The maximum weighted submatrix coverage problem: A CP approach

Guillaume Derval[0000−0002−6700−3519], Vincent Branders[0000−0001−8688−7498], Pierre Dupont[0000−0003−4835−6519], Pierre Schaus[0000−0002−3153−8941]

UCLouvain - ICTEAM/INGI
{firstname.lastname}@uclouvain.be

Abstract. The objective of the maximum weighted submatrix coverage problem (MWSCP) is to discover K submatrices that together cover the largest sum of entries of the input matrix. The special case of $K = 1$ called the maximal-sum submatrix problem was successfully solved with CP. Unfortunately, the case of $K > 1$ is more difficult to solve as the selection of the rows of the submatrices cannot be decided in polynomial time solely from the selection of K sets of columns. The search space is thus substantially augmented compared to the case $K = 1$. We introduce a complete CP approach for solving this problem efficiently composed of the major CP ingredients: 1) filtering rules, 2) a lower bound, 3) dominance rules, 4) variable-value heuristic, and 5) a large neighborhood search. As the related biclustering problem, MWSCP has many practical data-mining applications such as gene module discovery in bioinformatics. Through multiple experiments on synthetic and real datasets, we provide evidence of the practicality of the approach both in terms of computational time and quality of the solutions discovered.

Keywords: Constraint Programming · Maximum Weighted Submatrix Coverage Problem · Data mining.

1 Introduction

Constraint Programming (CP) has received an increasing interest for solving unsupervised (clustering) data-mining problems [14,18,12,1,3,7,5]. This article is interested into the mining of a numerical matrix to discover submatrices (also called biclusters) that capture a high total value. More exactly we consider an input matrix $\mathcal{M}$ with m rows and n columns where element $\mathcal{M}_{i,j}$ is a given real value. The matrix is associated with a set of rows $R = \{r_1, \ldots, r_m\}$ and a set of columns $C = \{c_1, \ldots, c_n\}$. We use $(R; C)$ to denote matrix $\mathcal{M}$. If $I \subseteq R$ and $J \subseteq C$ are subsets of the rows and of the columns, respectively, $\mathcal{M}_{I,J} = (I; J)$ denotes the submatrix $\mathcal{M}_{I,J}$ of $\mathcal{M}$ that contains only the elements $\mathcal{M}_{i,j}$ belonging to the submatrix with set of rows I and set of columns J.

The maximal sum submatrix problem introduced in [4] is to discover a subset of rows and columns of an input matrix that maximizes the sum of the covered entries. An example is provided in Fig. 1.

Definition 1. ***The Maximal-Sum Submatrix Problem.*** *Given a matrix $\mathcal{M} \in \mathbb{R}^{m \times n}$. Let $R = \{1, \ldots, m\}$ and $C = \{1, \ldots, n\}$ be index sets for rows and for columns, respectively. The maximal-sum submatrix is the submatrix $(I^*; J^*)$, with $I^* \subseteq R$ and $J^* \subseteq C$, such that:*

$$(I^*; J^*) = \operatorname*{argmax}_{I,J} f(I, J) = \operatorname*{argmax}_{I,J} \sum_{i \in I, j \in J} \mathcal{M}_{i,j} \tag{1}$$

The objective function rewards the selection of positive values and penalizes selection of negative values. In case of positive input matrices, the domain expert can subtract a constant threshold θ from all entries. The choice of this threshold is not discussed here. Therefore, the problem matrix is assumed to contain both positive and negative values in order to be interesting and challenging to solve.

	C1	C2	C3	C4	C5	C6
R1	-4.2	-2.1	-3.2	3.9	2.1	5.0
R2	-5.1	2.3	-4.1	3.1	4.0	-0.9
R3	-3.2	1.9	4.0	3.4	-2.1	-4.1
R4	-5.2	0.9	0.3	-4.1	3.0	2.0
R5	-0.1	0.1	-1.2	5.2	0.9	1.9
R6	-4.2	-5.0	0.9	2.7	0.2	-1.9

The submatrix $(\{R_1, R_2, R_4, R_5\}; \{C_2, C_4, C_5, C_6\})$, in red, is of maximal sum as the value of the objective function is 27.3.

For $K = 2$, the two submatrices depicted in red, $(\{R_1, R_2, R_4, R_5\}; \{C_2, C_4, C_5, C_6\})$, and blue, $(\{R_3, R_4, R_6\}; \{C_3, C_4\})$, are of maximal sum. The objective value equals 38.6.

Fig. 1: Example of matrix and associated submatrices of maximal sum.

The maximum weighted submatrix coverage problem, that we study in this work, generalizes the maximal-sum submatrix problem to K submatrices. An example is provided in Fig. 1.

Definition 2. ***The Maximum Weighted Submatrix Coverage Problem.*** *Given a matrix $\mathcal{M} \in \mathbb{R}^{m \times n}$ and a parameter K, the maximum weighted submatrix coverage problem is to select a set of submatrices $(\mathcal{R}_k, \mathcal{C}_k)$ with $k = 1, \ldots, K$ such that the sum of the cells covered by at least one submatrix is maximal:*

$$(\mathcal{R}_1^*; \mathcal{C}_1^*), \ldots, (\mathcal{R}_K^*; \mathcal{C}_K^*) = \operatorname*{argmax}_{(\mathcal{R}_1; \mathcal{C}_1), \ldots, (\mathcal{R}_K; \mathcal{C}_K)} \sum_{i \in R, j \in C} \mathcal{M}_{i,j} \times \mathbb{1}_{cover}((i, j)) \tag{2}$$

where $\mathbb{1}_{cover}$ is the indicator function over the set $cover = \bigcup_{k \in 1..K} \mathcal{R}_k \times \mathcal{C}_k$.

1.1 Applications

The maximum weighted submatrix coverage problem has many practical data mining applications where one is interested to discover K strong relations between two groups of variables (rows and columns) represented as a matrix:

- In gene expression analysis, rows correspond to genes and columns to samples and the value in $\mathcal{M}_{i,j}$ is the measurement of the expression of gene i in sample j. One is typically interested in finding subsets of genes that present high

expression in a subset of the samples as it would indicate that a particular biological pathway made of these genes is active in these samples.
- In migration data, value $\mathcal{M}_{i,j}$ represents the number of persons that moved from location i to j. The goal is the to identify groups of locations that together migrate to other groups of locations.
- A sports journalist could also be interested in Olympic games to discover group of countries that together obtained similar strong performances on the same subset of sports. The matrix value $\mathcal{M}_{i,j}$ then represents the number of medals obtained by the country i in sport j.
- Dendograms and Sankey plots are standard visualization tools to represent relations. Unfortunately those plots quickly suffer from cluttering for large matrices. The MWSCP can be used as a preliminary step to preselect submatrices that can then be analyzed more easily with those plots.

1.2 Related Work

The maximal-sum submatrix problem was introduced in [4] and efficiently solved using constraint programming with a dedicated global constraint.

The biclustering problems are concerned with the discovery of homogeneous submatrices (called biclusters in this context) rather than maximizing the sum of the covered entries. A comprehensive review can be found in [15]. Common approaches are heuristic based and greedily selects the next bicluster after randomization of entries covered by the previously discovered biclusters.

The maximum subarray problem introduced by [2] is looking for a maximal-sum submatrix with contiguous subsets of rows and contiguous subset of columns.

The maximum ranked tile mining problem has been introduced in [14]. This is a special case of the maximal-sum submatrix problem for which the matrix entries are discrete ranks, corresponding to a permutation of column indices on each row. Another relevant difference is the constraint that sets of entries covered by the submatrices are disjoint. This restriction is more convenient for solving the problem efficiently but unnatural for the applications motivating this work.

1.3 Contributions

Our contributions are:
- The introduction of the maximum weighted submatrix coverage problem (MWSCP) as a generalization of the maximal-sum submatrix problem.
- A CP approach for solving MWSCP including filtering, lower-bound, dominance rules, a variable heuristic, and a large neighborhood search.
- An evaluation of the performances of the CP approach as compared to a greedy baseline approach (using the maximal-sum submatrix problem as subroutine) and two mathematical programming models on synthetic and real datasets.

2 CP approach

Constraint programming (CP) is a flexible programming paradigm for solving (discrete) optimization problems. A CP model is a triplet (V, D, C) where V is the set of variables, D their domains and C is a set of constraints. In constraint programming the set domain bounds representation [8] is used to approximate the domain of a set variable $\mathcal{S}$ by a closed interval denoted $[\mathcal{S}^{\in}, \mathcal{S}^{\in} \cup \mathcal{S}^{\perp}]$ where $\mathcal{S}^{\in}$ are the mandatory elements and $\mathcal{S}^{\perp}$ are the possible additional ones ($\mathcal{S}^{\in} \cap \mathcal{S}^{\perp} = \emptyset$). Such an interval represents all the sets in between those two bound sets according to the inclusion relation $\{S \mid \mathcal{S}^{\in} \subseteq S \subseteq (\mathcal{S}^{\in} \cup \mathcal{S}^{\perp})\}$. A set variable is bound (or assigned) whenever it contains a single set in its domain. This situation (called an assignment) happens when set interval bounds are equal, that is the possible set is empty: $\mathcal{S}^{\perp} = \emptyset$.

For a set variable, the domain's update operations are:

- The inclusion of an item j in the mandatory set, denoted require$(j, \mathcal{S})$, which implies that $\mathcal{S}^{\in} \leftarrow \mathcal{S}^{\in} \cup \{j\}$ and $\mathcal{S}^{\perp} \leftarrow \mathcal{S}^{\perp} \setminus \{j\}$.
- The exclusion of an item j from the possible set, denoted exclude$(j, \mathcal{S})$, which implies that $\mathcal{S}^{\perp} \leftarrow \mathcal{S}^{\perp} \setminus \{j\}$ (and $j \notin \mathcal{S}^{\in}$).

For each submatrix k, a set variable $\mathcal{R}_k$ (resp. $\mathcal{C}_k$) is introduced to represent the possible rows (resp. columns) selections in submatrix k.

Preliminary notations. We define $\mathcal{R}_k^{\in,+j}$ (resp. $\mathcal{R}_k^{\in,-j}$) as the subset of $\mathcal{R}_k^{\in}$ whose matrix value in column j is positive (resp. strictly negative):

$$\mathcal{R}_k^{\in,+j} = \{i \in \mathcal{R}_k^{\in} \mid \mathcal{M}_{i,j} \geq 0\} \qquad \mathcal{R}_k^{\in,-j} = \{i \in \mathcal{R}_k^{\in} \mid \mathcal{M}_{i,j} < 0\} \tag{3}$$

Similar notations hold for $\mathcal{C}_k$ and $\perp$. The sum of the elements in a given row i (resp. column j) and in a column (resp. row) set S is noted as:

$$\underset{\text{row } i}{\text{sum}}(S) = \sum_{j \in S} \mathcal{M}_{i,j} \qquad \underset{\text{col } j}{\text{sum}}(S) = \sum_{i \in S} \mathcal{M}_{i,j} \tag{4}$$

The set of cells selected by at least one submatrix is denoted Cover$^{\in}$. The set of cells excluded by all submatrices is denoted Cover$^{\notin}$:

$$\text{Cover}^{\in} = \{(i,j) \mid \exists k : i \in \mathcal{R}_k^{\in} \wedge j \in \mathcal{C}_k^{\in}\} \tag{5}$$

$$\text{Cover}^{\notin} = \{(i,j) \mid \forall k : i \notin (\mathcal{R}_k^{\in} \cup \mathcal{R}_k^{\perp}) \vee j \notin (\mathcal{C}_k^{\in} \cup \mathcal{C}_k^{\perp})\} \tag{6}$$

The CP resolution is made via a Depth-First-Search (DFS) exploration. The following subsections discuss the search space, sketch the algorithm and its key components.

2.1 Search Space

As explained in [4], the search space of MWSCP with $K = 1$ can be limited to searching on a single dimension, for instance $\mathcal{C}_1$. Indeed, the variable $\mathcal{R}_1$ can be fixed optimally in polynomial time by a simple inspection argument: $\forall i \in \mathcal{R}_1^{\perp} : \underset{\text{row } i}{\text{sum}}(\mathcal{C}_1) > 0 \implies i \in \mathcal{R}_1^{\in}$.

For $K > 1$, once all the columns set variables are fixed ($\mathcal{C}_k\ \forall k \in [1..K]$) it remains to decide for each row i and each submatrix k whether i should be part of $\mathcal{R}_k$ or not. Those K decisions per row does not enjoy the monotonicity or the anti-monotonicity properties as illustrated on the next example.

Example 1. Let us consider $K = 2$ with column selection $\mathcal{C}_1 = \{1, 3\}, \mathcal{C}_2 = \{2, 3\}$. For the 1×3 input matrix $\mathcal{M} = [[2, 2, -3]]$. Individually for each submatrix, the sum of entries that would be covered by selecting this row in both $\mathcal{R}_1$ and $\mathcal{R}_2$ would be negative (-1). But since weights of covered elements count only once, the value -3 is added only once and the objective value obtained is 1. Now consider the matrix $\mathcal{M} = [[-2, -2, 3]]$. Individually for each submatrix, the sum of entries that would be covered by selecting this row in both $\mathcal{R}_1$ and $\mathcal{R}_2$ would be positive (1). But since weights of covered elements count only once, the value 3 is added only once and the final objective value is -1.

Actually, those K decisions per row cannot be optimally taken in polynomial time anymore as stated in Theorem 1. As a consequence, the CP search will have to branch both on the rows and columns variables rather than branching on the columns only.

Theorem 1. *For fixed variables $\mathcal{C}_k\ \forall k \in [1..K]$, fixing optimally $\mathcal{R}_k\ \forall k \in [1..K]$ is NP-Hard.*

Proof. We reduce the NP-Hard Set Cover Problem [11] to our problem: Given a universe $U = \{1, \dots, n\}$ and a set $\{S_1, \dots, S_K\}$ of K subsets of U, the Set Cover Problem is to find the minimum number of sets such that their union covers the universe. We construct a matrix with a single row and $n+K$ columns. The unique row values of this matrix are given by the regular expression $[K+1]\{n\}[-1]\{K\}$ (value $K + 1$ repeated n times followed by -1 repeated K times). The column variables are fixed to $\mathcal{C}_k = S_k \cup \{n + k\}$. In this reduction, S_k is selected if and only if $\mathcal{R}_k = \{1\}$ for every set k. A first observation is that any optimal solution covers the universe otherwise it could be improved by K by selecting any additional set that contains an uncovered element. The optimal objective function can thus be written as $n \cdot (K + 1) - |\{k \mid \mathcal{R}_k = \{1\}\}|$. As $n \cdot (K + 1)$ is fixed, maximizing this expression amounts at minimizing $|\{k \mid \mathcal{R}_k = \{1\}\}|$ which is exactly the set cover objective. □

2.2 Resolution via Depth-First-Search

The CP resolution through Depth-First-Search (DFS) exploration is sketched in Algorithm 1. All the procedures are assumed to take the decision variables $\{\mathcal{R}_1, \dots, \mathcal{R}_K, \mathcal{C}_1, \dots, \mathcal{C}_K\}$ and the input matrix $\mathcal{M}$ as parameters.

The procedure SELECTUNBOUNDSETVAR chooses a not yet bound set variable among $\{\mathcal{R}_1^\perp, \dots, \mathcal{R}_K^\perp, \mathcal{C}_1^\perp, \dots, \mathcal{C}_K^\perp\}$. The subsequent line chooses for the selected row/column set of some submatrix k, the specific row/column i (among the possible ones) to be included on the left branch and to be excluded on the right branch. The explored search tree is thus binary. Once the constraint is

Algorithm 1 Sketch of the DFS resolution algorithm

```
function SolveDFS( )
    if !allVariablesBound( ) then
        S ← selectUnBoundSetVar( )
        i ← selectValue(S^⊥)
        for action ∈ [require(i, S), exclude(i, S)] do
            saveState( )
            post(action)
            propagateDominanceRule( )
            (lb, cb, ub) ← updateBounds( )
            best ← max(best, cb)
            if ub > best then
                SolveDFS( )
            end if
            restoreState( )
        end for
    end if
end function
```

posted, and the previous state saved for later backtracking, the procedure propagateDominanceRule can include (exclude) rows or columns in every submatrix that can be proven to (not) participate in any optimal solution. The updateBounds function updates and returns the lower, current and upper bounds for the state. The current bound is obtained by transforming the partial assignment into a complete feasible solution that excludes all rows/columns in $\perp$. If the current bound cb is better than the best value found so far (stored in variable best), the current state $(\mathcal{R}_1^{\in}, \dots, \mathcal{R}_K^{\in}, \mathcal{C}_1^{\in}, \dots, \mathcal{C}_K^{\in})$ is a better solution and the value of the variable best (storing the best objective found so far) is updated (and the solution is logged). Once this is done, a check is made to ensure that there may still be a better solution below this tree node, by verifying that the upper bound is greater than the best objective value found so far; if that is the case, the DFS continues recursively. Once these steps are done, the state is backtracked and the next state visited.

Efficient backtracking is achieved through trailing, which is a state management strategy that facilitates the restoration of the computation state to an earlier version. Trailing enables the design of reversible objects. We refer to MiniCP [13] for a detailed description of trail-based solvers and to [17] for a trailed based implementation of set domains with sparse-sets.

The following subsections are dedicated to the four main functions of our algorithm: selectUnBoundSetVar, selectValue, propagateDominanceRule and updateBounds.

2.3 Functions selectUnBoundSetVar and selectValue

selectUnBoundSetVar chooses, at each step of the DFS, the next (unbounded) row/column interval set $\mathcal{S}$ to branch on, while selectValue selects the value $l \in \mathcal{S}^{\perp}$ to include/exclude from this set when branching. That is, when a pair $(\mathcal{S}, l)$ has been chosen, the DFS branches on the left, by setting require$(l, \mathcal{S})$, and on the right, by setting exclude$(l, \mathcal{S})$. The decision of the interval set and of the value are not done independently. To choose the next

(set,value) pair to branch on, our algorithm maintains two (reversible) counters per row or column and per submatrix:

– $t^{\text{row}}_{k,i}$ contains the sum of cell values that will be immediately added to the objective value if row i is included in $\mathcal{R}_k$:

$$t^{\text{row}}_{k,i} = \underset{\text{row } i}{\text{sum}}\left(\{j \mid j \in \mathcal{C}^{\in}_k \ \wedge (i,j) \notin \text{Cover}^{\in}\}\right) \tag{7}$$

– $p^{\text{row}}_{k,i}$ contains the sum of positive values in the line i that *could* be taken by submatrix k, i.e. whose columns have not been excluded:

$$p^{\text{row}}_{k,i} = \underset{\text{row } i}{\text{sum}}\left(\{j \mid j \in (\mathcal{C}^{\in}_k \cup \mathcal{C}^{\perp}_k) \ \wedge (i,j) \notin \text{Cover}^{\in}\}\right) \tag{8}$$

$t^{\text{col}}_{k,j}$ and $p^{\text{col}}_{k,j}$ are defined similarly. The algorithm then selects the (submatrix, row) (or (submatrix, column)) pair (k,i) (or (k,j)) that maximizes $t^{\text{row}}_{k,i}$ (or $t^{\text{col}}_{k,j}$). Ties are broken by maximizing $p^{\text{row}}_{k,i}$ (or $p^{\text{col}}_{k,j}$). The selected interval set and value are then $\mathcal{R}_k$ and i (or $\mathcal{C}_k$ and j).

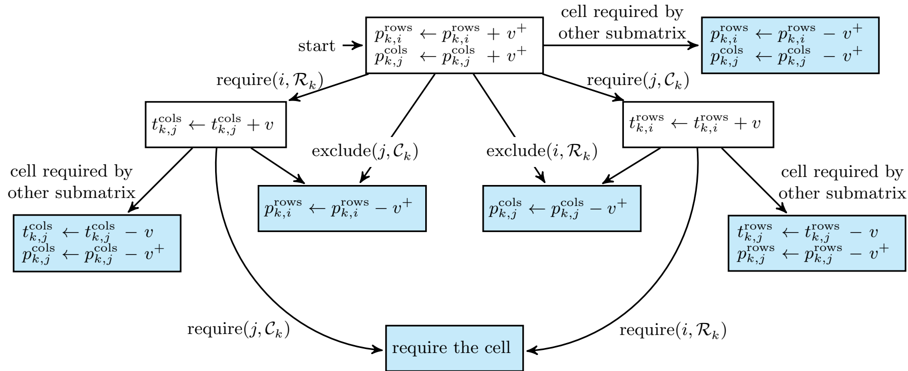

Fig. 2: FSM maintained for each (row, column, submatrix) i, j, k in the variable/value selection algorithm. For simplicity, $v = M_{i,j}$, $v^+ = \max(v, 0)$ and $v^- = \min(v, 0)$. FSMs states in blue are terminal states.

Recomputing these counters at each iteration is costly, as this operation is in $\mathcal{O}(Knm + K(n+m))$ for the MWSCP with an $m \times n$ matrix and K submatrices. We propose here to maintain these counters using the finite state machine (FSM) shown in Fig. 2. The algorithm we propose virtually maintains a FSM for each (row, column, submatrix) triplet. The FSMs are updated each time a row/column is added to/excluded from a submatrix:

– When a row i is included in/removed from the submatrix k, at most n FSMs must be updated (one for each cell in the row).
– When a column j is included in/removed from the submatrix k, at most m FSMs must be updated (one for each cell in the column).
– Updating a cell is $\mathcal{O}(1)$, if it does not become *selected* by a submatrix (i.e. the row and column of the cell are both in the mandatory sets of the submatrix).
– If a cell becomes *selected*, $K-1$ other cells must be updated.

Given that $\Delta_{\text{rows}}, \Delta_{\text{cols}}$ and Δ_{selected} are respectively the number of added or excluded (submatrix, row) tables, added/excluded (submatrix, column) tables and selected cells between two calls of the algorithm, this update runs in $\mathcal{O}(\Delta_{\text{rows}} n + \Delta_{\text{cols}} m + \Delta_{\text{selected}} K)$. To this update process must be added the verification of the counters to select the best set/value pair, which is in $\mathcal{O}(K(m+n))$.

Over a complete branch of the DFS tree (which has a maximum depth of $K(m+n)$), we have that:

$$\sum_{\text{branch}} \Delta_{\text{rows}} \leq K \cdot m \qquad \sum_{\text{branch}} \Delta_{\text{cols}} \leq K \cdot n \qquad \sum_{\text{branch}} \Delta_{\text{selected}} \leq n \cdot m \tag{9}$$

Over a complete branch, the FSM-based algorithm maintains the states and returns the best set/value pair in $\mathcal{O}(K^2(m+n)^2)$, which is a significant improvement over the recomputation-based algorithm which runs in $\mathcal{O}(K^2(n^2m+nm^2))$ over a complete branch.

2.4 Dominance rules

In some cases, given a partial assignment with some rows and columns already included in the set variables $\mathcal{C}_k$ and $\mathcal{R}_k$, dominance rules permit to detect additional rows or columns that must be included in any optimal solution extending this partial assignment, or rows or columns that never participate in an optimal solution. The current state is defined by $(\mathcal{R}_k^{\in}, \mathcal{R}_k^{\perp}, \mathcal{C}_k^{\in}, \mathcal{C}_k^{\perp})$, and we denote the optimal solution extending this state as $(\mathcal{R}_k^{*\in}, \emptyset, \mathcal{C}_k^{*\in}, \emptyset)$ with $\mathcal{R}_k^{\in} \subseteq \mathcal{R}_k^{*\in}$, $\mathcal{R}_k^{*\in} \subseteq (\mathcal{R}_k^{\in} \cup \mathcal{R}_k^{\perp})$, $\mathcal{C}_k^{\in} \subseteq \mathcal{C}_k^{*\in}$, $\mathcal{C}_k^{*\in} \subseteq (\mathcal{C}_k^{\in} \cup \mathcal{C}_k^{\perp})$.

Theorem 2 gives the condition to be satisfied to detect that a row i should be included in submatrix l in any optimal solution extending the current state.

Theorem 2.

$$\forall i \in \mathcal{R}_l^{\perp} \;:\; \underset{row\ i}{\text{sum}} \left((\mathcal{C}_l^{\in} \cup \mathcal{C}_l^{\perp,-i}) \setminus (\bigcup_{k|k\neq l} \mathcal{C}_k^{\in,+i} \cup \mathcal{C}_k^{\perp,+i}) \right) > 0 \Rightarrow i \in \mathcal{R}_l^{*\in} \tag{10}$$

Proof. (sketch) Let us assume the worst-case scenario: despite selecting all the columns with negative values in this row i, while other submatrices would take the columns with positive values, the submatrix still has a positive sum contribution for this row i. Therefore this row must be included in submatrix l in any optimal solution extending the current state. □

Theorem 3 gives the condition to be satisfied to detect that a row i will never be included submatrix l in any optimal solution extending the current state, using the best-case scenario.

Theorem 3.

$$\forall i \in \mathcal{R}_l^{\perp} \;:\; \underset{row\ i}{\text{sum}} \left((\mathcal{C}_l^{\in} \cup \mathcal{C}_l^{\perp,+i}) \setminus (\bigcup_{k|k\neq j} \mathcal{C}_k^{\in,-i} \cup \mathcal{C}_k^{\perp,-i}) \right) < 0 \Rightarrow j \notin \mathcal{R}_l^{*\in} \tag{11}$$

These two properties (and their symmetric counterparts for columns) can be used in any node of the search tree to reduce the search space.

2.5 propagateDominanceRule: dominance rules check

Dominance rules from equations (10) and (11) (and their symmetric counterparts for the columns) can be used to reduce the search space. As in the previous subsections, recomputing the rules at each call to PROPAGATEDOMINANCERULE is expensive ($\mathcal{O}(Kmn)$ at each call, $\mathcal{O}(K^2(m^2n + mn^2))$ over a complete branch of the DFS). We describe below how to maintain the rules on rows. Of course, the method is symmetric for columns.

As in SELECTUNBOUNDSETVAR and SELECTVALUE, we maintain virtual FSMs for each triplet (row, column, submatrix), as shown in shown Fig.3. The FSMs collectively maintain two reversible values, shared between FSMs, for each (submatrix k, row i) table:

- $\text{lb}_{k,i}$ is the value of the worst-case scenario for submatrix k and row i (the left part of equation (10))
- $\text{ub}_{k,i}$ is the value of the best-case scenario for submatrix k and row i (the left part of equation (11))

The FSMs also maintain the number of *supports* of each cell (i, j), i.e. the number of submatrices that could still select the cell:

$$\text{support}_{i,j} = \left|\{k \mid i \in (\mathcal{R}_k^{\in} \cup \mathcal{R}_k^{\perp}) \wedge j \in (\mathcal{C}_k^{\in} \cup \mathcal{C}_k^{\perp})\}\right| \tag{12}$$

Each $\text{support}_{i,j}$, shared across all FSMs, is maintained as reversible integer by the solver: its state can then be backtracked. The transition and update operations

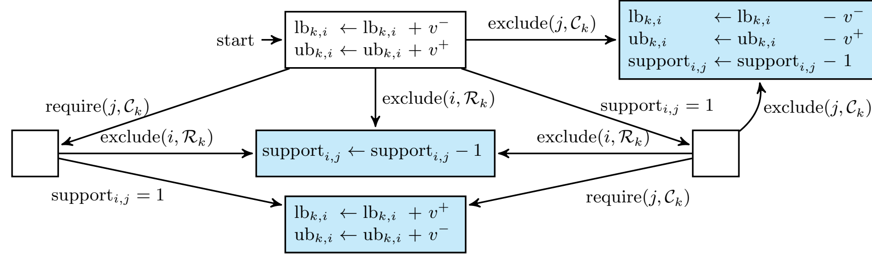

Fig. 3: FSM maintained for each (row, column, submatrix) i, j, k in PROPAGATEDOMINANCERULE. For simplicity, $v = M_{i,j}$, $v^+ = \max(v, 0)$ and $v^- = \min(v, 0)$. FSMs states in blue are terminal states.

of our FSMs are the following:

- When a row i (resp. column j) is excluded from a submatrix k, at most n (resp. m) cells' FSMs must be updated. The contribution of the cell (i, j) to $\text{ub}_{k,i}$ and $\text{lb}_{k,i}$ are removed and the support of the cell is decremented. Each of these operations are in constant time, and overall takes $\mathcal{O}(n)$ (resp. $\mathcal{O}(m)$).
- When a cell (i, j) becomes supported by only one remaining submatrix k ($\text{support}_{i,j} = 1$), and the column j is included in this submatrix k ($j \in \mathcal{C}_k^{\in}$, and since $\text{support}_{i,j} = 1$, it implies that $i \in (\mathcal{R}_k^{\in} \cup \mathcal{R}_k^{\perp})$), the value of lb and ub for this submatrix k is updated by the cell's value. This operation is also in constant time, and thus $\mathcal{O}(K)$ for all submatrices.

– When a row i (resp. column j) is included in a submatrix k, a check on all columns j (resp. rows i) must be performed to see if a cell (i,j) with $\text{support}_{i,j} = 1$ and $i \in \mathcal{R}_k^{\in}$ and $j \in \mathcal{C}_k^{\in}$ exists. If that is the case, $lb_{k,i}$ and $ub_{k,i}$ are updated to include the value of the cell. Overall, this operation is $\mathcal{O}(n)$ (resp. $\mathcal{O}(m)$).

Once the update of the FSMs is done, each (row, submatrix) pair is verified w.r.t. the rules, in $\mathcal{O}(Km)$. A call to propagateDominanceRule is in $\mathcal{O}(Km + \Delta_{\text{rows}} n + \Delta_{\text{cols}} m + \Delta_{\text{required}} K + \Delta_{\text{support}=1} K)$. Over a complete branch, the number of operations required is in $\mathcal{O}(Km^2 + Kmn)$. If the rules are applied symmetrically on columns, the overall running time is in $\mathcal{O}(K \max(m,n)^2)$.

2.6 updateBounds: efficient lower and upper bounds computations

In order to run the Branch & Bound, upper bounds on the objective for the current tree node must be computed efficiently. The chosen method also provides a lower bound, with no additional (asymptotic) computational cost.

The upper bound ub is the sum of every cell that is either selected in a submatrix or that is positive and could still be selected. The lower bound lb is similarly defined, but keeping negative-valued cells. Formally, they are computed as follows:

$$\text{ub} = \sum \{\mathcal{M}_{i,j} \mid (i,j) \in \text{Cover}^{\in} \vee (\mathcal{M}_{i,j} > 0 \wedge (i,j) \notin \text{Cover}^{\notin})\} \tag{13}$$

$$\text{lb} = \sum \{\mathcal{M}_{i,j} \mid (i,j) \in \text{Cover}^{\in} \vee (\mathcal{M}_{i,j} < 0 \wedge (i,j) \notin \text{Cover}^{\notin})\} \tag{14}$$

Recomputing these bounds from scratch in each node is again costly: $\mathcal{O}(Knm)$. The running time can be improved by maintaining incrementally the number of submatrices supporting each cell, in the same way as previously done in propagateDominanceRule.

These bounds, stored as reversible floating point numbers, can then be maintained easily:

– When a row i is included in a submatrix k, check if any column j is already in $\mathcal{C}_k^{\in}$, and that $(i,j) \notin \text{Cover}^{\in}$ yet. If that is the case and that $\mathcal{M}_{i,j} > 0$ (resp. < 0), increase ub (resp. lb) by $\mathcal{M}_{i,j}$. This operation runs in $\mathcal{O}(n)$.
– The similar operation must be performed when a column is included in a submatrix. Each of these operations runs in $\mathcal{O}(m)$.
– When a row i is excluded from a submatrix k, check if any column j is not already excluded ($j \notin (\mathcal{C}_k^{\in} \cup \mathcal{C}_k^{\perp})$). If that is the case, decrease $\text{support}_{i,j}$ by one. This operation runs in $\mathcal{O}(n)$.
– The same operation goes for excluded columns in $\mathcal{O}(m)$.
– When the $\text{support}_{i,j}$ is reduced to zero, if $\mathcal{M}_{i,j} > 0$ (resp. < 0), then decrease ub (resp. lb) by $\mathcal{M}_{i,j}$. This operation runs in $\mathcal{O}(1)$.

The whole maintenance process for the bounds behaves in $\mathcal{O}(\Delta_{rows} n + \Delta_{cols} m)$. Over a complete branch, the incremental method is in $\mathcal{O}(Knm)$, while the one based on recomputations is in $\mathcal{O}(K^2(n^2 m + nm^2))$.

2.7 The Large Neighborhood Search.

The exhaustive approach presented above eventually finds and proves the optimum value provided enough time is given. Unfortunately, the search space is so large that even for small matrices and a limited number of submatrices, it tends to quickly find a good solution but is not able to improve it. To overcome this limitation, we propose to embed the exhaustive CP search into a Large Neighborhood Search (LNS) [19]. LNS is a local search approach using CP to discover improvements around the current best solution:

- First the CP exhaustive search is used during a limited time, to discover an initial solution.
- For a given number of iterations, the CP exhaustive search is used again but this time with some variables partially fixed (fragment) as in the current best solution.

In addition, to limit the risk of having an iteration stuck for too long, we limit the DFS to 1000 *failures*.

The current best solution at iteration t has the form $((\mathcal{R}^{*\in}_{1,t}, \ldots, \mathcal{R}^{*\in}_{K,t}); (\mathcal{C}^{*\in}_{1,t}, \ldots, \mathcal{C}^{*\in}_{K,t}))$ We propose three different fragment selection heuristics (part of the solution to constrain when restarting the LNS for next iteration):

1. Select uniformly at random a subset of rows and columns in the set of lines and columns used by some submatrix: $R^p \subseteq (\bigcup_{k\in M^p} \mathcal{R}^{*\in}_{k,t})$, $C^p \subseteq (\bigcup_{k\in M^p} \mathcal{C}^{*\in}_{k,t})$, then for each submatrix, include the set of rows and columns intersecting with those sets: $\mathcal{R}^{\in}_{k,t+1} = \mathcal{R}^{\in}_{k,t} \cap R^p$, $\mathcal{R}^{\perp}_{k,t+1} = R \setminus \mathcal{R}^{\in}_{k,t+1}$ and similarly for columns.
2. A similar operator is defined with rows and columns selected inside the whole matrix: $R^p \subseteq R$, $C^p \subseteq C$. This allows for greater diversification, notably by allowing discovery of previously unselected rows/columns.
3. Selecting uniformly at random a subset of submatrices $M^p \subseteq \{1, \ldots, K\}$. For each of these submatrices, select at random different subsets of rows and columns $R^p_k \subseteq \mathcal{R}^{*\in}_{k,t}$, $C^p_k \subseteq \mathcal{C}^{*\in}_{k,t}$ that is constrained: $\mathcal{R}^{\in}_{k,t+1} = \mathcal{R}^{\in}_{k,t} \cap R^p_k$, $\mathcal{R}^{\perp}_{k,t+1} = R \setminus \mathcal{R}^{\in}_{k,t+1}$ and similarly for columns.

Empirical observations show that these three operators are complementary.

3 Experiments

This section describes experiments conducted to assess the performances of the proposed algorithms and to provide guidance on the selection of the appropriate solution. We first evaluate the methods on synthetic datasets, where the optimum is known, then on real datasets.

We compare our exhaustive CP and LNS methods against a greedy baseline approach, CP-Greedy, that solves at each step the maximal-sum submatrix (K=1) problem using the CP approach from [4]. This approach iteratively selects the next best submatrix, on a modified matrix in which the previously selected entries are set to 0 such that there is no incentive to select several times the

same (positive) entries. Each iteration is performed within $\frac{t^{\max}}{K}$ with $t^{\max}$ the allocated budget of time.

The implementation has been carried out on OscaR [16], using Java 1.8.0 (Hotspot VM) on an AMD Bulldozer clocked at 2.1GHz; one core and 3 Go of RAM per instance.

The source code is available here: `https://github.com/GuillaumeDerval/MWSCP`.

3.1 Synthetic Datasets

A synthetic dataset composed of 1,617 instances have been generated using a Python script (available on Zenodo [9]). For those, the optimal solution is known as they were all generated by implanting randomly K submatrices before adding some noise[1]. Table 1 describes parameter values considered in the generation.The parameters used to generate the instances are described in Table 1.

Approaches are compared using any-time profiles as described in Definition 3.

Definition 3. ***Any-Time Profile.*** *Let $f(a, i, t)$ be the objective value of the best solution found so far by an algorithm a for an instance i at time t. Let $t^{\max}$ be the provided budget of time before interrupting a run. Let f_i^* be the optimal solution for i if known (as is the case for synthetic data). The any-time profile of a is the solution quality $Q_a(t)$ of a on all instances as a function of time:*

$$Q_a(t) = \frac{1}{|i|} \sum_i \frac{f(a, i, t)}{\max(f(a_i^*, i, t^{\max}), f^*)} \text{ with } a_i^* = \underset{a}{\operatorname{argmax}}\, f(a, i, t^{\max}) \,. \tag{15}$$

Table 1: Parameters for the synthetic dataset generation

Parameter	Description	Values used
m, n	size of the matrix $\mathcal{M} \in \mathbb{R}^{m \times n}$	$(800, 200), (640, 250), (400, 400)$
K	number of submatrices	2, 4, 8
o	minimum overlap between submatrices (in % of cells)	0, 0.3, 0.6
σ	background noise variance (mean is 0)	0, 0.5, 1.0
r, s	size of submatrices (noisy, Gaussian with $\sigma = \frac{r \text{ or } s}{20}$)	(35, 70), (50, 50)
seed	seed for matrix generation	$[0, 9]$

Fig. 4 gives the any-time profiles of the CP-Greedy baseline method, along with CP-Exhaustive (the exhaustive process presented above) and CP-LNS. The results clearly illustrates the overall better performances of the CP-LNS whenever the computation time exceeds roughly 20 seconds.

Table 2a presents, for each parameter value considered in the synthetic data generation, the performances of the algorithms. Reported performances are computed as the average performance of each algorithm obtained before a certain limit of computation time.

Through analysis of the performances with respect to parameters' values, we observed that the major parameters are, in decreasing order of influence,

[1] Notice that the optimal solution may be slightly different than the implanted submatrices because of the noise addition.

the following: 1) the submatrices overlap, 2) K = the number of submatrices. The difficulty of reaching good solution increases quickly as the minimum overlap parameter increases until 50%, after which it decreases. Similarly, as the number of implanted submatrices increases, good solution quality becomes harder to grasp.

3.2 Real Datasets

We also experiment with non-synthetic datasets of several types (*olympic*, *migration*, *genes*) described in section 1.1. The results, presented in Table 2b, are similar to those obtained for synthetic datasets. CP-LNS is the best method on most datasets given 10 seconds of computation time, with two notable exceptions

Table 2: Comparison between CP-Greedy (GRE), CP-Exhaustive (EX) and CP-LNS (LNS). The table shows the $Q_a(t)$ for each algorithm a given a certain amount of time t (see equation (3)).

(a) Synthetic dataset

	10s			20s			100s			1080s		
Parameters	GRE	EX	LNS	GRE	EX	LNS	GRE	EX	LNS	GRE	EX	LNS
$\{m = 400, n = 400\}$	**0.70**	0.33	0.37	0.74	0.57	**0.76**	0.76	0.75	**0.95**	0.77	0.75	**0.97**
$\{m = 640, n = 250\}$	**0.71**	0.34	0.32	0.75	0.48	**0.79**	0.77	0.74	**0.95**	0.77	0.75	**0.97**
$\{m = 800, n = 200\}$	**0.73**	0.34	0.29	**0.77**	0.48	0.61	0.79	0.77	**0.94**	0.79	0.78	**0.96**
$K = 2$	**0.85**	0.78	0.32	0.85	**0.88**	0.83	0.85	0.90	**0.96**	0.85	0.91	**0.97**
$K = 4$	**0.72**	0.20	0.30	**0.77**	0.51	0.72	0.78	0.74	**0.94**	0.78	0.75	**0.96**
$K = 8$	**0.57**	0.03	0.36	**0.64**	0.13	0.61	0.68	0.62	**0.94**	0.68	0.62	**0.97**
$o = 0\%$	**0.58**	0.27	0.34	0.67	0.45	**0.71**	0.71	0.66	**0.97**	0.71	0.66	**0.98**
$o = 30\%$	**0.71**	0.34	0.31	**0.73**	0.50	0.69	0.75	0.75	**0.93**	0.75	0.76	**0.95**
$o = 60\%$	**0.85**	0.40	0.34	**0.86**	0.57	0.77	0.86	0.86	**0.94**	0.86	0.86	**0.97**
$\sigma = 0.0$	0.73	0.34	**0.78**	0.78	0.63	**0.80**	0.81	0.77	**0.98**	0.81	0.78	**1.00**
$\sigma = 0.5$	**0.72**	0.33	0.04	**0.75**	0.44	0.67	0.78	0.74	**0.94**	0.78	0.74	**0.97**
$\sigma = 1.0$	**0.69**	0.33	0.16	**0.73**	0.44	0.68	0.73	0.75	**0.93**	0.73	0.75	**0.94**
$\{r = 50, s = 50\}$	**0.71**	0.34	0.34	**0.75**	0.52	0.73	0.77	0.76	**0.94**	0.77	0.77	**0.96**
$\{r = 35, s = 70\}$	**0.71**	0.32	0.32	**0.76**	0.50	0.71	0.78	0.75	**0.95**	0.78	0.75	**0.97**

(b) Real datasets

$K = 4$		1s			5s			20s		
Type	Dataset	GRE	EX	LNS	GRE	EX	LNS	GRE	EX	LNS
migration	migration_0.001 [6]	**0.96**	0.92	**0.96**	0.96	0.92	**0.99**	0.96	0.92	**1.00**
migration	migration_0.003 [6]	0.87	0.89	**0.93**	0.87	0.89	**0.99**	0.87	0.89	**1.00**
migration	migration_0.005 [6]	0.83	0.79	**0.96**	0.83	0.79	**1.00**	0.83	0.79	**1.00**
olympic	olympic_0.01 [10]	0.88	0.69	**0.92**	0.88	0.91	**0.97**	0.91	0.91	**1.00**
olympic	olympic_0.02 [10]	0.79	0.69	**0.87**	0.84	0.84	**0.97**	0.84	0.84	**1.00**
olympic	olympic_0.04 [10]	0.62	0.81	**0.91**	0.76	0.82	**0.96**	0.93	0.82	**1.00**
olympic	olympic_0.06 [10]	0.80	0.92	**0.93**	0.97	0.92	**0.98**	0.97	0.92	**0.99**
$K = 4$		10s			20s			100s		
Type	Dataset	GRE	EX	LNS	GRE	EX	LNS	GRE	EX	LNS
gene	alizadeh-2000-v1_095 [20]	**1.00**	0.48	0.82	**1.00**	0.48	0.82	**1.00**	0.48	0.92
gene	armstrong-2002-v1_095 [20]	0.73	0.60	**0.92**	0.73	0.60	**0.99**	0.73	0.60	**1.00**
gene	bhattacharjee-2001_095 [20]	0.82	0.31	**0.98**	0.91	0.86	**0.99**	0.91	0.96	**1.00**
gene	bittner-2000_095 [20]	**0.96**	0.53	0.86	0.96	0.53	**0.98**	0.96	0.53	**0.98**
gene	bredel-2005_095 [20]	0.98	0.86	**1.00**	0.98	0.86	**1.00**	0.98	0.86	**1.00**
gene	chen-2002_095 [20]	0.74	0.80	**1.00**	0.89	0.80	**1.00**	0.89	0.80	**1.00**
gene	chowdary-2006_095 [20]	0.82	0.83	**1.00**	0.82	0.83	**1.00**	0.87	0.83	**1.00**
gene	dyrskjot-2003_095 [20]	0.97	0.94	**0.99**	0.97	0.94	**1.00**	0.97	0.94	**1.00**
gene	garber-2001_095 [20]	**0.59**	0.24	0.58	**0.82**	0.32	0.58	**1.00**	0.50	0.86
gene	golub-1999-v1_095 [20]	0.86	0.88	**0.92**	0.86	0.88	**0.95**	0.86	0.88	**0.96**

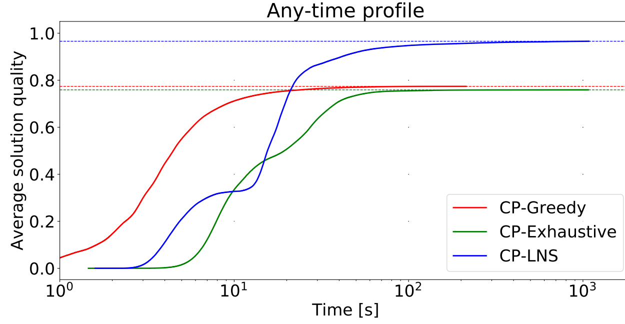

Fig. 4: Comparison between CP-Greedy, CP-Exhaustive and CP-LNS on $1,617$ matrices generated as described in section 3.1. The graph presents the any-time profile described in equation (3). For each instance, 18 minutes were allocated for computations.

(alizadeh and garber datasets), in which case LNS did not find the optimum in the 20 minutes allowed for each dataset.

3.3 Comparison Against Mixed Integer Linearly and Quadratically Constrained Programming

Table 3: Comparison between CP-LNS, MIP and MIQCP, on a synthetic dataset (generated as described in section 3.1). All methods were given a fixed time limit of 300 seconds. The metric used is the any-time profile at the time limit (see definition 3). CP-LNS finds the optimum on each dataset. The time when the best found solution was found is indicated inside parentheses. Experiments made on Gurobi 8.1.0.

(a) Varying number of submatrices and noise, with matrices of size 50×50 and submatrices of size 16×16.

K	σ	CP-LNS	MIP	MIQCP
2	0.0	**1.00** (1s)	**1.00 (0s)**	**1.00** (1s)
2	0.5	**1.00 (1s)**	**1.00** (7s)	**1.00** (7s)
2	1.0	**1.00 (1s)**	0.89 (233s)	0.79 (57s)
3	0.0	**1.00** (2s)	**1.00 (1s)**	**1.00** (2s)
3	0.5	**1.00 (3s)**	**1.00** (140s)	**1.00** (138s)
3	1.0	**1.00 (3s)**	0.74 (254s)	0.48 (256s)
4	0.0	**1.00** (2s)	**1.00 (1s)**	**1.00** (62s)
4	0.5	**1.00 (3s)**	**1.00** (252s)	0.88 (290s)
4	1.0	**1.00 (6s)**	0.64 (260s)	0.69 (225s)
5	0.0	**1.00 (4s)**	**1.00** (79s)	**1.00** (275s)
5	0.5	**1.00 (5s)**	0.82 (257s)	0.69 (237s)
5	1.0	**1.00 (6s)**	0.77 (24s)	0.36 (38s)

(b) Varying size of the matrix and noise, with matrices of size $m \times m$ and $K = 2$ submatrices of size $\lfloor \frac{m}{3} \rfloor \times \lfloor \frac{m}{3} \rfloor$.

m	σ	CP-LNS	MIP	MIQCP
50	0.0	**1.00 (0s)**	**1.00** (1s)	**1.00** (3s)
50	0.5	**1.00 (1s)**	**1.00** (5s)	**1.00** (7s)
50	1.0	**1.00 (1s)**	0.95 (207s)	0.82 (204s)
100	0.0	**1.00** (4s)	**1.00 (1s)**	1.00 (33s)
100	0.5	**1.00 (1s)**	0.86 (293s)	1.00 (45s)
100	1.0	**1.00 (3s)**	0.65 (269s)	0.82 (191s)
200	0.0	**1.00** (17s)	**1.00 (8s)**	**1.00** (135s)
200	0.5	**1.00 (21s)**	0.37 (191s)	3% (81s)
200	1.0	**1.00 (6s)**	0% (0s)	5% (134s)
400	0.0	**1.00 (1s)**	**1.00** (31s)	**1.00** (54s)
400	0.5	**1.00 (1s)**	0% (1s)	0% (0s)
400	1.0	**1.00 (1s)**	0% (1s)	4% (301s)

We tested our methods against MIP (linear) and MIQCP (quadratic terms in the constraints) methods. As these two methods do not perform well on bigger instances, we do not integrate them in our experiments on large matrices,

presented above.

MIP model		MIQCP model	
$\max \sum_{i,j} \mathcal{M}_{i,j} \cdot s_{i,j}$		$\max \sum_{i,j} \mathcal{M}_{i,j} \cdot s_{i,j}$	
$s_{i,j} \geq e_{i,j,k}$	$\forall i, j, k$	$K \cdot s_{i,j} \geq \sum_k r_{k,i} \cdot c_{k,j}$	$\forall i, j$
$s_{i,j} \leq \sum_k e_{i,j,k}$	$\forall i, j$	$s_{i,j} \leq \sum_k r_{k,i} \cdot c_{k,j}$	$\forall i, j$
$e_{i,j,k} + 1 \geq r_{k,i} + c_{k,j}$	$\forall i, j, k$		
$2 \cdot e_{i,j,k} \leq r_{k,i} + c_{k,j}$	$\forall i, j, k$		

All variables $\in \{0, 1\}$

MIP and MIQCP methods are plagued by the number of variables, that is in $\mathcal{O}(Knm)$ for MIP and $\mathcal{O}(K(n+m))$ for MIQCP, and by the number of constraints, which is $\mathcal{O}(Knm)$ for MIP and $\mathcal{O}(nm)$ for MIQCP. Tables 3a and 3b show that both models are slow compared to our LNS method, and are heavily affected by matrix size, number of submatrices to find and noise. For bigger submatrices, such as the synthetic and real ones presented in the previous section, both methods timeout either without returning solutions or with comparatively poor solutions.

4 Conclusions

We presented a generalization of the Maximal-Sum Submatrix Problem [4] to multiple submatrices, called the Maximum Weighted Submatrix Coverage Problem (MWSCP), along with a method to solve this problem based on constraint programming and large neighborhood search. Experiments on both synthetic and real datasets show that our CP-LNS method finds consistently better solutions (when more than 10 seconds are allocated) than both MIP/MIQCP, an exhaustive CP method and a greedy approach using the method from [4].

Acknowledgments Computational resources have been provided by the Consortium des Équipements de Calcul Intensif (CǵECI), funded by the Fonds de la Recherche Scientifique de Belgique (F.R.S.-FNRS) under Grant No. 2.5020.11.

References

1. Aoga, J.O., Guns, T., Schaus, P.: An efficient algorithm for mining frequent sequence with constraint programming. In: Joint European Conference on Machine Learning and Knowledge Discovery in Databases. pp. 315–330. Springer (2016)
2. Bentley, J.: Programming pearls: algorithm design techniques. Communications of the ACM **27**(9), 865–873 (1984)
3. Bessiere, C., De Raedt, L., Kotthoff, L., Nijssen, S., OSullivan, B., Pedreschi, D.: Data mining and constraint programming (2016)
4. Branders, V., Schaus, P., Dupont, P.: Mining a sub-matrix of maximal sum. In: Proceedings of the 6th International Workshop on New Frontiers in Mining Complex Patterns in conjunction with ECML-PKDD 2017 (2017)

5. Chabert, M., Solnon, C.: Constraint programming for multi-criteria conceptual clustering. In: International Conference on Principles and Practice of Constraint Programming. pp. 460–476. Springer (2017)
6. Dao, T., Docquier, F., Maurel, M., Schaus, P.: Global migration in the 20th and 21st centuries: the unstoppable force of demography (2018)
7. Duong, K.C., Vrain, C., et al.: Constrained clustering by constraint programming. Artificial Intelligence **244**, 70–94 (2017)
8. Gervet, C.: Interval propagation to reason about sets: Definition and implementation of a practical language. Constraints **1**(3), 191–244 (1997)
9. Guillaume, D., Vincent, B., Pierre, D., Pierre, S.: Synthetic dataset used in "the maximum weighted submatrix coverage problem: A cp approach" (Nov 2018). https://doi.org/10.5281/zenodo.1688740
10. IOC Research and Reference Service, The Guardian: Olympic sports and medals, 1896-2014. `https://www.kaggle.com/the-guardian/olympic-games`
11. Karp, R.M.: Reducibility among combinatorial problems. In: Complexity of computer computations, pp. 85–103. Springer (1972)
12. Kuo, C.T., Ravi, S., Vrain, C., Davidson, I., et al.: Descriptive clustering: Ilp and cp formulations with applications. In: IJCAI-ECAI 2018, the 27th International Joint Conference on Artificial Intelligence and the 23rd European Conference on Artificial Intelligence (2018)
13. Laurent Michel, Pierre Schaus, Pascal Van Hentenryck: MiniCP: A lightweight solver for constraint programming (2018), available from `https://minicp.bitbucket.io`
14. Le Van, T., Van Leeuwen, M., Nijssen, S., Fierro, A.C., Marchal, K., De Raedt, L.: Ranked tiling. In: Joint European Conference on Machine Learning and Knowledge Discovery in Databases. pp. 98–113. Springer (2014)
15. Madeira, S.C., Oliveira, A.L.: Biclustering algorithms for biological data analysis: a survey. IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB) **1**(1), 24–45 (2004)
16. OscaR Team: OscaR: Scala in OR (2012), available from `https://bitbucket.org/oscarlib/oscar`
17. de Saint-Marcq, V.L.C., Schaus, P., Solnon, C., Lecoutre, C.: Sparse-sets for domain implementation. In: CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS). pp. 1–10 (2013)
18. Schaus, P., Aoga, J.O., Guns, T.: Coversize: a global constraint for frequency-based itemset mining. In: International Conference on Principles and Practice of Constraint Programming. pp. 529–546. Springer (2017)
19. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: International conference on principles and practice of constraint programming. pp. 417–431. Springer (1998)
20. de Souto, M.C., Costa, I.G., de Araujo, D.S., Ludermir, T.B., Schliep, A.: Clustering cancer gene expression data: a comparative study. BMC bioinformatics **9**(1), 497 (2008)