edges), and therefore requires only 2 bits of storage. It is also possible to use the fact that the adjacency matrix is symmetric. In practical terms this means that, for example, an adjacency matrix representing a signed graph for a $(0, \pm 1)$-matrix containing 20,000 constraints can be stored in 50 megabytes, which is well within the reach of modern PCs.

An alternative procedure, based on row-wise scan of the matrix A, can be used in place of the above column-wise scan. This procedure considers every pair of vertices and scans the corresponding pair of rows of A in order to construct the necessary edge(s) between the two vertices. This procedure requires $n^2 m$ comparisons, and at most $2n^2$ edges are added to G. If G is represented by a suitable data structure that allows efficient edge insertions, the overall time complexity of the row-wise scan procedure is $O(n^2 m)$. Since every potential edge is considered only once, no intermediate data structures are necessary in order to allow efficiently finding or modifying existing edges.

For most real-world problems the constraint matrix A is sparse. In that case the performance bounds from the previous two paragraphs may not give a good indication of relative practical performance of efficient sparsity-exploiting implementations of the two algorithms. In practice sparse matrices are usually represented by special data structures that allow efficient enumeration of all non-zero elements of a row or a column in time that is proportional to the number of such elements. Let us assume for simplicity that every column of A has exactly k_c non-zero entries, and that every row of A has exactly k_r non-zeros. Clearly, $nk_r = mk_c = k$, where k is the total number of non-zero elements in A. A distinctive advantage of the column-wise scan procedure is that it is capable of enumerating all pairs (a_{ji}, a_{li}), where $1 < i < m$, $1 < j < l < n$, such that both a_{ji} and a_{li} are non-zero. Column-wise scan can efficiently avoid considering pairs where at least one element is equal to zero (such pairs have no effect on the resulting signed graph). Now let us consider the row-wise scan, which needs to find all columns in which both elements of two given rows are non-zero. In order to perform this, in the worst case a typical implementation of the row-wise scan may need to enumerate all non-zeros in each of the two rows. This can amount for up to $2k_r$ (potentially unnecessary) operations per pair of rows. Thus, for the whole matrix A the number of such operations can be $n(n-1)k_r = (n-1)k$, while the number of edges in the graph (double edges counted as one) can not exceed $\frac{k_c(k_c-1)}{2}m = \frac{1}{2}k(\frac{k}{m}-1)$. We have

$$\frac{\frac{1}{2}k(\frac{k}{m}-1)}{(n-1)k} \leq \frac{k}{2m(n-1)},$$

which is typically within 0.005 for larger Netlib problems (The Netlib problems [10] form an often used testbed for computational experiments on the topic, see the next section.) Thus, the column-wise scan procedure can avoid a large number of unnecessary operations that may need to be performed by the row-wise scan. The price for this is the higher cost of the operations that need to be executed for every pair of non-zero elements as well as the higher memory requirements of the column-wise scan compared to the row-wise scan. We have implemented and tested both procedures. Our estimates show that column-wise scan is generally faster, and our computational tests (which are not presented in this paper) confirm this. All timing results presented in the next section refer to the column-wise scan.

The row-wise scan procedure can be somewhat improved by terminating processing of a pair of rows once double edges between the corresponding vertices have been found. In our experience, however, the number of double edges is very small for Netlib problems. Over 80% of the problems we have tested have no double edges in the corresponding signed graphs, and for all the remaining problems not more than 0.1% of the row pairs have double edges between the corresponding vertices and thus do not require full processing. We therefore regard the impact of this improvement as negligible. We also do not rule out a possibility that a special data structure can be constructed that will make finding pairs of non-zero elements of A by the row-wise scan more efficient. However, we consider this as an implementation issue beyond the scope of this study.

SGA is a fast polynomial algorithm. Using breadth-first search or depth-first search (see [8]), T in Step 2 can be readily computed in time $O(|E|)$ for Step 2. The proof of Lemma 2.7 implies an $O(|E|)$-time recursive algorithm. We implemented Step 4 using the degree-greedy algorithm (see [18]): starting from empty S, append to S a vertex of $H = (G^W)^-$ of minimum degree, delete this vertex together with its neighbors from H, and repeat the above procedure till H has no vertex. This algorithm has complexity of $O(|E|)$ [18].

5 Computational results

We investigated the computational performance of SGA for a number of benchmark problems which are readily available to the scientific community. We chose models from Netlib (the models can be downloaded from `http://www.netlib.org/lp/data/`) and carried out computational experiments for most of the models. In total, computational performance was measured on 100 Netlib models of various sizes.

The characteristics of the test problems in terms of the number of constraints, variables and non-zero entries are summarized in Table 1. Detailed computational results are also presented in Table 1. Due to limited space, the table only contains detailed results for a representative sample of 44 models which have at least 500 rows.

For each of the test models, we first applied preprocessing as in [5] to reduce the size of the coefficient matrix and then a scaling procedure to increase the number of rows consisting of only $+1, 0$ and -1 entries. The number of rows in the resulting $(0, \pm 1)$-matrix is displayed in Table 1 under the heading TMROW. Several (reflected) network detection algorithms were then applied to this matrix.

In order to assess the performance of SGA, we compared SGA with three other algorithms. One of these algorithms, Row-Scanning Deletion (RSD), was introduced by Brown et al. (see [3], [6]) and the other two algorithms were published in [9].

The RSD algorithm works by performing a series of row operations that eliminate row conflicts in the $(0, \pm 1)$-matrix. Two rows are said to be in *conflict* if they both have a $+1$ or both have a -1 in the same column. A network matrix has no conflicts. In RSD, *row penalties* are computed for each row of the $(0, \pm 1)$-matrix based on the number of conflicts in which the row participates. During each step of the algorithm, a row with the maximum penalty is chosen and is either reflected or deleted. The row penalties are updated after each reflection and deletion. The procedure is repeated until the matrix has no conflicts.

The algorithms from [9] are based on extracting a sequence of Generalized Upper Bound (GUB) structures from the matrix. A *GUB structure* for an LP constraint matrix is a subset of rows such that each column has at most one non-zero entry within this subset. If the algorithm terminates after extracting two such structures, it is called *two-stage*; algorithms that extract sequences of more than two GUB structures are called *multi-stage*. It has been shown in [9] that multi-stage algorithms dominate their two-stage counterparts. Therefore in this study we only consider multi-stage GUB algorithms.

Following [9], we tested GUB algorithms based on two different methods for detecting GUB structures. One method is based on a concept akin to Markowitz's merit count [17] and another is based on finding a maximum independent set in a graph that corresponds to the coefficient matrix. The multi-stage GUB algorithms that are based on these two approaches are denoted by *M-GUBMC* and *M-INDSET* respectively. The reader is referred to [9] for further details on these algorithms.

We implemented SGA in C++ and the other algorithms in Fortran 77. Computational experiments were carried out on an Intel Celeron 550MHz computer with 384MB of RAM. The computational results are presented under the heading of each algorithm in Table 1.

We used the number of detected network rows (NETR) to measure the effectiveness of algorithms. Since we are interested in measuring effectiveness of SGA compared to other algorithms, we take the number of network rows detected by SGA as the basis for comparison. For a given model, we compute the following parameter (denoted by *IMPR*) for each of the non-SGA algorithms:

$$IMPR(\mathcal{A}) = \frac{NETR_{\mathcal{A}} - NETR_{SGA}}{NETR_{SGA}} \times 100\%,$$

where $NETR_{\mathcal{A}}$ is the number of network rows detected by the algorithm $\mathcal{A}$ and $NETR_{SGA}$ is the number of network rows obtained by SGA. This parameter shows how many more network rows the algorithm has found compared to SGA. The value is expressed as a percentage of the number of network rows detected by SGA. A positive value means that the algorithm has performed better than SGA; a negative value means worse performance. The value of this parameter is presented in Table 1 for M-GUBMC, M-INDSET and RSD under the respective "IMPR" headings.

Table 2 summarizes the values of the IMPR parameter for each of the algorithms. Each data value in the table shows on how many of the test models the value of the parameter has fallen within a given range. While compiling this table, we eliminated all models that have fewer than 100 rows in the $(0, \pm 1)$-matrix. Such models are of little practical interest from the point of view of reflected network detection, and they only distort the picture. This left us with 67 different models, results for which are represented in Table 2.

It can be seen from Table 2 that RSD and SGA perform very similarly in the sense that RSD outperforms SGA about as frequently as the opposite happens. On the other hand, M-GUBMC and M-INDSET are substantially less competitive. On our set of test models they seldom outperform SGA, while frequently falling short of matching SGA's performance. We therefore conclude that SGA and RSD are clear winners in terms of the number of network rows they detect.

We evaluated the efficiency of algorithms by measuring the elapsed CPU time. The results are presented in Table 1 under the heading TIME. These results include only the time spent on detection of network structures. Time required to read the initial data and to apply preprocessing and scaling procedures is excluded since it is the same for all algorithms. Table 3 shows a histogram of algorithm running times on the same set of instances that was used to compile Table 2.

Times taken by algorithms to complete the whole set of 100 models were as follows: SGA: 11.33s, RSD: 214.98s, M-GUBMC: 669.26s, M-INDSET: 3.61s.

These times can be clearly divided into two groups, separated by at least one order of magnitude. M-INDSET is the fastest, but unfortunately the quality of its results is not as good as that of SGA or RSD. The other fast algorithm is SGA, which in our tests was much faster than its main competitor, RSD, while still producing very similar results.

We also tested the performance of these algorithms on a set of three instances of a larger scale industrial model taken from the domain of supply chain planning [16]. These models are instances of a multi-stage multi-period production and distribution model and are known to possess a large proportion of embedded network rows. The models are available for download at `http://www.cs.rhul.ac.uk/home/gutin/models/`. The problem statistics of these models as well as the numbers of network rows detected by RSD, M-INDSET and SGA are displayed in Table 4. In these cases, while time taken is between one and three seconds for M-INDSET, SGA takes between two and five seconds per model. RSD is significantly slower than these two algorithms. We see from the results in Table 4 that SGA clearly outperforms the other two algorithms in terms of the number of network rows detected. Moreover, only SGA terminates by concluding that the $(0, \pm 1)$-matrix is a reflected network itself for the models 2 and 3. (Before the experiments, we were not aware of the fact that the matrices for models 2 and 3 are reflected networks.)

6 Conclusions

The effectiveness and efficiency results from the previous section show that SGA is a fast algorithm that produces competitive results.

The quality of the solutions found by SGA can be further improved if one is ready to spend more time on detecting an embedded reflected network. Indeed, our computational experiments show that the number of rows in the reflected network found depends (sometimes, significantly) on the spanning forest built. To enhance the results, one can build several spanning forests rather than just one. We believe that this approach is worthy of further study.

To obtain an independent set I we have used the degree-greedy algorithm. The set I, in many cases, can be enlarged by using local search improvement algorithms (see [19]). However, the improvement algorithms are normally time demanding.

Acknowledgements The authors are grateful to Thomas Zaslavsky for his valuable comments on the previous version of this paper and to the referees for their wealth of comments and suggestions that led to a much better presentation. The work of N. Gülpınar was supported in part by Turkish Educational Council and Muğla University. The work of G. Gutin was supported in part by a grant from EPSRC.

References

[1] B.M. Baker and P.J. Maye, A Heuristic for Finding Embedded Network Structure in Mathematical Programmes. Europ. Jour. OR 67 (1993) 52–63.

[2] J.J. Bartholdi, A Good Submatrix is Hard to Find. OR Letters 1 (1982) 190–193.

[3] R.E. Bixby and R. Fourer, Finding Embedded Network Rows in Linear Programs I. Extraction Heuristics. Manag. Science 34 (1988) 342–376.

[4] R.E. Bixby, and W.H. Cunningham, Converting Linear Programs to Network Problems. Math. of OR 5 (1980) 321–356.

[5] A.L. Brearley, G. Mitra and H.P. Williams, Analysis of Mathematical Programming Problems Prior to Applying the Simplex Algorithm. Math. Prog. 8 (1975) 54–83.

[6] G.G. Brown and W.G. Wright, Automatic Identification of Embedded Network Rows in Large-Scale Optimization Models. Math. Prog. 29, (1984) 41–56.

[7] G.G. Brown and D.S. Thomen, Automatic Identification of Generalized Upper Bounds in Large-Scale Optimization Models. Manag. Science 26 (1980) 1166–1184.

[8] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge (1990).

[9] N. Gülpınar, G. Gutin, G. Mitra and I. Maros, Detecting Embedded Pure Network Structures by Using GUB and Independent Set Algorithms. Comput. Optim. Applic. 15 (2000) 235–247.

[10] D.M. Gay, Electronic Mail Distribution of Linear Programming Test Problems. Math. Prog. Soc. Coal, Newsletter 13 (1985) 10–12.

[11] F. Harary, On the Notion of Balance of a Signed Graph. Michigan Math. Jour. 2 (1953/54) 143–146.

[12] P. Hansen, Labelling Algorithms for Balance in Signed Graphs, In: *Problémes Combinatoires et Theorie des Graphes* (Colloq. Internat., Orsay, 1976), 215–217. Colloques Internat. du CNRS 260 Paris (1978).

[13] F. Harary and J.A. Kabell, A Simple Algorithm to Detect Balance in Signed Graphs. Math. Social Science 1 (1980/81) 131–136.

[14] I. Heller and C.B. Tompkins, An Extension of a Theorem of Dantzig's, In: H.W.Kuhn and A.W.Tucker, eds., *Linear Inequalities and Related Systems* 247–252. Annals Math. Studies, 38, Princeton Univ. Press, Princeton, N.J. (1956).

[15] A.C. Hsu and R. Fourer, Identification of Embedded Network Structure in Linear Programming Models. GSIA Working Paper, 1997-58.

[16] C. Lucas, E. Messina and G. Mitra, Risk and Return Analysis of a Multi-Period Strategic Planning Problem, in: A.H. Christer, S. Osaki and L.C. Thomas, *Stochastic Modelling in Innovative Manufacturing*, Lect. Notes in Economics and Math. Systems 445 (1997) 81-96, Springer-Verlag.

[17] H.M. Markowitz, The Elimination Form of the Inverse and Its Application to Linear Programming. Manag. Science 3 (1957) 255–267.

[18] V.Th. Paschos, A $\Delta/2$-Approximation for the Maximum Independent Set Problem. Inform. Proc. Let. 44 (1992) 11–13.

[19] Y. Saab, Iterative improvement of vertex covers. Inform. Proc. Let. 55 (1995) 95–98.

[20] T. Zaslavsky, Bibliography of Signed and Gain Graphs. Electronic J. Combin., Dynamic Surveys DS8, 151 pp. `http://www.combinatorics.org/index.html`

[21] T. Zaslavsky, Glossary of Signed and Gain Graphs. Electronic J. Combin., Dynamic Surveys DS9, 41 pp. `http://www.combinatorics.org/index.html`

[22] T. Zaslavsky, Signed Graphs. Discr. Appl. Math. 4 (1982) 47–74.

[23] T. Zaslavsky, How Colorful the Signed Graph? Discr. Math. 52 (1984) 279–284.

Model	Problem Statistics			TMROW	SGA		RSD			M-GUBMC			M-INDSET		
	Rows	Cols	Nonzeros	Rows	NETR	TIME	NETR	TIME	IMPR	NETR	TIME	IMPR	NETR	TIME	IMPR
25fv47	822	1571	11127	224	204	0.00	207	0.01	1.47%	198	0.03	-2.94%	187	0.04	-8.33%
80bau3b	2263	9799	29063	1374	1336	0.17	1332	0.08	-0.30%	1335	0.61	-0.07%	1308	0.01	-2.10%
agg2	517	302	4515	141	58	0.00	61	0.00	5.17%	59	0.01	1.72%	62	0.01	6.90%
agg3	517	302	4531	141	58	0.02	61	0.02	5.17%	59	0.00	1.72%	62	0.00	6.90%
bnl1	644	1175	6129	275	259	0.01	257	0.00	-0.77%	260	0.02	0.39%	255	0.00	-1.54%
bnl2	2325	3489	16124	1418	1322	0.11	1321	0.09	-0.08%	1307	5.78	-1.13%	1262	0.01	-4.54%
cre-a	3517	4067	19054	1247	828	0.08	809	0.54	-2.29%	825	0.45	-0.36%	799	0.03	-3.50%
cre-b	9649	72447	328542	3690	2699	0.64	2728	74.00	1.07%	2731	20.99	1.19%	2602	0.27	-3.59%
cre-c	3069	3678	16922	997	736	0.05	739	0.34	0.41%	729	0.11	-0.95%	714	0.03	-2.99%
cre-d	8927	69980	312626	3040	2523	0.52	2648	23.28	4.95%	2581	8.74	2.30%	2544	0.17	0.83%
cycle	1904	2857	21322	507	505	0.03	505	0.02	0.00%	505	0.05	0.00%	502	0.00	-0.59%
czprob	930	3523	14173	719	718	0.03	718	0.43	0.00%	717	0.03	-0.14%	718	0.00	0.00%
d2q06c	2172	5167	35674	844	788	0.04	795	0.07	0.89%	757	0.24	-3.93%	711	0.01	-9.77%
degen3	1504	1818	26230	1503	707	0.27	777	4.09	9.90%	565	0.14	-20.08%	618	0.03	-12.59%
dfl001	6072	12230	41873	6019	3205	1.06	3584	45.94	11.83%	3167	219.97	-1.19%	2752	1.16	-14.13%
fffff800	525	854	6235	157	96	0.01	98	0.01	2.08%	94	0.00	-2.08%	97	0.16	1.04%
ganges	1310	1681	7021	631	517	0.01	522	0.01	0.97%	515	0.22	-0.39%	501	0.18	-3.09%
gfrd-pnc	617	1092	3467	590	522	0.02	481	0.01	-7.85%	469	0.08	-10.15%	441	0.00	-15.52%
greenbea	2393	5405	31499	915	876	0.03	880	0.11	0.46%	877	0.08	0.11%	858	0.01	-2.05%
greenbeb	2393	5405	31499	915	875	0.03	880	0.13	0.57%	878	0.08	0.34%	859	0.02	-1.83%
maros	847	1443	10006	305	300	0.02	295	0.00	-1.67%	293	0.03	-2.33%	282	0.00	-6.00%
modszk1	688	1620	4158	148	127	0.02	118	0.01	-7.09%	118	0.13	-7.09%	112	0.00	-11.81%
nesm	663	2923	13988	190	190	0.00	190	0.00	0.00%	188	0.06	-1.05%	178	0.00	-6.32%
osa-07	1119	23949	167643	1048	1043	0.11	1043	0.08	0.00%	1043	0.50	0.00%	1043	0.07	0.00%
osa-14	2338	52460	367220	2267	2262	0.39	2262	0.17	0.00%	2262	1.36	0.00%	2262	0.04	0.00%
osa-30	4351	100024	700160	4280	4280	0.72	4275	0.34	-0.12%	4275	3.67	-0.12%	4275	0.11	-0.12%
perold	626	1376	6026	150	143	0.00	143	0.00	0.00%	143	0.03	0.00%	135	0.00	-5.59%
pilot.ja	941	1988	14706	205	198	0.02	194	0.00	-2.02%	193	0.00	-2.53%	191	0.00	-3.54%
pilot	1442	3652	43220	276	249	0.02	249	0.05	0.00%	253	0.02	1.61%	253	0.02	1.61%
pilot.we	723	2789	9218	202	202	0.00	202	0.00	0.00%	201	0.01	-0.50%	201	0.02	-0.50%
pilot87	2031	4883	73804	340	302	0.01	299	0.04	-0.99%	302	0.02	0.00%	303	0.01	0.33%
pilotnov	976	2172	13129	209	203	0.00	199	0.00	-1.97%	198	0.02	-2.46%	197	0.00	-2.96%
scfxm2	661	914	5229	282	250	0.01	244	0.00	-2.40%	237	0.00	-5.20%	206	0.01	-17.60%
scfxm3	991	1371	7846	423	375	0.02	366	0.03	-2.40%	355	0.01	-5.33%	309	0.02	-17.60%
sctap2	1091	1880	8124	470	470	0.02	470	0.00	0.00%	470	0.00	0.00%	470	0.00	0.00%
sctap3	1481	2480	10734	620	620	0.03	620	0.00	0.00%	620	0.00	0.00%	620	0.00	0.00%
seba	516	1028	4874	402	134	0.27	138	0.23	2.99%	135	0.03	0.75%	134	0.02	0.00%
shell	537	1775	4900	483	482	0.01	482	0.00	0.00%	447	0.08	-7.26%	394	0.00	-18.26%
ship12l	1152	5427	21597	828	732	0.12	732	0.28	0.00%	732	0.54	0.00%	732	0.16	0.00%
ship12s	1152	2763	10941	456	312	0.03	360	0.12	15.38%	360	0.13	15.38%	360	0.00	15.38%
sierra	1228	2036	9252	1155	762	0.06	790	0.23	3.67%	672	0.05	-11.81%	672	0.02	-11.81%
stocfor2	2158	2031	9492	1262	1042	0.06	947	0.14	-9.12%	947	0.09	-9.12%	901	0.01	-13.53%
stocfor3	16676	15695	74004	9632	8054	5.77	7287	13.34	-9.52%	7287	7.17	-9.52%	6915	0.69	-14.14%
woodw	1099	8405	37478	301	301	0.01	301	0.00	0.00%	301	0.01	0.00%	301	0.00	0.00%

TMROW: The number of rows in the $(0, \pm 1)$-matrix; **RSD**: Row-Scanning Deletion Algorithm; **M-GUBMC**: Multi-Stage GUB with Merit Count Algorithm; **M-INDSET**: Multi-Stage Independent Set Algorithm; **SGA**: Signed Graph Algorithm; **NETR**: The number of network rows detected by the algorithm; **TIME**: Elapsed CPU time in seconds; **IMPR**: Number of network rows detected in excess of that of SGA, expressed as a percentage.

Table 1: Characteristics of test problems and performance of the four tested algorithms

Improvement over SGA	RSD	M-GUBMC	M-INDSET
-65% – -55%	0	1	1
-55% – -45%	0	0	0
-45% – -35%	0	0	0
-35% – -25%	1	1	1
-25% – -15%	0	3	7
-15% – -5%	7	13	16
-5% – 5%	51	47	38
5% – 15%	7	1	3
15% – 25%	1	1	1

Table 2: Distribution of relative performance of the algorithms compared to SGA on test problems with not fewer than 100 rows in the $(0, \pm 1)$-matrix

Running Time	SGA	RSD	M-GUBMC	M-INDSET
$< 0.01s$	20	24	15	33
$0.01 - 0.1s$	34	24	33	26
$0.1 - 1s$	11	14	11	7
$1 - 10s$	2	1	6	1
$10 - 100s$	0	4	1	0
$\geq 100s$	0	0	1	0

Table 3: Distribution of running times of the tested algorithms on test problems with not fewer than 100 rows in the $(0, \pm 1)$-matrix

MODELS	ROWS	COLS	NZEROS	TMROW	RSD	M-INDSET	SGA
model1	4740	36277	105875	4291	3509	3755	4139
model2	4450	62366	204954	3690	3060	3306	3690
model3	3692	59907	158650	3690	3060	3306	3690

Table 4: The problem statistics and the number of network rows for supply chain models