Metaheuristic approaches based on neighborhood search escape local optimality by applying predefined rules and constraints, such as tabu restrictions (in tabu search), acceptance criteria (in simulated annealing), and shaking (in VNS). We propose a general approach that attempts to learn (offline) the guiding constraints that, when applied online, will result in effective escape directions from local optima. Given a class of problems, the learning process is performed offline and the results are applied to constrained neighborhood searches to guide the solution process out of local optimality. Computational results on the Constrained Task Allocation Problem (CTAP) show that adding these guiding constraints to a simple tabu search improves the quality of the solutions found, making the overall method competitive with state-of-the-art methods for this class of problems. We also present a second set of tests on the matrix bandwidth minimization problem.

Keywords: Data mining, offline learning, neighborhood search, metaheuristic optimization

Last Revision: January 14, 2011

1. Introduction

Heuristic and metaheuristic techniques (Glover and Kochenberger 2003) are widely used for finding high quality solutions to large combinatorial optimization problems, for which identifying the optimal solution through an exact procedure would be computationally impractical. Some metaheuristic approaches are based on maintaining a set of solutions and creating new ones by way of executing combination procedures. Others, however, use a neighborhood search (NS) to move from one point to another in the solution space. The neighborhood of a solution is defined by the move mechanisms that are applied to transform one solution into another. Local search is the simplest form of a NS procedure, which limits the moves to those that direct the search from the current solution to a neighboring solution with a better objective function value. In other words, a local search accepts only improving moves to transform the current solution into one that is in its neighborhood. The process stops when no improving move is available and the resulting solution is called a local optimum (with respect to the defined neighborhood).

Metaheuristic procedures include strategies that aim at escaping from local optima with the goal of improving the best solution found during the search. In tabu search, escaping local optima is achieved by imposing tabu restrictions (Glover and Laguna 1997) or executing moves that are guided by proxy objective functions, as in path relinking (Glover, Laguna, and Marti 2003). Simulated annealing relies on randomization to allow non-improving moves to be executed in order to escape local optimal points.
Greedy Randomized Adaptive Search Procedures (GRASP) restart the search after reaching a local optimum. A common aspect to these strategies is that escape directions are determined by rules that are defined a-priori. For instance, tabu activation rules forbid certain moves by restricting a subset of attributes from assuming certain values for a predetermined number of iterations. In path relinking, only moves that take the current solution “closer” to a guiding solution are considered. In general, escaping from local optima in a NS-based approach requires the execution of a non-improving move and the reduction of the neighborhood. Often, the neighborhood reduction is achieved by enforcing guiding constraints. These guiding constraints are the result of design decisions that may or may not be customized to a particular class of problems. For instance, a tabu search approach for permutation problems may use swaps to explore the neighborhood of a given solution. After a swap of two elements, a tabu restriction could be imposed to forbid the exchange of positions of these two elements for a pre-specified number of iterations. This tabu-activation rule effectively restricts the available moves and hence shrinks the neighborhood of the current solution. In Variable Neighborhood Search (VNS), a metaheuristic methodology suggested by Mladenovic and Hansen (1997), escaping from local optima requires the systematic change from one neighborhood to another. This is achieved by embedding a set of move mechanisms of various degrees of complexity. Simple moves are tried first, followed by more complex moves as the search becomes trapped in local optima corresponding to the neighborhood defined by the moves currently under consideration. Both the set of moves (and therefore neighborhoods) to be explored and the order in which they will be considered are specified a-priori, a task that often requires a considerable effort in the development phase of the solution procedure.

Our goal is to define a general learning model, valid for any class of problems, whose output is the definition of the best set of guiding constraints and the conditions for their applications. The learning is performed offline by analyzing pairs of local optima drawn from a training set of problem instances belonging to a class under consideration. The learning process focuses on formulating constraints that guide the search in effective directions to escape local optimality. Effectiveness is measured by the ability to improve the current position after executing non-improving moves that are necessary to leave the local optimal point. We show how the rules learned during the offline process can be embedded in search procedures to be used online for tackling instances of the same class of problems. The process requires the following four elements:

1. A procedure to randomly generate feasible solutions for a given problem instance
2. A set of basic moves that define the neighborhood search
3. An attribute representation of a solution
4. An attribute representation of a pair of solutions

where the “attribute representation” of an object is a numerical vector whose components characterize the object.

Some work has already been done to couple machine learning and data mining (DM) techniques in meta-heuristic searches, as reported by Boyan, Buntine and Jagota (2000) and Jourdan, Dhaenens and Talbi (2006). In particular, the existing learning methods can be classified into two categories: online and offline methods. Both of these approaches are learning procedures that — after a training phase —
are used within a search process in an attempt to improve its effectiveness. The main difference between the two approaches lies on the training: online methods are trained using information collected during the current execution of the search procedure, while offline methods learn from a training set of instances of the same class of problems. Online procedures are used, for instance, to select a branch in a branch-and-bound algorithm given information gathered during the search itself, as in (Glover, Klingman, and Phillips 1989), or to find patterns useful to enhance the construction phase of GRASP, as proposed by Santos et al. (2008). On the other hand, offline procedures attempt to identify structures and patterns shared by the entire class of problems. In the literature, offline approaches have been further divided into two categories, depending on their goal: 1) algorithm-selection approaches and 2) parameter-tuning approaches. Algorithm-selection approaches use historical performance data with the goal of selecting the most promising procedure from a portfolio of available ones. Parameter-tuning approaches have the goal of finding the set of parameter values that will make a particular procedure perform at the highest level. Our work is an offline approach that cannot be directly cast as an algorithm-selection or parameter-tuning approach. We argue that our approach defines a new category of offline methods that may be referred to as data mining driven neighborhood search (DMDNS). This category refers to search procedures that modify neighborhoods with constraints that have been learned offline by mining data. Padmanabhan and Tuzhilin (2003) point out that this area has been largely neglected in the literature. Broadly speaking, DMDNS consists of two components: 1) an offline learning procedure that learns the guiding rules for escaping local optimality, and 2) a procedure that applies the guiding rules online. The offline learning procedure may be viewed as a process that is in some ways similar to fitness landscape analysis (Höhn and Reeves 1996). Both approaches attempt to characterize the search space. However, while the results of fitness landscape analysis are interpreted by the analysis in order to design global search strategies (e.g., effective neighborhood functions), the results of a learning procedure such as the one we propose are directly embedded in the neighborhood search in the form of constraints that are activated upon reaching local optima. In this work, we propose a data mining driven tabu search (DMDTS) and we show that our approach leads to improved escape directions when compared to those resulting from enforcing customary tabu search rules. We intend to establish a new research avenue in the development of learning procedures whose purpose is to capture aspects of the structure of a given class of problems by considering a “training set” of instances. The approach focuses on learning “off line” and only once for a given class of problems.

We will be using the Constrained Task Allocation Problem (CTAP) as an example throughout the paper in order to enhance the presentation of our concepts. We then apply the method to the Matrix Bandwidth Minimization Problem (MBMP), in an attempt to show its general application. We realize, however, that to claim success in a variety of problem settings, additional applications are required. We hope that our work inspires others to follow this line of research and expand the applicability of the concepts that we introduce.

2. Learning Procedure

Our goal is to develop a process that learns how to move from a current local optimum to a better local optimum. The first step consists of constructing a large set of local optima relative to a set of basic
moves. The set of local optima is constructed by generating random initial solutions and applying a pure local optimizer (based on the predefined basic moves). A training set of instances belonging to the class of problems under consideration is used for this purpose. For brand new practical problems, for which the availability of data may be limited, a training set could be constructed by generating instances with similar characteristics as the ones observed in practice. In fact, Section 5.3 shows that a randomly generated training set can be used effectively to capture characteristics of the real-world instances, suggesting that as long as the main characteristics are preserved, the information on the problem structure can be effectively learned from artificially created instances.

The second step consists of building the paired-data set that will be the input to the data mining learning procedure. For each local optimum $A$, we find all “close” local optima $B$ to construct pairs of the form:

\[\{(A,B) \mid B \in \mathcal{C}(A), A \in \mathcal{L}, B \in \mathcal{L}, A \neq B\}\]

where $\mathcal{L}$ is the local optima set and $\mathcal{C}(A)$ denotes the set of local optima close to $A$. Proximity between the two solutions is calculated by an appropriate distance measure (e.g., Euclidean) and “closeness” is determined with a threshold. Furthermore, we only keep those local optima $A$ that have at least a predefined number of close local optima $B$. Without this policy, the learning procedure would reduce to individuating the local optima that are “easiest” to improve in few moves, rather than individuating escape directions. Section 4 shows how this is implemented in the context of the CTAP. A pair $(A, B)$ is called an improving pair if the value of the objective function improves when moving from $A$ to $B$ and a non-improving pair otherwise. Let $\mathcal{I}$ be the set of pairs of local optima corresponding to improving pairs and $\mathcal{K}$ the set of pairs of local optima corresponding to non-improving pairs, then:

\[\mathcal{I} = \{(A,B) : B \in \mathcal{C}(A); A \in \mathcal{L}; B \in \mathcal{L}; f(A) > f(B)\}\]
\[\mathcal{K} = \{(A,B) : B \in \mathcal{C}(A); A \in \mathcal{L}; B \in \mathcal{L}; f(A) \leq f(B)\}\]

where $f(x)$ denotes the objective function value of solution $x$. Note that we are assuming a minimization problem and therefore the improving direction decreases the value of the objective function.

Our methodology requires that the representation of each pair of local optima be formed by $\vec{A}$ (a vector representing the attributes for $A$) and $\vec{A}_{AB}$ (a vector, which we will refer to as the local optimum difference, representing the changes to apply in order to move from $A$ to $B$). Note that the intention is for this representation to be independent of the size of the problem. That is, the number of attributes remains the same regardless of the size of the specific problem instance. As typically done in data mining, the pairs translate into a single vector of numerical values, which is the concatenation of $\vec{A}$ and $\vec{A}_{AB}$. As expected, the attributes chosen to represent a solution are problem-dependent. Therefore, a procedure that creates $\vec{A}$ from $A$ and $\vec{A}_{AB}$ from $A$ and $B$ is necessary to apply the method that we are proposing. This procedure should be carefully designed, keeping in mind that the combination of $\vec{A}$ and $\vec{A}_{AB}$ should enable the learning process. For instance, attributes of $\vec{A}_{AB}$ that contain information about the objective function may result in trivial rules, such as one that states that “in order to find an improving pair of local optima, the search directions should be restricted to those that eventually will
improve the objective function.” Section 4 provides an example of selecting attributes in the context of the constrained task allocation problem. Once the paired-data set is produced, the learning process starts. The following definitions facilitate the description of the learning procedure:

- **Triggering Condition**: A Boolean function $F$ that, given an $\vec{A}$ vector representing a local optimum $A$, produces an output such that $F(\vec{A}) = 1$ indicates that $A$ satisfies condition $F$. We refer to a triggering condition just as “condition” when the context is clear.

- **Guiding Constraint**: A Boolean function $G$ that, given a $\vec{A}_B$ vector representing a local optimum difference between $A$ and $B$, produces an output such that $G(\vec{A}_B) = 1$ indicates that the difference between the two local optima satisfies the guiding constraint $G$.

- **Guiding Rule**: A pair $(F, G)$ composed of a triggering condition $F$ and a guiding constraint $G$ such that if $F(\vec{A}) = 1$ then there exists at least one improving pair $(A, B)$ such that $G(\vec{A}_B) = 1$ and there does not exist a non-improving pair $(A, B)$ such that $G(\vec{A}_B) = 1$.

The methodology is based on the assumption that guiding rules that are learned from the training set can be generalized to new instances. In other words, if a local optimum $A$ of a new problem instance satisfies condition $F$, then there probably exists a better local optimum $B$ — i.e., $(A, B)$ is an improving pair — such that $\Delta_{AB}$ satisfies the guiding constraint $G$. This indicates that the guiding constraint $G$ must be enforced every time the search encounters a local optimum satisfying condition $F$, as described in Section 3. In the remaining, we refer to coverage of rule $r$ as the proportion of local optima $A$ in the paired-data set that satisfy the rule.

The learning process can be formulated as a mathematical programming problem, whose output is a set of guiding rules of the form $\{ (F_r, G_r), r = 1, \ldots, R \}$, where $R$ is a given value. However, before formally introducing the mathematical model, it is useful to describe briefly the basics of the binary classification applied in our method. Binary classification is a well-known data mining technique that has been broadly studied in the literature because of its numerous practical applications (Witten and Frank 2005). It consists of discriminating between two (positive and negative) classes of objects of the same domain. For instance, in medical applications, patients (i.e., objects) may be classified as sick (positive) or healthy (negative), where each patient is represented by a vector of real numbers. A binary classifier is an “entity” that performs the binary classification through training and then prediction. During training, the binary classifier analyzes a set of objects (training set) where the class of each object is known and “learns” the difference between the two classes. The output of the training procedure is a classification rule. During the prediction procedure, the binary classifier uses this classification rule to predict the unknown class of a new object.

In our method, every Boolean function $F$ and $G$ is associated with a binary classifier — thus we have $2R$ binary classifiers in total. For each rule $r$, the classifier associated with $F_r$, denoted by $C^F_r$, discriminates between local optima satisfying $F_r$ (positive class) and not satisfying $F_r$ (negative class). Similarly, for each rule $r$, the classifier associated with $G_r$, denoted by $C^G_r$, distinguishes between local-optimum differences (i.e., $\Delta_{AB}$) satisfying $G_r$ (positive class) and not satisfying $G_r$ (negative class).
Contrary to traditional binary classification, where in the training phase the class of each object is given, here the class of each object (local optimum or local optimum difference) has to be assigned. In other words, for each classifier $C^f_r$ (or $C^g_r$) we seek a partition of local optima (or local optimum differences) into two classes, such that the classification rule found by $C^f_r$ (or $C^g_r$) classifies them in the correct class. These classifiers are then used during a neighborhood search procedure that verifies whether a local optimum satisfies a condition in order to enforce the appropriate guiding constraint.

We use the hyperplane-based classifiers defined by Glover (1990), which are well known in the operations research community. He proposed the use of a hyperplane, identified by a vector $\vec{x}$ and a scalar $b$ (also called parameters of the classifier), to discriminate between two classes of objects, in such a way that the objects belonging to the positive class lie on the positive side of the hyperplane and the ones belonging to the negative class lie on the negative side of the hyperplane. An object $\vec{p}$ lies on the positive side if $\vec{x} \cdot \vec{p} > b$ and on the negative side if $\vec{x} \cdot \vec{p} < b$. Usually the mathematical model forbids points to lie too close to the hyperplane by introducing a separation zone of width $2\varepsilon$, as in Better et al. (2010). The $C^f$ classifiers classify local optima; therefore the length of the associated $\vec{x}$ vectors is equal to the number of attributes needed to represent a local optimum ($l^f$). The $C^g$ classifiers classify local-optimum differences; therefore the length of the associated $\vec{x}$ vectors is equal to the number of attributes needed to represent a local optimum difference ($l^g$).

The mathematical formulation employs the following sets of variables:

\[
f(A, r) = \begin{cases} 
1 & \text{if } A \text{ satisfies } F_r \text{ for } A \in L \text{ and } r = 1, \ldots, R \\
0 & \text{otherwise}
\end{cases}
\]

\[
g(A, B, r) = \begin{cases} 
1 & \text{if } \Delta_{AB} \text{ satisfies } G_r \text{ for } A \in L; \ B \in C(A) \text{; } r = 1, \ldots, R \\
0 & \text{otherwise}
\end{cases}
\]

\[
p(A, B) = \begin{cases} 
1 & \text{if } \Delta_{AB} \text{ satisfies at least one } G_r \forall \ (A, B) \in \mathcal{S} \\
0 & \text{otherwise}
\end{cases}
\]

\[
(\vec{x}_{F_r}, b_{F_r}) = \text{coefficients associated with the } C^f_r \text{ classifier, } r = 1, \ldots, R
\]

\[
(\vec{x}_{G_r}, b_{G_r}) = \text{coefficients associated with the } C^g_r \text{ classifier, } r = 1, \ldots, R
\]

The mathematical model is:

\[
\text{maximize } \sum_{(A, B) \in \mathcal{S}} p(A, B)
\]

subject to

\[
\sum_{B: (A, B) \in \mathcal{S}} g(A, B, r) \geq f(A, r) \quad A \in L; \ r = 1, \ldots, R \tag{1}
\]

\[
g(A, B, r) \leq 1 - f(A, r) \quad A \in L; \ B: (A, B) \in \mathcal{S}; \ r = 1, \ldots, R \tag{2}
\]

\[
g(A, B, r) \leq f(A, r) \quad A \in L; \ B: (A, B) \in \mathcal{S}; \ r = 1, \ldots, R \tag{3}
\]

\[
\sum_{r=1}^{R} g(A, B, r) \geq p(A, B) \quad \forall \ (A, B) \in \mathcal{S} \tag{4}
\]
The objective function is to maximize the number of improving pairs that satisfy a guiding constraint. When an improving pair \((A,B)\) satisfies a guiding constraint, a way to improve \(A\) has been found, and therefore we would like to maximize the number of times that \(AB\)-pairs satisfy guiding constraints. In the formulation, (1) and (2) enforce the guiding rule definition, i.e., if \(A\) satisfies condition \(F_r\), then there must be at least one improving pair \((A,B)\) and no non-improving pair \((A,B)\) satisfying guiding constraint \(G_r\). Constraint set (3) strengthens the guiding rule by forcing \(A\) to satisfy condition \(F_r\) if there is an improving pair \((A,B)\) that satisfies the guiding constraint \(G_r\). If an improving pair \((A,B)\) satisfies at least one guiding constraint \(G_r\), then constraint set (4) allows this pair to be counted toward the objective function value.

Constraints (5) and (6) force \(f(A,r) = 1\) if and only if \(\vec{A}\) lies on the positive side of the hyperplane defined by \((\vec{x}_{F_r}, b_{F_r})\). Constraints (7) and (8) force \(g(A,B,r) = 1\) if and only if \(\vec{A}_{AB}\) lies on the positive side of the hyperplane \((\vec{x}_{G_r}, b_{G_r})\). In these constraints, \(M\) is a conveniently large positive number, necessary to satisfy the inequalities corresponding to the side of the hyperplane where \(\vec{A}\) (or \(\vec{A}_{AB}\)) does not lie. We set \(\epsilon\) equal to 0.01, to force some minimum separation from the classification point and the hyperplane. The model ends with the set of integrality restrictions and bounds for the hyperplane coefficients, which we bound between -1 and 1. Note that changing these bounds would just scale up or down all hyperplane coefficients by the same factor, leading to the same classification rules. The value of \(M\) would need to be changed accordingly, in order to keep constraints (5) to (8) valid.

The classification rule embedded in the hyperplane-based classifiers is an inequality involving a linear combination of the attributes of local optima or their differences. Depending on the problem context, the attributes may be expressed in more than one unit of measure or dimension. For instance, in section 4, we apply our methodology to a problem in which we represent local optima using attributes that are measures in three different units. A linear combination involving heterogeneous dimensions (units) is meaningless and the classification accuracy suffers because the classifier is not
“expressive” enough. For instance, a triggering condition in a particular context may involve the logical “and” of a condition that consists of costs and a condition that consists of capacities. In this case, there may not be a single linear combination capable of capturing this complex pattern.

We overcome this by expanding the definition of the $C^f$ classifiers in our model. To this end, let $D$ be the number of the dimensions used in the vector representation of the local optima and $\vec{A}^d$ be the sub-vector consisting of the attributes of $A$ belonging to dimension $d$ ($d = 1, \ldots, D$). Every classifier $C^f_r$ must be defined by a set of hyperplanes, each representing a linear combination of attributes that are measured in the same units. Therefore, we replace the hyperplane-definition variables $(\vec{x}_{F_r}, b_{F_r})$ with the following:

$$(\vec{x}_{F_r}^d, b_{F_r}^d) = \text{coefficients associated with the } d\text{th dimension of the } C^f_r \text{ classifier}$$

The classification rules are changed accordingly, by adding a dimension-specific qualifier to the set of $f$-variables. In particular, we define $f^d(A, r)$ to indicate whether $\vec{A}^d$ lies or not on the positive side of the $(\vec{x}_{F_r}^d, b_{F_r}^d)$ hyperplane. The sets of constraints (5) and (6) must be changed as follows:

$$(\vec{A}^d \cdot \vec{x}_{F_r}^d + M \cdot (1 - f^d(A, r)) \geq b_{F_r}^d + \varepsilon \quad A \in \mathcal{L}; r = 1, \ldots, R; d = 1, \ldots, D) \quad (5')$$

$$(\vec{A}^d \cdot \vec{x}_{F_r}^d - M \cdot f^d(A, r) \leq b_{F_r}^d - \varepsilon \quad A \in \mathcal{L}; r = 1, \ldots, R; d = 1, \ldots, D) \quad (6')$$

And two new constraint sets must be added along with the integrality constraints for the $f^d$ variables:

$$f^d(A, r) \geq f(A, r) \quad A \in \mathcal{L}; r = 1, \ldots, R; d = 1, \ldots, D \quad (9)$$

$$\sum_{d=1}^{D} f^d(A, r) - D + 1 \leq f(A, r) \quad A \in \mathcal{L}; r = 1, \ldots, R \quad (10)$$

$$f^d(A, r) \in \{0,1\} \quad A \in \mathcal{L}; r = 1, \ldots, R; d = 1, \ldots, D$$

Constraints (9) and (10) ensure that $f(A, r) = 1$ if and only if $f^d(A, r) = 1$ for all dimensions $d = 1, \ldots, D$.

Instead of using the output of the model as “pure classifiers”, we use it for ranking. That is, for prediction purposes, the output of the classifiers is not simply the predicted class but rather a score which is “a numeric value that represents the degree to which an instance is a member of a class” (Fawcett 2006). Therefore instead of simply checking the value of $F_r(\vec{A})$ for a particular local optimum, we calculate its score as follows. For each of the $D$ hyperplanes associated with $C^f_r$, we compute the relative Euclidean distance between the hyperplane and $\vec{A}$, which is equal to a positive Euclidean distance if $\vec{A}$ lies in the positive side of the hyperplane, or to a negative Euclidean distance otherwise. The minimum among the $D$ relative distances is the score obtained by applying the classification rule $F_r$ to $\vec{A}$. We often refer to it as “the Euclidean distance between $C^f_r$ and $A$”. If $\vec{A}$ lies on the negative side of at least one hyperplane, the score is negative and $F_r(\vec{A}) = 0$; otherwise it is positive and $F_r(\vec{A}) = 1$.

In either case, this distance measures how close $\vec{A}$ is to the decision boundary. We do the same for a
given difference $\delta_{AB}$ and instead of simply checking the value of $G_r(\delta_{AB})$, we calculate the Euclidean distance between $C^f_r$ and $\delta_{AB}$, where this distance will be strictly positive when $G_r(\delta_{AB}) = 1$.

There are two reasons for ranking instead of applying “purely binary” classifiers. First, when a local optimum $A$ is encountered during the search, we must retrieve the triggering condition that is satisfied the most, which is the one that maximizes the distance between $C^f_r$ and $A$. Second, our method attempts to satisfy as much as possible the guiding constraint corresponding to that triggering condition, i.e. it attempts to maximize the distance between $C^g_r$ and $\delta_{AB}$.

3. Escaping Local Optimality

The output of the learning procedure is a set of $R$ guiding rules of the form $\{(F_r, G_r), r = 1, ..., R\}$. Considering those local optima in the training set that are close to a given local optimum $L$ and that satisfy $F_r$, then there exists at least one local optimum $M$ that is better than $L$ and for which $G_r(L, M) = 1$ and at the same time there does not exist a local optimum $M$ that is worse than $L$ for which $G_r(L, M) = 1$. Assuming that this also holds for local optima not in the training set, we want to design a procedure to escape local optimality in such a way that the search identifies directions to move from the current local optimum $L$ to a better solution $M$. The escape procedure takes as input a local optimum $L$ and attempts to find an escape direction that will move the search to a better solution $M$ by enforcing the guiding constraint corresponding to the triggering condition that is satisfied by the largest margin.

The escape is performed through a sequence of tabu search steps that are limited to reaching solutions $S$ such that the $(L, S)$ pair satisfies $G_r$. The tabu search procedure is limited to a short-term memory structure with TabuTenure drawn from a uniform distribution between minTenure and maxTenure every time a move is performed. In the remainder of the article, we use $\text{TS}_n$ to denote a tabu search process that stops after nonImprove consecutive non-improving moves. We also use $\text{TS}_m$ to denote a tabu search process that stops either after maxSteps moves or immediately after reaching a solution better than $L$. In both cases, the short term memory is managed by the minTenure and maxTenure parameters. Figure 1 shows the complete outline of the escape procedure.

In step 1, the procedure identifies the triggering condition that is satisfied the most by calculating the Euclidean distance between the local optimum point $L$ and all the conditions $C^f_r$. If no triggering condition is satisfied, then the one that is “closest” to being satisfied is chosen. Let this condition be $C^f_r$. Step 2 initializes a $\text{TS}_m$. The objective function in this part of the process is set to maximizing the Euclidean distance between the condition $C^f_r$ and $\delta_{LS}$, where $L$ is the local optimum from which the procedure is trying to escape and $S$ is the current solution. This objective function — referred to as the distance objective function to distinguish it from the original cost objective function — has the goal of directing the search toward a point $S$ for which $G_r(\delta_{LS}) = 1$. The process starts with no additional constraints imposed upon the neighborhood of the current solution.
Input: Local optimum $L$, $maxSteps$, $minTenure$, $maxTenure$ and guiding rules

1. Calculate the Euclidean distance between the local optimum point $L$ and all the conditions $C^f_v$ in order to identify the most satisfied condition $C^f_v$.
2. Initialize a tabu search $TS_m$ with the distance objective function, termination criterion $maxSteps$ and tabu tenure parameters $minTenure$ and $maxTenure$.
3. Perform tabu search steps until any of these criteria is satisfied:
   a. If a solution $S$ better (according to the cost objective function) than $L$ is reached, the escape has been successful and $S$ is returned.
   b. If a solution $S$ is reached such that Euclidean distance between $C^f_v$ and $\Delta_{LS}$ is strictly greater than zero then go to step 4.
   c. If no improving move according to the distance objective function is available or $maxSteps$ iterations have been performed then the procedure terminates with an unsuccessful escape.
4. Change to the cost objective function and restrict moves to those for which the Euclidean distance between $C^f_v$ and $\Delta_{LS}$ is strictly positive, where $S$ is the current solution.
5. Continue the tabu search process until a solution $S$ better than $L$ is reached (successful escape) or $maxSteps$ iterations have been performed (unsuccessful escape).

Figure 1. Outline of the escape procedure

Tabu search iterations are performed in step 3, where the most improving move (according to the distance objective function) is chosen in each iteration. After the move is performed, conditions 3a to 3c are checked in the order shown. First, if a solution $S$ that is better than $L$ is reached, the escape is successful and $S$ is returned. Second, if a solution $S$ such that $G_r(\Delta_{LS}) = 1$ is found, then the search has reached the goal of satisfying the guiding constraint and therefore there is no need to keep searching using the distance objective function and the process moves to step 4. Third, if all solutions in the neighborhood lead to a deterioration of the distance objective function or the number of iterations has reached the maximum allowed ($maxSteps$) then the procedure terminates with an unsuccessful escape.

Steps 4 and 5 are executed if the guiding constraint is satisfied (criterion 3b). In this case, we continue the search in the limited space of solutions that satisfy the guiding constraint, in an attempt to find a solution that is better than $L$. Therefore, we set the constraint that the Euclidean distance between $C^f_v$ and $\Delta_{LS}$ must be strictly positive for the search to move to solution $S$. The merit of a move is calculated using the cost objective function. The current number of iterations of the tabu search procedure is not reset to zero and the tabu search memory structure retains its current information. If a better solution is reached then the escape is successful and $S$ is returned. Otherwise, the search stops after a total of $maxSteps$ iterations have been performed. In summary, the escape procedure is a modified version of $TS_m$, where the search is driven not only by the tabu restrictions, but also by the most appropriate guiding constraint. Section 5.1 compares these two procedures.
We have designed three neighborhood-based procedures (i.e., local search, tabu search and escape) that use different strategies to select the next move to make and to determine when to stop. The basic neighborhood of the three procedures is the same, as defined by a common set of move mechanisms. However, some procedures modify the neighborhood and/or the move evaluation in order to navigate the solution space:

- **Local search** — The neighborhood includes all solutions that can be reached by the move mechanisms defined for the problem context. It uses the original (cost driven) objective function to evaluate moves and it stops when no improving move is available in the neighborhood of the current solution. It returns a local optimum point with respect to the entire neighborhood.

- **Tabu search** — The neighborhood includes only those solutions that can be reached by performing non-tabu moves from the current solution. The tabu status of a move may be overridden if the neighborhood contains a solution that is better than the incumbent. Two stopping criteria are used, one for $TS_n$ and one for $TS_m$, as specified above.

- **Escape procedure** — The neighborhood may be modified to include only those solutions that meet a guiding constraint. The procedure uses tabu search for which the objective function is switched from one that is based on Euclidean distance to the original based on cost. The procedure may fail to produce a path to a solution that is better than the local optimum from which it is trying to escape.

We combine these procedures into a single search organized as shown in Figure 2. We will refer to this procedure as a data mining driven tabu search (DMDTS) because of the data mining techniques employed to create the set of guiding rules used in the escape procedure.

**Input: Local optimum $L$, maxSteps, minTenure, maxTenure and guiding rules**

1. If there exists at least one guiding rule $r$ such that $F_r(L) = 1$ then perform the escape procedure otherwise perform $TS_m$ and obtain solution $S$.
2. If $S$ is better than $L$ then perform a local search starting in solution $S$, let $L$ now be the current local optimum and go back to 1. Otherwise, stop and return $L$.

**Figure 2. Data mining driven tabu search**

As shown in Figure 2, DMDTS requires an initial local optimum point $L$ (which could be chosen from the set generated for the offline learning process or generated by more sophisticated methods), the values for the tabu search parameters ($maxSteps, minTenure$ and $maxTenure$) and a set of guiding rules. If at least one of the conditions is satisfied at the current local optimum, the escape procedure is invoked. Otherwise, the simple tabu search procedure is used to escape, allowing a maximum number of iterations. Step 2 attempts to improve upon the solution $S$ found in step 1 if this solution is better than $L$. The process stops when step 1 fails to improve upon the current local optimum $L$, which is also the best solution found during the call to DMDTS.
4. Constrained Task Allocation Problem

In order to test the merit of our methodology, we have chosen a difficult constrained combinatorial optimization problem. The Constrained Task Allocation Problem (CTAP), defined by Hadj-Alouane et al. (1999), consists of finding the feasible assignment of $m$ tasks to $n$ CPUs that minimizes the total assignment cost. An assignment is feasible if the capacity of each CPU is greater than or equal to the sum of the capacity required by the tasks assigned to it. There are two types of costs: fixed cost, paid if a CPU is used, and communication cost, paid for every pair of tasks assigned to different CPUs. The mathematical model is:

Minimize

$$
\sum_{i=1}^{m-1} \sum_{j=i+1}^{m} c_{ij} \left( 1 - \sum_{k=1}^{n} x_{ik} x_{jk} \right) + \sum_{k=1}^{n} s_k y_k
$$

Subject to

$$
\sum_{k=1}^{n} x_{ik} = 1 \quad i = 1, \ldots, m
$$

$$
\sum_{i=1}^{m} a_i x_{ik} \leq b_k y_k \quad k = 1, \ldots, n
$$

$$
x_{ik} \leq y_k \quad i = 1, \ldots, m; \quad k = 1, \ldots, n
$$

$$
x_{ik} \in \{0,1\}, y_k \in \{0,1\} \quad i = 1, \ldots, m; \quad k = 1, \ldots, n
$$

Where the notation correspond to the one used by Ernst, Jiang, and Krishnamoorthy (2006):

- $x_{ik}$: binary decision variable that equals 1 if task $i$ is assigned to CPU $k$
- $y_k$: binary decision variable that equals 1 if CPU $k$ is used
- $c_{ij}$: communication cost between tasks $i$ and $j$
- $s_k$: fixed cost of using CPU $k$
- $a_i$: capacity needed by task $i$
- $b_k$: capacity of CPU $k$

The data set that we used is the same as the one in Experiment 2 of Lusa and Potts (2008). The set consists of 108 randomly generated instances of sizes ranging between 20 and 100 tasks and 5 and 30 processors, for a total of 12 different sizes. For each problem size, 3 “loose”, 3 “medium” and 3 “tight” instances were generated, resulting in $3 \times 12 = 36$ different combinations of size and “tightness” (where tightness refers to the relation between the capacity required by the tasks and the available capacity provided by the processors). Further details about these instances may be found in Lusa and Potts.
To have representation from each of the 36 combinations, we selected the first instance of each type to create a training set with a total of 36 instances. The remaining 72 instances were put aside as the test set.

We first define a solution representation and a set of basic moves. Solutions are represented by a vector \( \pi \) of size \( m \) consisting of the assignments of the tasks to processors. That is \( \pi(i) \) contains the index of the processor to which task \( i \) is assigned. Suppose that in the current solution \( \pi(i) = k \) and \( \pi(j) = l \). We define two simple moves:

1. The value of \( \pi(i) \) is changed from \( k \) to \( l \), indicating that task \( i \), currently assigned to processor \( k \), will be now assigned to processor \( l \).

2. The value of \( \pi(i) \) is changed from \( k \) to \( l \) and the value of \( \pi(j) \) is changed from \( l \) to \( k \), indicating that tasks \( i \) and \( j \) will switch processors.

With this solution representation and basic moves, we developed a local search and a simple tabu search. The local optimizer searches for the move that minimizes the assignment cost and executes the move if it improves the objective function value of the current solution. The process stops when no improving move is available in the neighborhood of the current solution. The simple tabu search adds to the local optimizer the possibility of executing non-improving moves. After the execution of a type 1 move, the attribute \( (i, k) \) is recorded to prevent task \( i \) from being assigned to processor \( k \) in the next TabuTenure iterations. When a type-2 move is executed, the attribute pair \( (j, l) \) is also recorded to prevent moves that will assign task \( j \) to processor \( k \) in the next TabuTenure iterations. The tabu-status of a move is overridden if the move results in a solution that is better than the incumbent. In order to construct the paired-data set, we generate local optima by executing the following procedure 1,000 times for each problem instance in the training set:

1. Generate a solution by randomly sorting tasks and processors, and by assigning each task to the first processor found with sufficient remaining capacity. If no such processor is found, the task is assigned to the one that yields the minimum capacity violation.

2. If the solution obtained is infeasible, then run the simple tabu search by switching the objective to minimizing the maximum capacity violation. As soon as a feasible solution is found, go to step 3. If no feasible solution is found in 1,000 iterations, then go to step 1.

3. Run the local search procedure by always selecting the feasible move that reduces the assignment cost the most.

For step 2, we choose the value of TabuTenure from a discrete uniform distribution defined on the interval \([20, 200]\). In tabu search implementations, this is usually a function of the size of the problem, as shown in the experimental section. However, we observed that a feasible solution is always found in a few iterations. This suggests that the value of the parameters of the tabu tenure interval is not critical.
for the purpose of finding a feasible solution. We also observed that for small problem instances (e.g., those with only 5 CPUs) the local search performed at step 3 converged to the same solution more than once. For these instances, the number of local optima collected was less than 1,000.

We now must define a measure of distance, set a “proximity” threshold $p$, and set a minimum number of “close” local optima $t$. The distance $d(A, B)$ between two local optimal solutions $A$ and $B$ is simply the number of assignment differences (i.e. Hamming distance between the solution representations). Therefore, the maximum distance between two solutions occurs when no task is assigned to the same processor in both solutions, resulting in a total distance $d(A, B) = m$. We consider that two solutions are “close” to each other if:

$$\frac{d(A, B)}{m} \leq 0.15$$

Furthermore, we consider only the local optimal points with at least 5 local optima within their proximity. The choice of these two values is found through a simple sensitivity analysis, which is reported in Appendix B.

The next step consists of formulating the attributes used to represent local optima. This is where knowledge about the problem context is exercised. We based these attributes on the definition of two “appealing” moves (M1 and M2) from a local optimal solution $A$ and the identification of seven processors (CPU1 to CPU7) in this solution. M1 and M2 are appealing because they have the potential of improving the value of the objective function if some constraints were relaxed or some costs ignored.

- **M1.** The best feasible reassignment of task $i$ from its current processor $k$ to processor $l$ if the largest communication cost between $i$ and any other task currently assigned to $k$ is ignored
- **M2.** Same as M1 but allowing for a capacity violation in $l$ that is no more than half of the capacity requirements of task $i$
- **CPU1.** Processor $k$ in M1
- **CPU2.** Processor $l$ in M1
- **CPU3.** Processor $k$ in M2
- **CPU4.** Processor $l$ in M2
- **CPU5.** The processor currently in use with the fewest tasks assigned to it
- **CPU6.** The processor not currently in use with the minimum fixed cost
- **CPU7.** The processor not currently in use with the largest total capacity

Note that the definitions above are independent of the size of the instance under consideration. Using these definitions, we characterize $A$ with a vector $\vec{A}$ consisting of 30 attributes:

- For each CPU $i (i = 1,...,7)$, create attributes for total capacity, available capacity, fixed cost, the percentage of tasks assigned to it ($4\times7 = 28$ attributes)
- The change in the objective function if move M1 were executed (1 attribute)
- The change in the objective function if move M2 were executed (1 attribute)

We then create 35 attributes to characterize $\vec{A}_{AB}$:
• For each CPU \( i (i = 1, ..., 5) \) in use and for each CPU \( j (j = 1, ..., 7, j \neq i) \) different from CPU\( i \), make the attribute value equal to 1 if there is at least one task that is assigned to CPU\( i \) in \( A \) and to CPU\( j \) in \( B \), and 0 otherwise (5*6 = 30 attributes)

• For each CPU\( i (i = 1, ..., 5) \) in use make the attribute value equal to 1 if there is at least one task that is assigned to CPU\( i \) in \( A \) and to processor \( k \) in \( B \), such that \( k \) is unused in \( A \), and 0 otherwise (5 attributes)

Three dimensions are used in \( \vec{A} \) to characterize \( A \): capacity \((d = 1)\), percentage of tasks \((d = 2)\), and cost \((d = 3)\), as given by the change in the objective function. Therefore, the \( C^I_T \) classifiers use 3 hyperplanes, allowing a greater expressivity than using a single hyperplane classifier. Conversely, only one dimension is used in the attributes associated with \( \vec{A}_{AB} \): the attributes are 1 if there is a subset of tasks moving from a given CPU to another and 0 otherwise.

The following is an example of a guiding rule that can be expressed by our attribute representation:

“\( \text{If the total capacity of CPU6 in } A \text{ is larger than or equal to the difference between the total capacity of CPU5 and the available capacity of CPU5 (condition on dimension 1) AND the fixed cost of CPU6 is less than the fixed cost of CPU5 (condition on dimension 3) THEN move some tasks from CPU5 to CPU6 (guiding constraint)} \)"

This classification \( C^I_T \) rule is a logical AND of two relatively simple classification rules (linear combinations), one that considers only capacity attributes and one that considers only cost attributes (as shown below). Note that it is impossible to formulate such a rule using only one linear combination that involves all attributes. Interestingly, if this condition is verified by \( A \), the objective function will certainly improve by moving all the tasks in CPU5 to CPU6, making this rule highly reliable.

One possible representation of this rule in terms of the mathematical formulation is as follows. Let \((x_{F_1}^{I}, b_{F_1}^{I})\) be the coefficients of the hyperplanes of the classifier \( C^I_T \) (the only \( C^I \) classifier in this example) relative to dimension \( d = 1, 2, 3 \). The coefficients of \( x_{F_1}^{I} \) are all equal to 0 except the ones corresponding to:

• the total capacity of CPU5, for which the value of the coefficient is equal to \(-1\)
• the total capacity of CPU6, for which the value of the coefficient is equal to \(1\)
• the available capacity of CPU5, for which the value of the coefficient is \(1\)

In addition, the hyperplane has \( b_{F_1}^{I} \) equal to 0. Then, a solution \( A \) satisfies the condition on dimension 1 if and only if it satisfies \( \vec{A}_1 \cdot x_{F_1}^{I} \geq b_{F_1}^{I} \). The guiding rule above does not involve dimension 2, therefore all components of \( x_{F_1}^{I} \) are equal to 0 and \( b_{F_1}^{I} \) is equal to \(-1\), so that any solution \( A \) always satisfies \( \vec{A}_2 \cdot x_{F_1}^{I} \geq b_{F_1}^{I} \). Finally, the coefficients of \( x_{F_1}^{I} \) are all equal to 0 except the ones corresponding to:
the fixed cost of CPU5, for which the value of the coefficient is equal to 1
• the fixed cost of CPU6, for which the value of the coefficient is equal to $-1$

$b_{F_{1}}^3$ is equal to 0 in this hyperplane. Then, a solution $A$ satisfies the condition on the third dimension if and only if it satisfies $A^3 \cdot \tilde{x}_{F_{1}}^3 \geq b_{F_{1}}^3$. Therefore, $F_{1}(\tilde{A}) = 1$ when the conditions on both dimensions (1 and 3) are satisfied.

The coefficients of $\tilde{x}_{G_{1}}$ are all equal to 0 except for the one corresponding to the presence of at least one task assigned to CPU5 in $A$ and to CPU6 in $B$. Also, let $b_{G_{1}}$ be equal to 1. Then, $G_{1}(\tilde{A}_{AB}) = 1$ if and only if there is at least one task moving from CPU5 to CPU6.

5. Computational Experiments on the CTAP

We perform three main experiments to test our ideas on the CTAP. The first experiment assesses the effectiveness of the escape procedure and the other two tests the performance of the $DMDTS$. The search parameters for these experiments are set as follows, where $m$ is the number of tasks and $n$ is the number of CPUs.

\[
\begin{align*}
\text{maxSteps} & = m \\
\text{nonImprove} & = m \\
\text{minTenure} & = 0.01nm \\
\text{maxTenure} & = 0.1nm
\end{align*}
\]

The tabu search is set up as described in section 4. That is, there are two types of moves and a short term memory controlled by a tabu tenure that is drawn from a uniform distribution between $\text{minTenure}$ and $\text{maxTenure}$.

5.1. Experiment 1

For this experiment, we consider the subset of 36 problem instances used to build the paired-data set in section 4. The experiment consists of 30 executions of the steps outlined in Figure 3. We use two versions of the escape procedure, one for which the learning stage is limited to one guiding rule (i.e., $R = 1$) and one for which $R = 2$. For the purpose of reporting results, we denote these versions as $EP - 1$ and $EP - 2$, respectively. We are interested in measuring the success rate of the two versions of the escape procedure. Success is defined as in Figure 1, that is, when the procedure is able to move from the initial local optimum $L$ to another one of a better quality. We also track the relative improvement achieved by successful escapes. Table 1 reports the results in terms of the success rate and the improvement for $EP - 1$, $EP - 2$ and $TS_{m}$. The results associated with the escape procedures are divided into the “satisfying” and “not satisfying” sets.
1. Learn the guiding rules from a subset of the paired-data set involving a random sample of 30% of all local optima (training set).

2. The remaining local optima (test set) are partitioned into two subsets labeled **satisfying** and **not satisfying**, indicating whether there exists at least one guiding rule \( r \) such that \( F_r(L) = 1 \).

3. From each local optimum \( L \) in the test set
   a. execute \( TS_m \) followed by a local search after a successful escape
   b. execute the escape procedure followed by a local search after a successful escape.

**Figure 3.** Process to test the effectiveness of the escape procedure

We used CPLEX 11.0 to find exact solutions to the learning model. Solving the learning model required an average of 16 seconds for \( EP - 1 \) and 7.2 minutes for \( EP - 2 \) on an Intel Xeon 3.20 GHz machine with 2 GB of RAM and running Windows Server 2003 Enterprise Edition. Different random seeds were used in each of the 30 runs to obtain the sample of local optimal solutions and therefore build the training and test sets. The random seed also affected the sequence of tabu tenure values chosen for both the escape and the tabu search procedures.

The results confirm that the escape procedure is more effective when the triggering condition of the guiding rule is satisfied, as indicated by the high average success rate (81.58% for \( EP - 1 \) and 68.38% for \( EP - 2 \)). While the expectation is that the escape procedure will not perform well when applied to local optima for which no guiding rule applies, we observed two cases (tests 11 and 27 for \( EP - 2 \)) where the success rate associated with the “not satisfying” subset was higher than the rate for the “satisfying” group. We attribute this to the low ratio between the cardinality of the training set and the one of the test set. The partitions corresponding to tests 11 and 27 resulted in rules that were optimal for the training set but lead to inferior escape directions in the test set. Unlike the success rate, the improvement does not dramatically decrease if the triggering condition is not satisfied. This is in agreement with the design of our experiment, since the same guiding constraint is used for both the “satisfying” and the “non satisfying” group.

The results in Table 1 also show that \( EP - 1 \) performs at a higher level than \( EP - 2 \). Additional experiments show that the performance of both \( EP - 1 \) and \( EP - 2 \) increases with the size of the training set relative to the test set. Nevertheless, the performance of \( EP - 2 \) is generally inferior. This result is counterintuitive given the flexibility added by embedding more than one guiding rule in the search. The proportion of local optima that satisfy at least one guiding rule increases from 23.92% for \( EP - 1 \) to 34.8% for \( EP - 2 \), however, the success rate drops. We attribute the inferior performance of \( EP - 2 \) to the higher complexity of the rules when compared to the complexity of the single rule used in \( EP - 1 \). We have observed that the neighborhood coverage of the single rule in \( EP - 1 \) is often greater than the individual coverage provided by the 2 rules used in \( EP - 2 \). This may be caused by the objective function of the learning model, which maximizes the total coverage as opposed to the coverage of individual rules. The result is that the 2 rules in \( EP - 2 \) tend to be more complex and more specific to the observed local optima instead of simple and general. Therefore, they fail to adhere to an
important concept of classification, the Occam's razor (Domingos 1999), which states that simple classification rules are preferred to complex ones.

Table 1. Summary of results of Experiment 1

<table>
<thead>
<tr>
<th>No.</th>
<th>EP – 1</th>
<th>EP – 2</th>
<th>(T_{SmT})</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Satisfying (23.92%)</td>
<td>Not Satisfying (76.08%)</td>
<td>Satisfying (34.80%)</td>
</tr>
<tr>
<td>1</td>
<td>73.47%</td>
<td>51.15%</td>
<td>69.74%</td>
</tr>
<tr>
<td>2</td>
<td>73.68%</td>
<td>47.52%</td>
<td>69.16%</td>
</tr>
<tr>
<td>3</td>
<td>73.81%</td>
<td>52.11%</td>
<td>66.15%</td>
</tr>
<tr>
<td>4</td>
<td>100.00%</td>
<td>47.83%</td>
<td>74.36%</td>
</tr>
<tr>
<td>5</td>
<td>98.04%</td>
<td>46.90%</td>
<td>84.78%</td>
</tr>
<tr>
<td>6</td>
<td>82.05%</td>
<td>48.82%</td>
<td>82.05%</td>
</tr>
<tr>
<td>7</td>
<td>82.69%</td>
<td>39.06%</td>
<td>60.32%</td>
</tr>
<tr>
<td>8</td>
<td>83.33%</td>
<td>33.33%</td>
<td>73.81%</td>
</tr>
<tr>
<td>9</td>
<td>85.71%</td>
<td>50.00%</td>
<td>67.19%</td>
</tr>
<tr>
<td>10</td>
<td>97.50%</td>
<td>44.81%</td>
<td>69.31%</td>
</tr>
<tr>
<td>11</td>
<td>72.00%</td>
<td>56.69%</td>
<td>62.32%</td>
</tr>
<tr>
<td>12</td>
<td>81.40%</td>
<td>54.74%</td>
<td>71.88%</td>
</tr>
<tr>
<td>13</td>
<td>83.72%</td>
<td>60.14%</td>
<td>57.00%</td>
</tr>
<tr>
<td>14</td>
<td>88.37%</td>
<td>37.33%</td>
<td>81.48%</td>
</tr>
<tr>
<td>15</td>
<td>62.50%</td>
<td>57.94%</td>
<td>67.74%</td>
</tr>
<tr>
<td>16</td>
<td>87.76%</td>
<td>28.46%</td>
<td>82.61%</td>
</tr>
<tr>
<td>17</td>
<td>84.85%</td>
<td>58.00%</td>
<td>78.26%</td>
</tr>
<tr>
<td>18</td>
<td>64.00%</td>
<td>57.14%</td>
<td>65.79%</td>
</tr>
<tr>
<td>19</td>
<td>74.14%</td>
<td>56.69%</td>
<td>68.25%</td>
</tr>
<tr>
<td>20</td>
<td>78.72%</td>
<td>45.65%</td>
<td>64.29%</td>
</tr>
<tr>
<td>21</td>
<td>100.00%</td>
<td>39.74%</td>
<td>78.57%</td>
</tr>
<tr>
<td>22</td>
<td>75.68%</td>
<td>44.60%</td>
<td>68.18%</td>
</tr>
<tr>
<td>23</td>
<td>79.59%</td>
<td>47.92%</td>
<td>61.54%</td>
</tr>
<tr>
<td>24</td>
<td>85.71%</td>
<td>49.61%</td>
<td>68.00%</td>
</tr>
<tr>
<td>25</td>
<td>71.79%</td>
<td>45.39%</td>
<td>71.15%</td>
</tr>
<tr>
<td>26</td>
<td>90.00%</td>
<td>49.28%</td>
<td>68.42%</td>
</tr>
<tr>
<td>27</td>
<td>97.73%</td>
<td>54.74%</td>
<td>53.03%</td>
</tr>
<tr>
<td>28</td>
<td>66.67%</td>
<td>54.43%</td>
<td>43.66%</td>
</tr>
<tr>
<td>29</td>
<td>79.17%</td>
<td>60.14%</td>
<td>69.23%</td>
</tr>
<tr>
<td>30</td>
<td>73.17%</td>
<td>51.75%</td>
<td>53.06%</td>
</tr>
<tr>
<td>Avg.</td>
<td>81.58%</td>
<td>49.06%</td>
<td>68.38%</td>
</tr>
</tbody>
</table>

The escape procedures outperform \(T_{SmT}\), indicating that the learning process is able to provide effective directions to move away from local optima. The simple tabu search rules are able to find a
path out of the basin of attraction of the local optimal point (with an overall success rate of 62.35%) but the improvement is significantly smaller than the one achieved by the escape procedures. In a related experiment, we used randomly generated guiding rules (that is, we generated random coefficients for the hyperplanes instead of solving the learning model) to test whether learning was actually helping the procedure. When using random guiding rules, the average success rate of $EP - 1$ dropped to 48.75% with a corresponding average improvement of 14.09%. Interestingly, in this “random rules” scenario the average improvement is also greater than the one obtained by $TS_m$. This may be explained by the attributes used to define the local-optimum differences. These attributes represent groups of tasks that move from one CPU to another. Therefore, applying a guiding rule (even one randomly constructed) often leads to a change of the set of CPUs used. The objective function value of the solution reached after the move will change considerably due to the change in fixed costs, which determine the largest portion of the total cost. Then, the average improvement, which is computed only among those cases where the solution quality improves, ends up being relatively large.

The main and the secondary experiments show that the learning model is capable of identifying effective escape directions and the conditions to apply them. They also show that the escape procedure is able to use this knowledge effectively to reach better solutions. From a data mining standpoint, the relatively small percentage of the available data that was used for training (30% of the local optima) did not lead to under-fitting, suggesting that the learning process that we propose does not require an impractical amount of effort involving extremely large training sets.

5.2. Experiment 2

In this second set of experiments, we assess the performance obtained by embedding the escape strategy into a complete search. In particular, we focus on testing the performance of the data mining driven tabu search procedure described in section 3. Since this procedure requires a local optimum as one of its inputs, we developed a multi-start process (i.e., construction and improvement) based on the GRASP methodology. GRASP constructions result in local optima that on the average are of higher quality than those obtained by a totally random process, such as the one we used for the learning process. The goal during the learning stage was to construct a data set with a sample of local optima that were not necessarily concentrated around the high quality solutions. The goal now is to study the effectiveness of the resulting search procedure when applied to local optima that are constructed with a method that is not completely random and that balances both solution quality and diversity. The details of the GRASP process are included in Appendix A. Each GRASP iteration results in a local optimum that then is used as the input to either $DMDTS$ or $TS_n$. Following the results obtained in the first set of experiments, we limit the escape procedure within $DMDTS$ to employing a single guiding rule. The values of the search parameters are set as stated at the beginning of section 5.

The test set consists of 72 of the 108 instances generated by Lusa and Potts (2008). These are the instances that were not used for the learning process and the set contains 24 in each of three categories: loose, medium and tight. We execute 50 GRASP iterations, resulting in 50 different local optima. Then, $DMDTS$ and $TS_n$ are applied to the resulting local optimum. We record, for each problem type at the end of a GRASP iteration, the number of $DMDTS$ and $TS_n$ “wins”, where a win for a method is defined as finding a solution that is better than the competing method. We then subtract the
number of $T_{S_n}$ wins from the $DMDTS$ wins to find the $DMDTS$ score (which ranges from -72 to +72). The $DMDTS$ scores are graphically shown in Figure 4, where L, M and T are used to indicate the $DMDTS$ scores for loose, medium and tight instances, respectively. The TOT area shows the total $DMDTS$ score.

![Figure 4. DMDTS scores for 50 GRASP iterations](image)

We perform a one-tailed paired $t$-test for each instance in the test set with the goal of assessing if the value of the solution found by $DMDTS$ across the 50 iterations is smaller (better) than the one found with $T_{S_n}$. With $\alpha = 0.05$, the test concludes that in 28 cases $DMDTS$ performs significantly better than $T_{S_n}$, in 6 cases significantly worse, and in 38 cases the difference is not significant. The results in Figure 4 show that, when considering all instances, $DMDTS$ outperforms $T_{S_n}$ after 50 GRASP iterations. $DMDTS$ has more consistently positive results for loose and medium instances than for tight instances. This is due to the limited number (7) of CPUs involved in the attribute representation used for the local-optimum differences. For the large tight instances, where 20 or more CPUs are used, this attribute representation is often insufficient, and the “task shuffling” produced by a simple tabu search proves to be more effective in this case. It would be of great interest to analyze the effect of using a different learning procedure for each instance type, but this is outside the scope of this study. A change in the attribute representation would also lead to a change in performance. The horizon of 50 GRASP iterations was chosen to approximate the computational effort employed by procedures that have appeared in the literature for the CTAP and with which we compare $DMDTS$ below. However, we have verified that for extremely long computational times the advantage of $DMDTS$ versus $T_{S_n}$ tends to disappear. In other words, as the number of GRASP iterations increase, the lines in Figure 4 converge to zero. In this case, the advantage of $DMDTS$ on $T_{S_n}$ starts to vanish after around 100 iterations. This simply shows that in long searches, the sampling effect of the GRASP constructions overshadows the efficiency gains achieved by the learning process.

There are two prominent heuristic procedures for the CTAP in the literature, the hybrid tabu search (HTS) method developed by Chen and Lin (2000) and the variable neighborhood search (VNS) proposed by Lusa and Potts (2008). We use the “win, tie or lose” criterion to compare the results of $DMDTS$ versus these established methods. The results are summarized in Table 2. The values in this table
indicate the percentage (in the set of 72 instances) of DMDTS wins, ties and loses against the alternative approaches. The DMDTS results are those obtained at the end of the 50 GRASP iterations in the previous experiment and they are compared to the solutions reported in the literature.

With respect to the average time required to obtain the results used to compute the values shown in Table 2, the fastest method is HTS, while VNS and DMDTS achieve similar performance: VNS is 2.64 times slower than HTS and DMDTS is 2.19 times slower than HTS. These relative comparisons of computational effort are accurate in the case of HTS and VNS, because Lusa and Potts (2008) ran these procedures on the same computer equipment. DMDTS was executed on a different computer (specified above), but accounting for the difference in equipment, we have determined that, given the same amount of computer time, DMDTS and VNS examine approximately the same number of points in the solution space.

<table>
<thead>
<tr>
<th>DMDTS vs.</th>
<th>Win</th>
<th>Tie</th>
<th>Lose</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTS by Chen and Lin (2000)</td>
<td>66.67%</td>
<td>13.89%</td>
<td>19.44%</td>
</tr>
<tr>
<td>VNS by Lusa and Potts (2008)</td>
<td>18.06%</td>
<td>27.78%</td>
<td>54.17%</td>
</tr>
</tbody>
</table>

The results shown in Table 2 indicate that DMDTS is able to outperform (on the test problems used for this experiment) an existing method based on tabu search (HTS). This is an encouraging result because it shows that a hybrid of tabu search obtained by performing the offline learning procedure seems to have an advantage over one that is based on design choices made by the developers of the procedure. The limitations of DMDTS are shown when comparing its outcomes to the results obtained by the VNS procedure. As stated by Lusa and Potts (2008), this method is specialized to the CTAP and employs the following 5 neighborhoods to search the solution space:

1. Reallocate a task $i$ from processor $k$ to processor $l$
2. Exchange two tasks (task $i$ from processor $k$ to processor $l$ and task $j$ from processor $l$ to processor $k$)
3. Reallocate a cluster of tasks from processor $k$ to processor $l$
4. Reallocate a cluster of tasks from different processors to processor $l$
5. Empty processor $k$

It is remarkable, however, that with only moves 1 and 2 from the list above, DMDTS is capable of finding 13 new best-known solutions to the set of 72 problems. Table 3 reports the objective function values of the best solutions found by Lusa and Potts (2008) and the new benchmarks that we obtained with DMDTS. The problem instances are identified using Lusa and Potts notation, i.e., the first 3 digits correspond to the number of tasks, followed by 2 digits identifying the number of CPUs, one digit to indicate a loose (1), medium (2), or tight (3) instance, and 3 digits for the instance number within the set (010, 050 or 100).
Table 3. New best-solutions for 13 CTAP problem instances.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Lusa and Potts (2008)</th>
<th>DMDTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>060_10_1_050</td>
<td>120,100</td>
<td>120,057</td>
</tr>
<tr>
<td>080_05_1_050</td>
<td>185,653</td>
<td>185,601</td>
</tr>
<tr>
<td>080_05_1_100</td>
<td>352,343</td>
<td>352,291</td>
</tr>
<tr>
<td>080_05_2_050</td>
<td>76,037</td>
<td>75,967</td>
</tr>
<tr>
<td>080_05_2_100</td>
<td>139,072</td>
<td>139,002</td>
</tr>
<tr>
<td>080_05_3_050</td>
<td>273,240</td>
<td>273,170</td>
</tr>
<tr>
<td>080_05_3_100</td>
<td>524,203</td>
<td>524,019</td>
</tr>
<tr>
<td>080_10_1_050</td>
<td>107,651</td>
<td>107,637</td>
</tr>
<tr>
<td>080_10_1_100</td>
<td>181,704</td>
<td>181,677</td>
</tr>
<tr>
<td>100_10_3_050</td>
<td>213,252</td>
<td>213,183</td>
</tr>
<tr>
<td>100_20_2_050</td>
<td>112,153</td>
<td>112,094</td>
</tr>
<tr>
<td>100_30_2_050</td>
<td>109,523</td>
<td>109,513</td>
</tr>
<tr>
<td>100_30_2_100</td>
<td>164,733</td>
<td>164,726</td>
</tr>
</tbody>
</table>

As shown in Table 2, Lusa and Potts (2008) obtain solution of better quality than those found by DMDTS on 39 instances out of 72 (54.17%). On average, these solutions are 0.89% better than DMDTS with a maximum of 5.33%.

5.3. Experiment 3

In our last experiment, we applied DMDTS to eight additional CTAP instances from the literature. Hadj-Aloune et al. (1999) introduced these problems and Ernst, Jiang, and Krishnamoorthy (2006) modified one and used the resulting set of nine instances for testing their exact procedure, concluding that:

“Our experiments indicated that the CTAP is a much harder problem than the UTAP [uncapacitated task allocation problem] mainly because of the capacity constraints. Neither integer programming formulations nor the column generation formulation for the CTAP performed well for the test problems.”

The same set of problems was used by Lusa and Potts (2008). They also coded and applied Chen and Lin (2000) hybrid tabu search to these problems. As Lusa and Potts explain, the problems were originated at “an automobile microcomputer system and a Hughes air-defense system.” Both Ernst, Jiang and Krishnamoorthy (2006) and Lusa and Potts (2008) provide a detailed description of the characteristics of these problem instances, which have been labeled A to H. (Ernst, Jiang and Krishnamoorthy (2006) added a ninth problem to the set, which they labeled H', by ignoring the preallocation of eight tasks in problem H.) Table 4 of Ernst, Jiang and Krishnamoorthy (2006) reports the best solutions found by Hadj-Aloune et al. (1999) and the best solutions found by CTAP2t, which is the integer programming formulation that worked best for these problems. Table 1 in Lusa and Potts (2008) reports these results and adds performance information associated with two versions of their VNS procedure and the hybrid
tabu search of Chen and Lin (2000). The information includes simple statistics (minimum, average and maximum) on the objective function values found in 50 runs performed on each problem instance. We also perform 50 runs of our DMDTS procedure and compare our outcomes with previous results in Table 4. We don’t include computational times, because we have already established the relative standing regarding computational effort of our method.

The Lusa and Potts (2008) results in Table 4 correspond to their GVNS2 procedure, which is reportedly their best procedure. This table and a number of additional results shown by Lusa and Potts (2008) support their conclusion that their VNS procedure outperforms the previous state-of-the-art for CTAP (namely, Chen and Lin’s hybrid tabu search). Our DMDTS is competitive and is able to find one new best solution (for problem F) but we can’t claim that it outperforms Lusa and Potts’ VNS implementation because we fail to match the best-known solutions to problems A and D.

Table 4. Comparison of five solution procedures on eight real-world CTAP instances

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A (20, 6)</td>
<td>13,450</td>
<td>13,804</td>
<td>13,519</td>
<td>13,450</td>
<td>13,866</td>
</tr>
<tr>
<td>B (20, 6)</td>
<td>11,946</td>
<td>11,946</td>
<td>11,946</td>
<td>11,946</td>
<td>11,946</td>
</tr>
<tr>
<td>C (20, 6)</td>
<td>11,120</td>
<td>11,120</td>
<td>11,156</td>
<td>11,126</td>
<td>11,120</td>
</tr>
<tr>
<td>D (40, 12)</td>
<td>39,738</td>
<td>39,680</td>
<td>41,457</td>
<td>39,214</td>
<td>39,690</td>
</tr>
<tr>
<td>E (40, 12)</td>
<td>38,602</td>
<td>36,575</td>
<td>37,731</td>
<td>35,671</td>
<td>35,671</td>
</tr>
<tr>
<td>F (40, 12)</td>
<td>35,016</td>
<td>35,821</td>
<td>36,410</td>
<td>34,674</td>
<td>34,624</td>
</tr>
<tr>
<td>G (15, 5)</td>
<td>16</td>
<td>16</td>
<td>N/A</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>H (41, 2)</td>
<td>40</td>
<td>N/A</td>
<td>40</td>
<td>40</td>
<td>40</td>
</tr>
</tbody>
</table>

Best solutions are shown in bold.

Nevertheless, this shows that the learning model could correctly characterize some improving directions that other ad-hoc methods do not consider; it also shows that the DMDTS is able to effectively enforce these directions. From a learning point of view, it is particularly encouraging to observe that some valuable information about the structure of this class of problems has been learned from random instances and then applied successfully to real world problems.

6. Application to the Matrix Bandwidth Minimization Problem

In order to show that the DMDTS may be applied to other optimization problems, we consider the Matrix Bandwidth Minimization Problem (MBMP), which consists of finding the permutation of rows and columns that minimizes the bandwidth of a matrix. This problem is equivalent to labeling the vertices of an undirected graph so that the maximum difference between the labels of any pair of adjacent vertices is minimized. The most effective approaches to the MBMP use neighborhood search procedures (Martí et al. 2001, Martí et al. 2008, Piñana et al. 2004), where the elementary move is the “swap” between the labels of two vertices. The maximum difference between the label of a vertex \( v \)
and the one of the adjacent vertices is called “bandwidth of \( v \)” and denoted by \( B(v) \). A vertex \( v \) is critical if \( B(v) \) is equal to the objective function. If a solution contains more than 2 critical vertices, it is possible that no swap leads to a decrease in the objective function. In fact, there may be several swaps that do not change the objective function value. Therefore, if such a solution is encountered during the search, a procedure based on these moves may perform many swaps that do not change the objective function value of the current solution and spend considerable computational time without improving the solution quality, until some stopping criterion is met. In other words, a procedure that relies solely on computing changes of the objective function value caused by swap will likely be ineffective to escape local optimality. This problem is overcome by establishing search direction with calculations that are different from changes on the objective function value. This characteristic makes our proposed methodology particularly suitable for the MBMP.

As previously established, our methodology requires an attribute representation for solutions and solution pairs. To apply it to the MBMP, first we group the vertices according to their bandwidth, and then we describe these groups through attributes. Let \( w \) be the objective function value of a solution. Then, we build \( G \) “bandwidth groups”, each formed by the vertices whose bandwidth is included in a given range. The range of the first group is \([w - \alpha w, w]\), where \( \alpha \) is a design parameter; the range of the second group is \([w - 2\alpha w, w - \alpha w]\), and so on. A solution is characterized by two attributes for each bandwidth group: 1) the percentage of vertices belonging to the group and 2) their average degree. We add another attribute that is equal to the bandwidth of the vertex with the highest degree (in case of ties, the vertex whose neighbors have the highest average degree).

The attributes used to describe a pair of solutions \((A, B)\) are defined by the bandwidth groups in \( A \). For each group, we add an attribute equal to \(+1\) if the number of vertices in the group increased from \( A \) to \( B \), \(0\) if it did not change, or \(-1\) if it decreased; we add an attribute equal to \(+1\) if their average degree increased from \( A \) to \( B \), \(0\) if it did not change, \(-1\) if it decreased. Finally, the last attribute is equal to \(+1\) if the bandwidth of the vertex with the highest degree increased from \( A \) to \( B \), \(0\) if it did not change, \(-1\) if it decreased.

As in the case of the CTAP, we compared the performance obtained by the simple tabu search described in section 5 to the one obtained by \( DMDTS \). Relatively to the 126 instances considered by Martí et al. (2001), \( DMDTS \) finds a better solution than the one found by the simple tabu search in 107 cases, a worse solution in 17 cases, and a solution with the same value in 2 cases. In average, the solution found by \( DMDTS \) is 28.50% better than the one found by the simple tabu search. Although these results show the advantage of using our methodology to enhance a simple tabu search, the value of the solution found is generally far from the best known solution values. In order to obtain more competitive results, we applied our methodology to the method developed by Martí et al. (2001), who propose a tabu search algorithm where the quality of a swap is not measured by the objective function, but is inversely proportional to the number of vertices that become critical or “near critical”. We considered the procedure that they refer to as \( TS(200) \), which executes a tabu search with a limit of 200 non improving steps starting from a random solution, without applying a re-starting mechanism or any other long term memory structure. We overlaid our method on \( TS(200) \) by using a guiding constraint to evaluate the swaps in order to escape local optimality (escape phase). Once the guiding constraint is satisfied, we use their evaluation function with the restriction that the guiding constraint be satisfied (exploration phase). Once a better solution is reached (successful escape), the swaps are again
evaluated using the function proposed by Martí et al. (2001), until a new local optimum is reached. Maximum number of non-improving steps, tabu tenure, and all the other search parameters are set as in the original algorithm.

Among the 126 instances considered by Martí et al. (2001), we selected those for which the best solution obtained by running $TS(200)$ from 50 different randomly chosen starting solutions has a value that is over 30% from the best known solution value. This was done to identify the instances (11 in this case) in which the performance of $TS(200)$ could be improved. In all other instances the procedure is capable of matching or nearly matching the best known solutions, making it almost impossible to assess the merit of overlaying our methodology. Nonetheless, we applied $DMDTS$ to the remaining 115 instances to verify that there is no performance loss compared to the original $TS(200)$.

First, we selected 30 instances (training set) from the 115 instances and reserved the 11 “difficult” instances as our test set. Second, we built the paired-data set by forming a pair $(A, B)$ whenever the expected relative distance between $A$ and $B$ was less than or equal to 0.02. The expected relative distance between two solutions is the ratio between their distance and the expected distance. The distance between two solutions is computed as the average absolute difference between the labels assigned to the same vertex. To define the expected distance between two solutions of an instance of size $n$, we note that a vertex that is labeled $i$ in solution $A$ may be labeled $1, 2, ..., n$ in solution $B$. Therefore, the absolute difference between the labels of $i$ is uniformly distributed in $(i-1, i-2, ..., 0, 1, 2, ..., n-i)$. So, the expected absolute difference between the labels of $i$ is:

$$
\frac{1}{n} \left( \sum_{j=1}^{i-1} j + \sum_{j=1}^{n-i} j \right)
$$

The expected distance is the average across all vertices:

$$
\frac{1}{n^2} \left[ \sum_{i=1}^{n} \left( \sum_{j=1}^{i-1} j + \sum_{j=1}^{n-i} j \right) \right]
$$

The attributes are built using 8 bandwidth groups and $\alpha = 0.1$. These parameters were tuned by analyzing the performance obtained on the training set. The time limit for the learning procedure is 20 minutes.

For each test instance, a random solution was generated and both $TS(200)$ and $DMDTS$ were executed from this solution. After the search was performed for all 11 instances, we recorded the number of wins of $DMDTS$ over $TS(200)$ and the score of $DMDTS$, as defined in Section 5. This procedure was repeated 50 times, each time using a different seed to generate the initial solutions. Figure 5 shows the score for each repetition.
Figure 5. DMDTS scores for 50 repetitions

After 50 repetitions, DMDTS wins 7 times (average deviation 11.6%), loses once (deviation -12.5%), and ties 3 times over TS(200), leading to a score equal to 6. Interestingly, the pattern depicted in Figure 5 is very similar to the one in Figure 4, corresponding to the CTAP. In both cases, the superiority of DMDTS is evident only after a few repetitions. As we did for the CTAP, we performed a one-tailed paired t-test for each instance in the test set with the goal of assessing if the value of the solution found by DMDTS across the 50 iterations is smaller (better) than the one found by TS(200). With $\alpha = 0.05$, the test determined that DMDTS performs significantly better than TS(200) on 7 instances, while the difference is not significant in 4 instances. Further analysis of the data suggests that, sometimes, the guiding constraint leads the DMDTS to a low quality solution. For some instances this may happen for all the initial repetitions, thus reducing the DMDTS score. At some point though, the guiding constraint leads to a high quality solution, better than any solution found by tabu search up to then. At this point the DMDTS score starts increasing. In terms of computational time, DMDTS takes about 3 times longer than the simple tabu search, mainly because of the extra time spent computing the attributes at local optima. Nevertheless, one should compare the time spent to overlay DMDTS on TS(200), which consists of designing the attributes and collecting local optima data against the time spent to modify TS(200) in order to improve its performance. We argue that this comparison favors DMDTS.

7. Conclusions

Existing approaches based on neighborhood search may be viewed as consisting of a local search and a set of predefined rules (e.g., tabu activation) that impose constraints (e.g., tabu restrictions) that are enforced during the search in order to allow the process to escape local optimality. We propose a method that learns the rules (learning procedure) from a training set of instances of a problem class and applies them (escape procedure) to new instances. Our method is general and we show its application to the constrained task allocation problem. The escape procedure is also one instance of a set of possible ways in which the rules that result from the learning procedure may be used to constrain a neighborhood search. The results that we obtained for the CTAP are encouraging. First, we showed that the “enhanced” tabu search outperforms the version that does not take advantage of the learning process. Then we showed that it also outperforms a tabu search that is hybrid in nature (Chen and Lin,
2000), where the hybridization is the result of a predefined noising strategy that modifies the search directions by perturbing key problem data. This is equivalent to adding “soft” constraints with the purpose of finding new search directions out of local optimal points. The difference is that the directions are not the result of rules that were learned but of design choices (e.g., the parameters to perturb and the amount of the perturbation).

Although our method requires the design of problem-specific attributes, the tradeoff is that the effort required to implement the search procedure is minimized. In other words, our framework focuses on the characterization of the problem (e.g., by the definition of attributes) rather than on the design of specialized solution strategies. While what we propose may not always be the faster approach, our experiments show that it can achieve better results. Furthermore, our approach may be overlaid on existing procedures based on neighborhood search in order to improve performance, as we did for the MBMP.

Our proposal involves some offline effort because the learning procedure is based on a mathematical formulation that is solved exactly with commercial software. This opens the door to interesting opportunities for future research, for instance, one consisting of the development of efficient exact or heuristic procedures to accelerate the learning process. The need for such a method depends on the context because typically the learning process would be performed once for a given class of problems. A more challenging problem perhaps will be to design on-line learning procedures that use a portion of the computational budget to learn rules and apply them to the problem instance being solved. Furthermore, computational studies on the impact obtained by varying the training set or the effort spent on the learning procedure would certainly be of interest.

Other research directions point to the application of our concepts to more sophisticated neighborhood searches, such as those based on VNS, or to search techniques that are based on evolving a population of solutions via combination mechanisms. In the former, the learning must consider a paired-data set originated from points that are local optima relative to all VNS moves. In the latter, the rules could identify situations for applying certain combination methods over others. We envision an alternative paradigm for the design of metaheuristic procedures, where data mining techniques are used not as a second level enhancement (e.g. for parameter tuning), but as a main tool to design the rules used to search the solution space.

**Acknowledgments**

The authors are thankful to Amaia Lusa and James Bean for providing the data sets that we used for our computational experiments on the CTAP. We also would like to thank Rafael Martí for sharing the MBMP data sets and associated tabu search implementation. Finally, we thank the anonymous referees for their valuable suggestions.

**References**


Appendix A

A Greedy Randomized Adaptive Search Procedure for the Constrained Task Allocation Problem

GRASP, or greedy randomized adaptive search procedure, is a multi-start technique, introduced by Feo and Resende (1995), that constructs and improves solutions at each iteration. The technique is often used as a standalone metaheuristic, but we use it to generate high-quality local optima from which we want to test our escape procedures. A solution for the CTAP is constructed by assigning at each step a task to a CPU, where, initially, all the CPUs are empty (i.e., no task has been assigned to any of the processors). The process stops when all tasks have been assigned. In the context of GRASP, the candidate list (CL) is formed by all the possible assignments of the unassigned tasks. The assignments in the CL have a score that measures their attractiveness and the score is updated after every assignment. A restricted candidate list (RCL) is constructed containing the best $\alpha\%$ of the assignments in CL, where the candidate assignment with the smallest score is the best. The next assignment is chosen from RCL at random. The construction step is completed when RCL is empty, that is when all tasks have been assigned. We then use the local search procedure described in section 4 to find a local optimum point and this completes a GRASP iteration.

Given a partial solution, the score of assigning task $i$ to processor $k$ is an estimate of the increase in total cost, which includes both the fixed and the communication costs. The incremental total cost is estimated as the sum of these two components:

1. The incremental fixed cost is estimated as the sum of the fixed costs of all the currently unused processors that will have to be used to meet the resource requirements of the unassigned tasks after task $i$ is assigned to processor $k$. The calculation sorts the processors in a “bang-for-buck” order (i.e., by the ratio of fixed cost to total capacity). The fixed costs are added until the capacity requirements of the unassigned tasks are satisfied (considering that some of these requirements will be satisfied by the processors currently in use). The calculation is a lower bound on the incremental fixed cost because it assumes that the tasks may be split between two or more processors.

2. The incremental communication cost is estimated as the cost of the communication cost of task $i$ and all the unassigned tasks that are not assigned to processor $k$ in future steps. Since future assignments are unknown, we calculate a per-unit-of-capacity communication cost between task $i$ and all the unassigned tasks. The total communication cost is then calculated as the product of the per-unit cost and the difference between the total capacity requirements of the unassigned tasks and the remaining capacity in processor $k$. This difference estimates the tasks (represented by capacity units) that will have to be assigned to other processors.

In our computational experiments, we set $\alpha = 15\%$, a value that is typically recommended and used in GRASP implementations and that also proved to be effective in our setting.
Appendix B

Sensitivity Analysis for the Parameters of the Constrained Task Allocation Problem

Associated with applying the proposed methodology to the CTAP, the value of the “proximity” threshold $p$ and of the minimum number of “close” local optima $t$ must be set up. These are the two parameters necessary for the construction of the paired-data set. We let $p$ assume the values 0.05, 0.10, 0.15, and 0.2. Values above 0.2 would force the model to look for escape directions that are too complex (i.e. escape directions involving the change of more than 20% of the task assignments); values below 0.05 would lead to an empty paired-data set — because it would be difficult to find two local optima in close proximity of each other. We let $t$ assume the values 2, 5, and 8. We discard the value $t = 1$ because the paired-data set would be formed by a set with only one local optimum. Therefore, the learning procedure would find a valid triggering condition but a meaningless guiding constraint, given that once a local optimum $A$ satisfies the condition there is no need to discriminate between pairs (as there is only one). In other words, the guiding rule would be “If $A$ is easy to improve, then move to any direction.” Since we are interested in guiding rules that are specific to the characteristics of the local optima, we need to consider only local optima with more than one local optimum close to them. We limit the value of $t$ to 8, otherwise the paired data set would be empty — not many local optima are close to 8 or more local optima.

For each combination of the $(p, t)$ values, half of the paired-data set is used to solve the learning problem, while the other half is used to evaluate coverage and accuracy. The coverage is the proportion of initial local optima that satisfy the triggering condition. For example, a 30% coverage means that the guiding constraint is activated in 30% of the local optima. The accuracy is a measure of the reliability of the guiding constraint. The first step to compute it is to consider only the pairs $(A, B)$ such that $A$ satisfies the triggering condition; among the pairs that satisfy the guiding constraint, the accuracy is the proportion of improving pairs. For example, an 80% accuracy means that when the guiding rule is activated, 80% of the close local optima located along the direction pursued are better than the initial one. We did not consider the combinations $(p, t)$ for which the resulting paired-data set was too large, i.e. when the learning procedure cannot find a positive value solution within 2 minutes. We also discarded the combinations for which the paired-data set was too small (less than 50 pairs), in order to ensure that the guiding rule obtained is valid on a large sample of local optima. Table 5 reports the results of our sensitivity analysis on the combinations that have not been discarded.

<table>
<thead>
<tr>
<th>$p$</th>
<th>$t$</th>
<th>Number of pairs</th>
<th>Coverage</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.10</td>
<td>2</td>
<td>177</td>
<td>0.23</td>
<td>0.84</td>
</tr>
<tr>
<td>0.15</td>
<td>5</td>
<td>142</td>
<td>0.41</td>
<td>0.85</td>
</tr>
<tr>
<td>0.20</td>
<td>8</td>
<td>216</td>
<td>0.25</td>
<td>0.75</td>
</tr>
</tbody>
</table>

The two combinations $(0.10, 2)$ and $(0.15, 5)$ have a similar accuracy, but we choose the rule obtained with $(0.15, 5)$ because it has a higher coverage, and therefore it can be applied more often.