Detecting and exploiting permutation structures in MIPs

Domenico Salvagnin

DEI, University of Padova, salvagni@dei.unipd.it

Abstract. Many combinatorial optimization problems can be formulated as the search for the best possible permutation of a given set of objects, according to a given objective function. The corresponding MIP formulation is thus typically made of an assignment substructure, plus additional constraints and variables (as needed) to express the objective function. Unfortunately, the permutation structure is generally lost when the model is flattened as a mixed integer program, and state-of-the-art MIP solvers do not take full advantage of it. In the present paper we propose a heuristic procedure to detect permutation problems from their MIP formulation, and show how we can take advantage of this knowledge to speed up the solution process. Computational results on quadratic assignment and single machine scheduling problems show that the technique, when embedded in a state-of-the-art MIP solver, can indeed improve performance.

1 Introduction

Many combinatorial optimization problems can be formulated as the search for the best possible permutation of a given set of objects, according to a given objective function. Without loss of generality, in this paper we will consider problems of the form:

$$\min f(\pi) \tag{1}$$

$$\pi \in \Pi^n \tag{2}$$

where Π^n is the set of all permutations π of the ground set $N = \{1, \ldots, n\}$. A natural way to formulate this class of problems within a mixed-integer linear programming (MIP) paradigm is to encode the permutation π by introducing n^2 binary variables x_{ij} and $2n$ linear constraints, obtaining the so-called assignment polytope. Finally, additional artificial variables y, together with the corresponding linking constraints, are usually introduced in the model in order to properly formulate the objective function f. Note that if the objective function is linear in x, we have the so-called linear assignment polytope, which is polynomially solvable. The problem is thus reformulated as

$$\min g(x, y) \tag{3}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \quad \forall j \in N \tag{4}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \quad \forall i \in N \tag{5}$$

$$Ax + By \geq b \tag{6}$$

$$x_{ij} \in \{0, 1\} \quad \forall (i, j) \in N^2 \tag{7}$$

$$y \geq 0 \tag{8}$$

It is important to note that, being a permutation problem, the structure of the formulation above is such that if all variables x are fixed, then it is always possible to compute in closed form the values of the artificial variables y, and thus to obtain a complete solution. In other words, once the permutation is known, there is no need to solve an optimization problem to compute its objective value. In particular, if it always possible to express y as a function Γ of x, such that $f(x) = g(x, \Gamma(x))$.

Many combinatorial optimization problems, as for example the quadratic assignment problem (QAP), the traveling salesman problem (TSP), and single machine scheduling, belong to the above class, although not all are always modeled in this way. For example, while the TSP is clearly a permutation problem, it is usually not formulated as such, so there is no explicit assignment substructure in the model.

Unfortunately, the permutation structure is generally lost when the model is flattened as a mixed integer program, and state-of-the-art MIP solvers do not take full advantage of it. For example, while it is always trivial to compute a feasible solution of a permutation problem, state-of-the-art MIP solvers occasionally find it very challenging to find one, even with their rich arsenal of primal heuristics. Even worse, while it is well-known that permutation problems are usually well-suited for local-search based metaheuristics, that are capable of finding near-optimal solutions with very modest computing times, MIP solvers have a very hard time in improving their first poor-quality solutions.

The main issue here is that current state-of-the-art MIP technology lacks a powerful modeling language based on global constraints, a tool which has long been standard in constraint programming [1]. As such, it is almost impossible for the modeler to pass high-level information to the solver, such as, for example, combinatorial substructures. Given the current state of things, it has become standard practice in MIP implementations to devise algorithms that basically try to *reverse-engineer* combinatorial substructures from a flat list of linear inequalities. For example, in [2] a procedure for detecting network structures was presented; such structure, when present, is then used to improve cutting plane separation. Unfortunately, while these procedures are usually cheap and effective, they are still heuristic in nature and can be fooled by the many transformations that are applied to a given MIP formulation in the preprocessing phase. Extending them to be completely preprocessing-invariant is often not done in practice for performance reasons (the resulting algorithms would be too slow), with the consequence that on some instances the substructure is not detected even if it is present.

While we cannot but share the lament for the current state of things, and encourage MIP vendors to invest into more powerful modeling interfaces to the solvers, as (partially) done, e.g., in the open solver SCIP [3], in the meantime we can still try to improve the situation for specific substructures, as is done for permutation problems in the present paper. Note that while permutation problems are usually solved with specialized codes, some challenging instances have indeed been solved with MIP technology, as for example in [17], so improving the performance of MIP solvers on this class of problems is of practical interest and can broaden the applicability of the MIP paradigm.

The outline of the paper is as follows. In Section 2 we describe a heuristic procedure to automatically detected permutation problems, while in Section 3 we show how to exploit the permutation structure to implement an efficient and general purpose primal heuristic based on local search. Two classes of permutation problems, that we used as benchmark, as described in Section 4. Computational results are given in Section 5, showing that the technique, when embedded in a of state-of-the-art MIP solver, can indeed improve performance, according to several performance measures. Conclusions and future directions of research are finally addressed in Section 6.

2 Detecting permutation structures

Detection of permutation problems is done in two steps:

1. in the first step we look for assignment polytopes, thus identifying the binary variables x that encode the permutation, and the corresponding assignment constraints;
2. in the second step we check that, once the variables x are fixed, the remaining variables y can indeed be computed in a straightforward way. In particular, this implies finding a topological order among variables y, such that we can compute them in one pass from left to right.

The first step is organized as a clustering algorithm. First, all constraints involving at least one non binary variable, that are not equalities and whose right-hand-size is different from 1 are removed, as they cannot be part of the assignment structure. The remaining constraints are then partitioned into clusters: each cluster can contain only constraints with the same number of variables and with pairwise disjoint support. Constraints are assigned to the first compatible cluster in a first-fit fashion; if no cluster is compatible, a new cluster is created with the current constraint in it. After the set of clusters Q has been initialized, we look for pairs (q_1, q_2) of matching clusters. Two clusters are matching if they

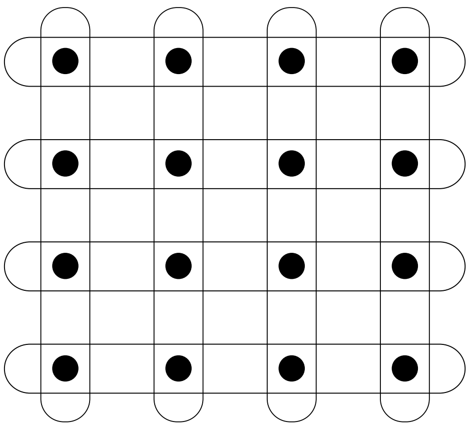

Fig. 1. Assignment structure.

cover exactly the same set of variables and each constraint in cluster q_1 intersects each constraint in cluster q_2 in exactly one variable. Intuitively, variables x are naturally double-index variables and can be arranged into a matrix, and the constraints in a pair of matching clusters (q_1, q_2) corresponds to the rows and columns, respectively, of this matrix, see Figure 1. Once a matching pair of cluster is found, it is removed from Q and the process continues until all pairs have been considered. Details are depicted in Algorithm 1. Note that the described detection algorithm is slightly more general than needed, since it can detect more than one assignment substructures.

The second step constructs a weighted dependency graph $G = (V, E, w)$ between the variables of the formulation, along with a topological order O on the nodes of G, and is based on row counts. At the beginning, all assignment variables x are added (in arbitrary order) as nodes to G, and to the ordered list O. All assignment constraints are removed from the model. Each constraint c_i left in the model, is assigned a count r_i, which counts all the variables in the constraint not yet in O. If some constraint ends up having a row count of zero at this step, then it means that this is not a pure permutation problem, but a constrained one, i.e., not all permutations are feasible since there are additional constraints on the assignment variables x: in this case we just abort the procedure. Then an iterative procedure begins, that loops over all constraints looking for those with $r_i = 1$. As soon as one is found, say c_i, the following steps are executed:

- let y_j be the variable left in c_i. All constraints involving y_j are considered and the singleton ones are collected in a set Y. Note that Y includes c_i.
- If $|Y| = 1$, then y_j can be computed directly from constraint c_i (plus its own bounds): y_j is added as a new node in G and to list O. In addition, edges are added to G, connecting y_j to all the other variables in c_i. Edges are weighted with the coefficients of the constraint, so that it is possible to

Algorithm 1: Assignment subproblems detection.

```
   Input: a list of constraints $C = \{c_1, \ldots, c_m\}$
   Output: a list of assignment substructures $A = \{A_1, \ldots, A_k\}$
   /* clustering */
1  $Q = \emptyset$;
2  foreach $c \in C$ do
3      foreach $q \in Q$ do
4          if $c$ is compatible with $c$ then
5              $q = q \cup \{c\}$;
6      if $c$ still not in a cluster then
7          $Q = Q \cup \{\{c\}\}$;
   /* matching clusters */
8  $A = \emptyset$;
9  for $q_1 \in Q$ do
10     for $q_2 \in Q$ do
11         if $q_1$ and $q_2$ are matching then
12             $A = A \cup \{(q_1, q_2)\}$;
13             $Q = Q \setminus \{q_1, q_2\}$;
14 return $A$
```

compute the correct value of y_j given the values of the variables on which it depends. Node y_j is marked as being of type `LINEAR`.

– If $|Y| > 1$, then y_j depends on more than one affine expression involving other variables. If those expressions are all consistent, then y_j can be computed as either a min or a max of them (plus its own bounds). If this is the case, dummy nodes are added to G and to list O, together with the corresponding edges, in order to encode those affine expressions, similarly to the previous case. Then a new node y_j is added to G and O, with edges connecting it to the dummy nodes. Node y_j is marked as being of type `MIN` or `MAX`, depending on the direction of the inequalities. If the constraints are not consistent, then it is not possible to trivially compute y_j from the preceding variables in the topological order, meaning that this is not a permutation problem. In this case we abort the procedure.
– Variable y_j is marked as done and all row counts of constraints in which y_j appears are decreased by 1. Constraints in Y, which by definition have now a row count of 0, are removed from the model.

At each iteration at least one constraint is removed from the model, so this phase terminates in $O(m)$ iterations, where m is the number of constraints in the formulation. The procedure can be implemented quite efficiently if the constraint matrix is stored both row and column-wise.

Note that if cutting planes are added to the original formulation by the modeler, then those will end up in the dependency graph and propagate; however, since they are not needed to get a correct model, they are redundant in the graph and contribute only as a slowdown factor. For these reason, it is convenient to exploit the facilities provided by the underlying solver, if any, to mark a subset of constraints as cuts, so that they can be ignored by the graph construction algorithm (for example, the `MPS` and `LP` file formats used by CPLEX allow this extension). A similar reasoning applies to indicator constraints: if explicitly marked as such, they can be handled more efficiently by the algorithm. For example, the linear expression need not be updated if the indicator variable is false.

Example 1. Let us consider the following artificial tiny permutation problem

$$\begin{aligned}
\min t \\
x_{11} + x_{12} = 1 \\
x_{21} + x_{22} = 1 \\
x_{11} + x_{21} = 1 \\
x_{12} + x_{22} = 1 \\
y_1 \geq 4x_{11} + 5x_{12} \\
y_2 \geq 3x_{21} + 2x_{22} + y_1 \\
t \geq y_1 + 2y_2 \\
t \geq 3y_1 + y_2 \\
x \in \{0,1\}^4 \\
y, w \geq 0
\end{aligned}$$

The corresponding dependency graph is depicted in Figure 2, and the ordered list is

$$O = [x_{11}, x_{12}, x_{21}, x_{22}, y_1, y_2, z_1, z_2, t]$$

□

3 Exploiting permutation structures

Once the permutation structure, if any, is identified, we can exploit its knowledge to improve the performance of the underlying MIP solver. A natural option, pursued in this paper, is to use the permutation structure to implement a general purpose primal heuristic, based on local-search. Alternative options, such as using the permutation structure for preprocessing and cut strengthening/separation, are possible as well, and left as future work.

The basic idea is to implement a local-search based metaheuristic, namely iterated local search (ILS) [4], using the dependency graph to explore neighborhoods and evaluate solutions. Given a permutation π, the neighborhood $\mathcal{N}(\pi)$ is defined as all permutations that can be obtained by swapping two elements of the permutation. Clearly, the neighborhood has polynomial size, containing exactly $n(n-1)/2$ permutations for each center π. The idea behind ILS is to perturb the current locally optimal solution s^* to get a new center t and call

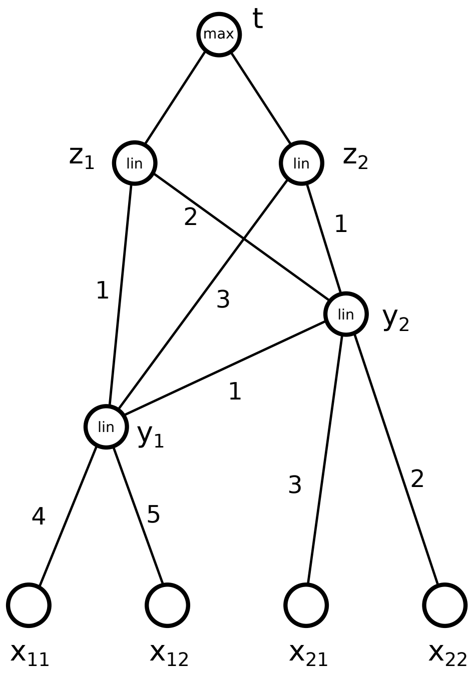

Fig. 2. Dependency graph of example problem.

again the local search procedure from there, obtaining a new local optimum t^*. If the new solution t meets an acceptance criterion, then t is chosen as the next starting point, otherwise it is rejected and the procedure is repeated from s^*. Intuitively, ILS implements a heuristic random walk on the set of locally optimal solutions of a given optimization problem. A high level pseudocode for ILS is given in Algorithm 2.

Algorithm 2: Basic ILS procedure

```
1 $s_0$ = GenerateRandomSolution ();
2 $s^*$ = LocalSearch ($s_0$);
3 repeat
4     $s'$ = Perturb ($s^*$, history);
5     $t$ = LocalSearch ($s'$);
6     $s^*$ = AcceptanceCriterion ($s^*$, $t$, history);
7 until termination condition;
```

Note that the perturbation mechanism and the acceptance criterion are in general dependent on the history of the system: this allows for more effective and elaborate strategies. The simplest, yet very common, acceptance criterion is to accept the new solution t if and only if its objective value is better than that of s. Other strategies include a pure random walk option, in which the new solution t is always accepted, regardless of its cost, and a simulated annealing [5,6] like acceptance criterion based on temperature, in which t is always accepted if it is an improving solution, but is also accepted with a given probability even if its objective value is worse (the probability is usually dependent on the "temperature" T of the system and on the difference between the two objective values, with slightly worsening steps being more likely). The first two strategies do not make use of the history of the system, while the third does. In our implementation, we chose the annealing criterion. As far as the perturbation step is concerned, a perturbed permutation π' is obtain from π by performing k swaps, where k is adjusted dynamically during the execution of the algorithm, and is always contained in the interval $\{k_1, \ldots, k_2\}$. Finally, as far as local search is concerned, we implemented a first-improving pivoting rule. The choice of ILS as a general purpose metaheuristic is motivated by the fact that it is relatively easy to implement and proved to be quite successful in many permutation problems, such as TSP [7], QAP [8], and scheduling problems [9].

Implementing local search on top of the dependency graph is quite straightforward. A first solution is evaluated by assigning a value to all variables x, and then computing the values of the variables y (and the intermediate expressions needed to evaluate max and min, if any) following the order in O. Then, whenever a swap is performed, the change in value of the 4 affected variables is propagated following the graph (much like in constraint propagation systems), thus achieving incremental evaluation.

4 Testbed

We considered two classes of problem that exhibit a permutation structure, namely quadratic assignment and single machine scheduling problems, as described in the next subsections.

4.1 Quadratic Assignment Problems

The NP-hard (and notoriously very difficult in practice) *Quadratic Assignment Problem* (QAP), in its Koopmans and Beckmann form [10], can be sketched as follows; see, e.g., [11] for details. We are given a complete directed graph $G = (V, A)$ with n nodes n^2 arcs, and a set of n facilities to be assigned to its nodes. In what follows, indices i, j correspond to nodes, indices u, v to facilities, $b_{ij} \geq 0$ is a given (directed) *distance* from node i to node j, and $a_{uv} \geq 0$ is a given required *flow* from facility u to facility v. By using binary variables $x_{iu} = 1$ iff facility u is assigned to node i, QAP can be stated as the following quadratic binary problem:

$$\min \textstyle\sum_{i=1}^{n} \sum_{u=1}^{n} \sum_{j=1}^{n} \sum_{v=1}^{n} a_{uv}\, b_{ij}\, x_{iu} x_{jv} \tag{9}$$

$$\textstyle\sum_{i=1}^{n} x_{iu} = 1 \quad \forall u \in N \tag{10}$$

$$\textstyle\sum_{u=1}^{n} x_{iu} = 1 \quad \forall i \in N \tag{11}$$

$$x_{iu} \in \{0,1\} \quad \forall (i,u) \in N^2 \tag{12}$$

Most MIP models for QAP work with additional 0-1 variables $y_{iujv} = x_{iu}x_{jv}$ that are used to linearize the quadratic objective function—the Adams-Johnson model [12] being perhaps the best-known such formulation. These kinds of models require $\Theta(n^4)$ variables and $\Theta(n^3)$ constraints, so they become huge even for medium-size instances, with unacceptable slowdowns in solving the LP relaxations during the branch-and-cut tree.

A different approach is to look for MILP models requiring just $O(n^2)$ variables and constraints. An obvious model is the MILP one credited to Kaufman and Broeckx [13] that requires the introduction of just n^2 additional (continuous) variables

$$w_{iu} = \left(\sum_{j=1}^{n} \sum_{v=1}^{n} a_{uv} b_{ij} x_{jv} \right) x_{iu} \tag{13}$$

which can be easily linearized with big-M coefficients. The corresponding MIP model reads

$$\min \sum_{i=1}^{n} \sum_{u=1}^{n} w_{iu} \tag{14}$$

$$\sum_{i=1}^{n} x_{iu} = 1 \quad \forall u \in N \tag{15}$$

$$\sum_{u=1}^{n} x_{iu} = 1 \quad \forall i \in N \tag{16}$$

$$w_{iu} \geq \sum_{j=1}^{n} \sum_{v=1}^{n} a_{uv} b_{ij} x_{jv} - M(1 - x_{iu}) \quad \forall (i,u) \in N^2 \tag{17}$$

$$x_{iu} \in \{0,1\} \quad \forall (i,u) \in N^2 \tag{18}$$

$$w_{iu} \geq 0 \tag{19}$$

This model is known to be of little use in practice *as is* because of the big-M constraints (17). In particular, it can be proved [14] that the root-node bound is always zero. However, a much stronger formulation can be obtained by adding to (14)−(19) the (polynomial) family of cutting planes

$$w_{iu} \geq L_{iu} x_{iu} \tag{20}$$

where each L_{iu} is defined as the optimal value of the linear (and polynomially solvable) assignment problem:

$$\min \sum_{j=1}^{n} \sum_{v=1}^{n} a_{uv} b_{ij} x_{jv} \tag{21}$$

$$\sum_{j=1}^{n} x_{jv} = 1 \quad \forall v \in N \tag{22}$$

$$\sum_{v=1}^{n} x_{jv} = 1 \quad \forall j \in N \tag{23}$$

$$x_{iu} = 1 \tag{24}$$

$$x_{jv} \in \{0, 1\} \quad \forall (j, v) \in N^2 \tag{25}$$

It can be shown that adding constraints (20) to the model, the resulting root relaxation bound is at least as strong as the so-called Gilmore-Lawler [15,16] bound. This lightweight model, together with the family of cutting planes (20), was used recently in [17] to solve highly symmetric QAP instances, proving to be a reasonable tradeoff between bound strength and enumeration speed.

In this paper, we considered all the instances in the standard QAPLIB [18] testbed with $n < 20$, and excluding the instances of the `esc` class, which are well-known to be massively symmetric. Overall, we are left with 31 instances.

As far as the structure of the problem is concerned, the quadratic assignment problem is clearly a permutation problem. Once all variables x_{iu} are assigned a value, then the value of variables w_{iu} is automatically derived by our algorithm as

$$w_{iu} = \max \left\{ \sum_{j=1}^{n} \sum_{v=1}^{n} a_{uv} b_{ij} x_{jv} - M(1 - x_{iu}), L_{iu} x_{iu}, 0 \right\} \tag{26}$$

As far as our ILS metaheuristic is concerned, evaluating a neighboring solution has cost $O(n^2)$, assuming that the constraints defining w_{iu} are dense (as is usually the case). While linear in the size of the model, this is suboptimal with respect to an ad-hoc and QAP-specific implementation, where a neighboring solution can be evaluated $O(n)$ arithmetic operations: this is the price to pay for a general purpose (and relatively simple) implementation, directly based on a linear formulation of the model. Note that in this particular case the issue could be solved by expressing constraints (17) as indicator constraints, as hinted at the end of Section 2. Indeed, whenever a variable x_{iu} flips value, we do not need to update $O(n^2)$ expressions, but only $O(n)$, since only n w_{iu} variables are nonzero in any solution, thus implementing a form of partial incremental propagation. When a non up-to-date variable w_{iu} needs to be evaluated, it is computed from scratch, which again can be done in $O(n)$ because only n x_{iu} variables are nonzero in any solution. Unfortunately, properly handling these cases complicates the implementation significantly, so we do not support it yet. Finally, we can obtain a small performance improvement by marking (20) as cuts, so that they are ignored by the algorithm.

4.2 (Weighted) Total Tardiness Minimization in Single Machine Scheduling

In single machine scheduling problems [19], we are given a set of n jobs, to be processed on a single machine, without preemption. Each job j is characterized by its processing time p_j, a due date d_j, and a nonnegative weight w_j. Different objective functions are of interest when solving single machine scheduling problems. In the present paper we will restrict to (weighted) total tardiness of the schedule.

Given a job j, its tardiness T_j is defined as

$$T_j = \max\{C_j - d_j, 0\} \tag{27}$$

where C_j is the completion time of job j. In the scheduling notation of [20], the variants considered in this paper are denoted as $1||\sum T_j$ and $1||\sum w_j T_j$, for the simple and weighted total tardiness, respectively.

For the unweighted case [19], a simple MIP model reads

$$\min \sum_{k=1}^{n} T_k \tag{28}$$

$$\sum_{j=1}^{n} x_{jk} = 1 \quad \forall k \in N \tag{29}$$

$$\sum_{k=1}^{n} x_{jk} = 1 \quad \forall j \in N \tag{30}$$

$$T_k \geq \sum_{j=1}^{n} p_j \left(\sum_{u=1}^{k} x_{ju} \right) - \sum_{j=1}^{n} d_j x_{jk} \quad \forall k \in N \tag{31}$$

$$x_{jk} \in \{0,1\} \quad \forall (j,k) \in N^2 \tag{32}$$

$$T_j \geq 0 \quad \forall j \in N \tag{33}$$

where variable $x_{jk} = 1$ iff job j is assigned position k in the processing, while T_k is the tardiness of the job in position k. In constraints (31), the first term is the cumulative processing time of the first k jobs in the sequence, while the second term is the due date of the job in position k. Note that in the model we do not explicitly keep track of the tardiness of each job by index, but only by position.

A MIP formulation for the weighted case is considerably more involved, because, differently from the unweighted case, we need to know the tardiness of each job by its index j and not just by its position k. Indeed, at least four formulations can be implemented, as surveyed in [21], with different tradeoffs between size and strength. In the present paper, we considered the one based on the assignment polytope, much in the spirit of the unweighted case. The MIP model reads

$$\min \sum_{j=1}^{n} w_j T_j \tag{34}$$

$$\sum_{j=1}^{n} x_{jk} = 1 \quad \forall k \in N \tag{35}$$

$$\sum_{k=1}^{n} x_{jk} = 1 \quad \forall j \in N \tag{36}$$

$$\gamma_1 \geq \sum_{j=1}^{n} p_j x_{j1} \tag{37}$$

$$\gamma_k \geq \gamma_{k-1} \sum_{j=1}^{n} p_j x_{jk} \quad \forall k \in N \setminus \{1\} \tag{38}$$

$$C_j \geq \gamma_k - M(1 - x_{jk}) \quad \forall (j,k) \in N^2 \tag{39}$$

$$T_j \geq C_j - d_j \quad \forall j \in N \tag{40}$$

$$x_{jk} \in \{0,1\} \quad \forall (j,k) \in N^2 \tag{41}$$

$$\gamma_k \geq 0 \quad \forall k \in N \tag{42}$$

$$C_j, T_j \geq 0 \quad \forall j \in N \tag{43}$$

As in the previous model, variable $x_{jk} = 1$ iff job j is assigned position k in the processing. In addition, γ_k is the completion time of the job in position k, C_j is the completion time of job j, and T_j is the tardiness of job j. As in the QAP case, the presence of big-M coefficients in constraints (39) makes the formulation quite weak in practice. However, by sorting the jobs by processing time before generating the model, it is possible to strengthen the model by adding a polynomial family of inequalities, which can be easily computed, see [21] for the details: we implemented this strengthened variant.

In this paper, we considered all the instances in the standard ORLIB [22] testbed with $n = 40$. Overall, there are 125 instances, and we consider both the unweighted and weighted variants (in the first case, by just ignoring the weights).

As far as the structure of the problem is concerned, with the chosen formulations the single machine scheduling problem is clearly a permutation problem. Once all variables x_{jk} are assigned a value, then we can automatically compute the value of variables T_k in the unweighted case, and of variables γ_k, C_j and T_j (in this order) in the weighted case.

As far as our ILS metaheuristic is concerned, evaluating a neighboring solution has cost $O(n)$ in the unweighted case, and $O(n^2)$ in the weighted case. Again, while linear in the size of the model, this is suboptimal in the weighted case, where an ad-hoc implementation can evaluate a neighboring solution in $O(n)$ arithmetic operations.

It is important to note that this slowdown is caused not by inefficiencies in the algorithm, but rather by the MIP formulation itself, because $O(n^2)$ linear constraints are needed to link variables γ_k and C_j. For example, if we were allowed to use nonlinear expressions, only $O(n)$ constraints of the form

$$C_j = \sum_{k=1}^{n} \gamma_k x_{jk}$$

would suffice. Note that the above constraints are essentially `element` [23] global constraints, so in the ideal case we should be able to formulate the problem using those, exploiting their presence to obtain a more efficient ILS implementation, and then let the solver automatically linearize them in order to get a standard MIP model. Thus, this is yet one more argument for implementing global constraint technology within MIP solvers.

5 Computational Experiments

We implemented our codes in C++, using IBM ILOG CPLEX 12.5.1 [24] as black box MIP solver. All tests have been performed on a cluster of identical PC, each with an Intel Xeon E3-1220 V2 CPU running at 3.10GHz and 16GB of RAM (only one CPU was used by each process). Each method was given a time limit of 10,000 seconds per instance.

We compare two variants of our state-of-the-art MIP solver on the instances described in the previous section. We denote with `detect` the full version of our code, which detects the permutation structure of the problems and applies the ILS heuristic throughout the branch-and-cut tree (implemented through the callback mechanism of CPLEX), and with `cpx` the same code with our detection algorithm disabled, in order to have a fair comparison between the two. As far as the parameters of `detect` are concerned, the detection algorithm is triggered at the end of the root node processing, and only if the integrality gap left is greater than 2%, in order to avoid wasting time in case of very easy instances. The default parameters of the ILS metaheuristic are $iterLim = 1000$ and $noImprovLim = 100$, while the perturbation interval $[k_1, k_2]$ is set to $[3, 7]$. If the integrality gap is less than 10%, then the ILS metaheuristic is run with stricter limits, namely $iterLim = 100$ and $noImprovLim = 20$. Finally, the heuristic is called every $10,000$ nodes and, if not effective for 5 times in a row, it is completely switched off for the rest of the search.

We use 4 performance measures to compare `cpx` and `detect`:

- `#solved`: number of instances solved within the time limit.
- `time`: shifted geometric mean, with a shift of 1 second, of the running time on the subset of instances solved by both methods.
- `pint`: shifted geometric mean, with a shift of 0.01, of the primal integral on the subset of instances solved by both methods (the lower the better). The primal integral [25] measures the overall behavior of the solver as far as the primal bound is concerned, and overcomes many shortcomings of traditional

figures when measuring the effect of primal heuristics. Given the primal gap function $\gamma(x)$ of a feasible solution x, defined as

$$\gamma(x) = \begin{cases} 0 & \text{if } cx = \overline{z} = 0 \\ 1 & \text{if } cx \cdot \overline{z} < 0 \\ \frac{|cx - \overline{z}|}{\max(|cx|, |\overline{z}|)} & \text{otherwise.} \end{cases}$$

where $\overline{z}$ is the value of the optimal (or best known) solution, we define the primal gap pgap as a function of the running time t as

$$\text{pgap}(t) = \begin{cases} 1 & \text{if no feasible solution is known at time } t \\ \gamma(\tilde{x}) & \text{if } \tilde{x} \text{ is the incumbent at time t.} \end{cases}$$

Note that $\gamma(x)$, and thus also pgap(t), is always between 0 and 1. The primal integral is defined as the integral over t of pgap(t).

– `gap`: average final integrality gap on the subset of instances unsolved by at least one method. The integrality gap is computed as

$$\text{gap}(\overline{z}, \underline{z}) = \begin{cases} 0 & \text{if } \overline{z} = \underline{z} = 0 \\ 1 & \text{if } \overline{z} \cdot \underline{z} < 0 \\ \frac{|\overline{z} - \underline{z}|}{\max(|\overline{z}|, |\underline{z}|)} & \text{otherwise.} \end{cases}$$

where $\overline{z}$ is the value of the global primal bound and $\underline{z}$ is the value of the global dual bound. This is the integrality gap as reported by many commercial solvers, and has the advantage of always being a number between 0 and 1.

Table 1. Comparison of the two methods.

testbed	method	#solved	time (s)	pint	gap
QAP	`cpx`	19	60.5	0.569	8.8%
	`detect`	20	55.5	0.113	8.3%
$1\|\|\sum T_j$	`cpx`	90	6.4	0.198	18.0%
	`detect`	88	5.3	0.128	17.6%
$1\|\|\sum w_j T_j$	`cpx`	40	12.8	2.140	31.8%
	`detect`	41	8.0	0.857	29.7%

Aggregated results of the comparison between the two methods are reported in Table 1. According to the table, the two methods are approximately equivalent as far as the number of solved instances is concerned: this is not surprising, as it is well-known that in general our ability in solving MIPs is largely dominated by the dual bound, which is not affected by our method. According to many computational studies [26,27], the effect of primal heuristics on the overall solution process is approximately in the order of 10-15% on average. Still, in our testbeds, where finding good quality solutions is challenging for CPLEX, the effect of our method is a significant reduction of the overall running time, up to almost 40% on the weighted total tardiness testbed. In addition, the primal integral is also significantly reduced, dropping by a factor of 4 in the QAP testbed and by a factor of approximately 2 on the single scheduling instances. Finally, on the unsolved instances, the final integrality gap was also consistently reduced—although by a little amount.

6 Conclusions

We described a heuristic procedure to automatically detected permutation problems, and exploited the permutation structure to implement an efficient and general purpose primal heuristic based on local search. Computational experiments on two classes of permutation problems, namely QAP and single machine scheduling, showed that the technique, when embedded in a of state-of-the-art MIP solver, can indeed significantly improve performance.

Future research includes efficiently implementing the extensions needed to support indicator constraints, as well as devising heuristic procedures to detect other common substructures, such as, for example, linear encodings of the `element` global constraint. This would bring the general purpose ILS procedure inline with the problem specific implementations. Extensions to more general classes of permutation problem, such as rectangular or higher-dimensional assignments, could also broaden the applicability of the method.

References

1. Gent, I.P., Petrie, K.E., Puget, J.F.: Symmetry in constraint programming. In Rossi, F., van Beek, P., Walsh, T., eds.: Handbook of Constraint Programming. Elsevier (2006) 329–376
2. Achterberg, T., Raack, C.: The MCF-separator: detecting and exploiting multi-commodity flow structures in MIPs. Mathematical Programming Computation **2**(2) (2010) 125–165
3. Achterberg, T.: SCIP: solving constraint integer programs. Mathematical Programming Computation **1**(1) (2009) 1–41
4. Lourenço, H.R., Martin, O.C., Stützle, T.: Iterated local search: Framework and applications. In Glover, F., Kochenberger, G., eds.: Handbook of Metaheuristics. Volume 57. Kluwer Academic Publishers (2002) 321–353
5. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science **220** (1983) 671–680
6. Černý, V.: Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. Journal of Optimization Theory and Applications **45**(1) (1985) 41–51
7. Johnson, D.S., McGeoch, L.A.: Experimental analysis of heuristics for the STSP. In Gutin, G., Punnen, A., eds.: The Traveling Salesman Problem and its Variations. (2002) 369–443

8. Stützle, T.: Iterated local search for the quadratic assignment problem. European Journal of Operational Research **174**(3) (2006) 1519–1539
9. Congram, R.K., Potts, C.N., van de Velde, S.L.: An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem. INFORMS Journal on Computing **14**(1) (2002) 52–67
10. Koopmans, T., Beckmann, M.: Assignment problems and the location of economic activities. Econometrica **25** (1957) 53–76
11. Burkard, R., Dell'Amico, M., Martello, S.: Assignment Problems. SIAM (2009)
12. Adams, W., Johnson, T.: Improved linear programming-based lower bounds for the quadratic assignment problem. In: Proceedings of the DIMACS Workshop on Quadratic Assignment Problems. Volume 16 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science., American Mathematical Society (1994) 43–75
13. Kaufman, L., Broeckx, F.: An algorithm for the quadratic assignment problem using Benders' decomposition. European Journal of Operational Research **2** (1978) 204–211
14. Xia, Y., Yuan, Y.: A new linearization method for quadratic assignment problem. Optimization Methods and Software **21** (2006) 803–816
15. Gilmore, P.: Optimal and suboptimal algorithms for the quadratic assignment problem. SIAM Journal on Applied Mathematics **14** (1962) 305–313
16. Lawler, E.: The quadratic assignment problem. Management Science **9** (1963) 586–599
17. Fischetti, M., Monaci, M., Salvagnin, D.: Three ideas for the quadratic assignment problem. Operations Research **60**(4) (2012) 954–964
18. Burkard, R., Karisch, S., Rendl, F.: QAPLIB – A quadratic assignment problem library. European Journal of Operational Research **55** (1991) 115–119
19. Baker, K.R., Trietsch, D.: Principles of Sequencing and Scheduling. Wiley (2009)
20. Graham, R., Lawler, E., Lenstra, J., Kan, A.R.: Optimization and approximation in deterministic sequencing and scheduling: A survey. Annals of Discrete Mathematics **5** (1979) 287–326
21. Keha, A.B., Khowala, K., Fowler, J.W.: Mixed integer programming formulations for single machine scheduling problems. Computers & Industrial Engineering **56**(1) (2009) 357–367
22. Beasley, J.E.: OR-library: distributing test problems by electronic mail (1990)
23. Hentenryck, P.V., Carillon, J.P.: Generality versus specificity: an experience with AI and OR techniques. In: AAAI-88. (1988)
24. IBM ILOG: CPLEX 12.5.1 User's Manual. (2013)
25. Berthold, T.: Measuring the impact of primal heuristics. Operations Research Letters **41** (2013) 611–614
26. Achterberg, T.: Constraint Integer Programming. PhD thesis, Technische Universität Berlin (2007)
27. Achterberg, T., Wunderling, R.: Mixed integer programming: Analyzing 12 years of progress. In: Facets of Combinatorial Optimization. (2013) 449–481