Computers & Operations Research 37 (2010) 1977–1986

Contents lists available at ScienceDirect

Computers & Operations Research

journal homepage: www.elsevier.com/locate/caor

Black box scatter search for general classes of binary optimization problems

Francisco Gortázar [a], Abraham Duarte [a], Manuel Laguna [b], Rafael Martí [c,*]

[a] Departamento de Ciencias de la Computación, Universidad Rey Juan Carlos, Spain
[b] Leeds School of Business, University of Colorado at Boulder, USA
[c] Departamento de Estadística e Investigación Operativa, Universitat de València, Spain

ARTICLE INFO

Available online 2 February 2010

Keywords:
Optimization
Metaheuristics
Hard optimization problems

ABSTRACT

The purpose of this paper is to apply the scatter search methodology to general classes of binary problems. We focus on optimization problems for which the solutions are represented as binary vectors and that may or may not include constraints. Binary problems arise in a variety of settings, including engineering design and statistical mechanics (e.g., the spin glass problem). A distinction is made between two sets of general constraint types that are handled directly by the solver and other constraints that are addressed via penalty functions. In both cases, however, the heuristic treats the objective function evaluation as a black box. We perform computational experiments with four well-known binary optimization problems to study the efficiency (speed) and effectiveness (solution quality) of the proposed method. Comparisons are made against both commercial software and specialized procedures on a set of 376 instances. We chose commercial software that is similar in nature to the proposed procedure, namely, it treats the objective function as a black box and the search is based on evolutionary optimization techniques.

1. Introduction

Black box optimizers have a long tradition in the field of operations research. These procedures treat the objective function evaluation as a black box and therefore do not take advantage of its specific structure. Black box optimizers have also been referred to as context-independent procedures. However, the context-independent notion is more difficult to define because no solver is totally independent from the context. In fact, it can be argued that solvers are developed within a spectrum that ranges from almost no dependence on context to total dependence on context. Knowledge about the context may be divided between the objective function and the set constraints. For instance, some solvers may not have information about the structure of the objective function but have information regarding the feasibility region as defined by a set of constraints. In the case of mathematical programming approaches, such as linear programming, the solvers have a very specific structure that they exploit (even though they "do not know", for instance, if they are solving a transportation or an aggregate production planning problem). The structure—and therefore context dependency—is given by the formulation and not by the unknown (to the solver) real world context.

Nonlinear optimization approaches that do not use derivatives (or estimate them) to find search directions—such as Nelder and Mead or Powell—may be considered as being at the high level of the context-independence range because they treat the objective function as a black box. Methods that estimate derivatives—like those based on generalized reduced gradients (GRG)—assume objective function smoothness and that assumption alone moves them closer to the context-dependence end of the spectrum. However, we cannot ignore the fact that some of these procedures (for example the standard version of Microsoft Excel's Solver) are routinely used to search for solutions to problems that do not meet the "smoothness" requirement. The Solver acts as a general purpose procedure in the sense that it does not require that the user provides any context. In the experimental section, we compare our proposed procedure with the Evolutionary Premium Solver (SDK callable library), which implements advanced search strategies for black box optimization.

A similar philosophy is used by the general-purpose commercial optimization software known as OptQuest (by OptTek Systems, Inc.). This software operates by treating the objective function evaluation as a black box. However, OptQuest is not totally "ignorant" of the context given that a solution representation must be chosen in order to run this solver. OptQuest allows users to represent solutions as a mixture of continuous, discrete, integer, binary, permutation and other specialized variables

* Corresponding author. Tel.: +34 96 3543090 4362; fax: +34 96 354 3238.
E-mail addresses: francisco.gortazar@urjc.es (F. Gortázar), abraham.duarte@urjc.es (A. Duarte), laguna@colorado.edu (M. Laguna), rafael.marti@uv.es (R. Martí).

0305-0548/$ - see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cor.2010.01.013

(project, categorical or enumeration). Clearly, the solution representation gives OptQuest some information about the problem context and therefore the solver context-independence changes with each particular application because the amount of information that it receives varies. The software chooses solvers based on the characteristics of the optimization model: Pure or mixed, constrained or unconstrained and deterministic or stochastic.

The first characteristic refers to the solution representation. That is, if only one type of variables (e.g., continuous or discrete) is used, then the problem is pure, otherwise the problem is mixed. We consider both constrained and unconstrained problems in the realm of deterministic optimization.

Evolver by Palisade Corp. is a popular black-box optimizer based on genetic algorithms. When coupled with @Risk, a module for risk analysis, Evolver can be used to search for solutions to problems for which the objective function is the result of a Monte Carlo simulation. Evolver can be used as a callable library in the Evolver Development Kit or within Microsoft Excel as its modeling environment, where decision variables are declared as "adjusting cells." The use of specialized problem solving methods, such as "recipe", "order" or "budget", is encouraged because they give the solution process additional information that could result in improved outcomes and thus move the solution process closer to the context-dependent end of the spectrum. We compare our proposed procedure with the Evolver Development Kit, Solver and OPtQuest, selecting the appropriate solving method to tackle binary problems.

In addition to general purpose (and commercially available) software for optimizing unknown objective functions with discrete variables, the literature contains some examples of procedures that have been designed with some degree of context-independence. For instance, Rosen and Harmonosky [1] proposed a simulated annealing that uses an adaptation of the response surface methodology to optimize simulations with a large number of discrete decision variables. Guikema et al. [2] also address the problem of optimizing simulations when the input variables are discrete. Their method is based on using a standard genetic algorithm aided by ridge regression to limit the number of calls to the simulator.

Holland's [3] genetic algorithms proposal was in fact a black-box optimizer that used an array of bits as the generic representation. The proposed procedure did not include local search and the standard genetic operators (such as single-point crossover) were not linked to the problem context. As GAs became more popular and researchers and practitioner applied them to many hard optimization problems, the context-independent nature of the original proposal began to vanish when improved outcomes were obtained by the addition of problem structure. Hence, most modern GA implementations are hybridized (e.g., coupled with local searches) and incorporate domain-specific knowledge into the search process.

Cross Entropy [4] is a more recent method for black-box optimization that originated from the need of estimating very small probabilities in rare-event simulation. Respectable performance has been achieved by implementations of the cross entropy method for some difficult combinatorial problems. However, as shown by Laguna et al. [5] in the context of the max-cut problem, cross entropy also benefits from hybridization and it is uncertain for how long the method will remain true to its origins as a black box procedure.

The remaining of the article discusses our development and testing of a black box procedure for constrained and unconstrained optimization problems whose solutions are represented as binary strings. The scatter search methodology is used as the basic framework for our solution procedure. Although the scatter search philosophy is to take advantage of domain-specific knowledge, this methodology has been used in the past as the basis for black box procedures (e.g., for OptQuest and for the procedure to tackle permutation problems developed by Campos et al. [6]).

The main contribution of our work is the development or adaptation of scatter search methods that are helpful in the solution of binary problems for which the objective function is treated as a black-box evaluator. In particular, we developed three diversification generation methods for binary solutions (Section 2.2), an improvement method based on two neighborhoods (Section 2.3) and seven combination methods (Section 2.4). We also adapted to our binary problems a reactive strategy within the combination methods, which was originally proposed by Campos et al. [6] for permutation problems. Finally, our procedure includes a strategy for selectively applying the improvement method in order to reduce the computational effort associated with the black-box evaluator.

2. Scatter search procedure

Scatter Search (SS) is a metaheuristic that explores solution spaces by evolving a set of reference points. It consists of five methods and their associated strategies. Three of them, the Diversification Generation, the Improvement and the Combination Methods, are typically problem-dependent and are designed for the problem or the class of problems being solved. The other two, the Reference Set Update and the Subset Generation Methods are context independent by design, and standard implementations are available [7].

2.1. Classes of problems

We have adapted SS to develop a black box solver for optimization problems with binary variables. We assume that a problem instance consists of finding a set of values for $x=(x_1, x_2,\ldots, x_n)$—where $x_i=0$ or 1—in order to maximize an unknown objective function. The user has the choice of specifying whether the problem is unconstrained or constrained. For *unconstrained* problems, any binary vector of n elements is a feasible solution. If the problem contains constraints the user may choose to transform it into an unconstrained problem by constructing a penalty function whose values are returned by the black box linked to the optimizer. The solver, however, is capable of directly dealing with two general classes of constraints: *multiple choice* and *budget.* Defining these constraints as part of the input to the optimizer adds a level of context-dependency that may result in improved outcomes (just as OptQuest or Evolver benefit from additional specificity in the choice of variable types or input constraints).

Multiple-choice problems are such that k items must be chosen from a total of n. This translates into formulating a multiple-choice constraint that forces that exactly k variables take on the value 1. In mathematical terms:

$$\sum_{i=1}^{n} x_i = k \tag{1}$$

The user specifies the value of k and the solver limits the search to solutions with exactly k variables set to 1.

Budget problems are those containing constraints that limit the amount of resources used by a given solution. In this case, resource utilization increases with the number of variables that are set to the value of one. Infeasible solutions occur when the available resources are exceeded. The mathematical form of this constraint may vary because resource utilization may be linear or

nonlinear with respect to the variables that take on the value of 1. In the case of the knapsack problem, for instance, the constraint has the following mathematical form:

$$\sum_{i=1}^{n} a_i x_i \le B \tag{2}$$

The budget is given by the value of b and the a_i coefficients indicate the amount of resources needed if the ith option is selected. In our implementation, the solution evaluator returns a Boolean value indicating whether the solution is feasible or not. In other words, the solution method does not know the level of infeasibility or what specific variables need to be given a value of zero to make the solution feasible. The solution method, however, does know that for this type of problems, feasibility may be eventually achieved by switching variable values from 1 to 0.

As mentioned above, if the problem contains constraints that cannot be casted as multiple-choice or budget, these constraints must be handled by the black-box by way of a penalty function. The penalty function should be such that the objective function value of any infeasible solution must be worse than the objective function value of any feasible solution. The specific form of the penalty function is the responsibility of the user and a number of alternatives have been studied in connection with evolutionary algorithms [8]. In the remainder of the paper, we refer to constrained problems as those containing multiple choice or budget constraints (as specified by the user). Problems with general constraints are considered unconstrained and the solution method focuses on optimizing the penalized objective function.

2.2. Diversification generation method

The search starts with the application of a diversification generation method, as shown in Fig. 1, that results in a population P of $PSize$ points from which a subset is selected as the initial reference set ($RefSet$). To obtain solutions with different structures we apply three different generators of binary vectors and create $PSize/3$ solutions with each one of them.

The first one, G1, proposed by Glover [9] uses a systematic approach to create a diverse set of binary vectors. It generates a collection of solutions associated with an integer $h=2,3,\ldots,h_{\max}$, where $h_{\max} \le n-1$. From a seed solution x, we generate solutions, x', for each value of h, by the following rule: $x'_{1+kh} = 1-x_{1+kh}$ for $k=0,1,2,3,\ldots, \lfloor n/h \rfloor$, where $\lfloor k \rfloor$ is the largest integer satisfying $k \le n/h$. All other components of x' are equal to x.

We apply this generator directly in the case of unconstrained problems and slightly modified in the case of constrained problems. For multiple-choice problems, the modification consists of an early termination of the method when the number of 1s in the solution matches k. For budget problems, the process stops when setting the value of a variable to one makes the solution infeasible.

Once we generate $PSize/3$ solutions with the method above, which focuses on diversification and not on the quality of the resulting solutions, we compute $score(i)$ for each variable x_i to estimate its contribution to the objective function value in order to generate the remaining $2PSize/3$ solutions considering both quality and diversity. The score calculation has the following mathematical form, where $\bar{f}_A$ indicates the average of the objective function f over the solutions in A:

$$score(i) = \frac{\bar{f}_{P_i^1}}{\bar{f}_{P_i^1} + \bar{f}_{P_i^0}} \quad \text{where } P_i^1 = \{x \in P : x_i = 1\},\ P_i^0 = \{x \in P : x_i = 0\} \tag{3}$$

The second generator, G2, constructs a solution step by step, starting with all variables set to 0, and switching in each step one variable from 0 to 1. The variables are randomly selected and the probability of changing a selected variable from 0 to 1 is given by

$$Prob(x_i = 1) = \min(0.1 + score(i), 1) \tag{4}$$

The addition of 0.1 to the computation of the probability reflects a bias to change the variable value from 0 to 1 (this is an arbitrary value and any other "small" value would work as well). For unconstrained problems, G2 performs steps as long as the solution under construction improves. For multiple-choice problems, G2 stops after k variables change their values from 0 to 1. For budget problems, it stops when the solution becomes infeasible (i.e., G2 changes variables from 0 to 1 as long as the solution remains feasible).

The third generator, G3, can be viewed as a destructive method. It was suggested by Glover [9] and adapted by Duarte and Martí [10]. It starts with all variables set to 1 and at each step it switches the value of one variable from 1 to 0. Variables are probabilistically selected according to their $score$ values. The complement of the scores (i.e., $1-Prob(x_i=1)$) is used to calculate the probability of changing the value from 1 to 0. The stopping criterion is customized for each type of problem in a similar way as in G2.

After each construction the $score$ values are updated. We have empirically found that better results are obtained when the contribution of the last constructed solution to the $score$ value is smoothed. The smoothed score, $sscore$, is computed as a function of the score in the previous constructions and the current score. Let $score_t(i)$ be the score of variable i after the tth construction, and let $score_{t-1}(i)$ be its score in step $t-1$ (before the construction), both computed with expression (3). Then, we compute the smoothed score, $sscore(i)$, as

$$sscore(i) = \alpha score_{t-1}(i) + (1-\alpha) score_t(i) \tag{5}$$

where the parameter α controls the contribution of the current score relative to its previous value. Then, we use $sscore$ instead of $score$ to generate the values of the variables in the next constructions. We point out that the smoothing of the scores is similar to the way in which probability values are updated in the cross entropy method [4].

The initial $RefSet$ must balance solution quality and diversity, and we follow the standard reference set update method, selecting the best $b/2$ solutions (where $b=|RefSet|$) from P and then the $b/2$ solutions in P that are most diverse with respect to those already in the $RefSet$ (computing the maximum of the minimum distances between each solution and the solutions already in $RefSet$ as it is customary in scatter search). Diversity is measured according to the Hamming distance between solutions.

In the standard SS design [7] the improvement method is applied to all the solutions in P. However, in context-independent solvers, local search methods are usually extremely time-consuming since every trial move must be evaluated by invoking the black-box evaluator. Therefore, the improvement method in our procedure is applied selectively instead of across the board.

The improvement method consists of a local search procedure, which we describe in Section 2.3. It is applied to the best $b/2$ solutions in the $RefSet$ (see step 3 in the pseudo-code of Fig. 1). The reference set is a collection of b solutions that are used to generate new solutions by way of applying a combination method. In order to design a context-independent methodology that performs well across a wide collection of different binary problems, we propose—in Section 2.4—a set of seven combination methods from which one is probabilistically selected according to its performance in previous iterations. The selection process is reactive, as described below.

We follow the standard subset generation method and, in step 5 of Fig. 1, the procedure generates all pairs of reference solutions

1. Start with *GlobalIter*=0 and $P = \emptyset$. Use the diversification generation method to construct a solution. Let x be the resulting solution. If $x \notin P$ then add x to P (i.e., $P = P \cup \{x\}$), otherwise, discard x. Repeat this step until $|P| = PSize$.
2. Use the reference set update method to build $RefSet = \{ x^1, \ldots, x^b \}$ with b solutions from P. Order the solutions in *RefSet* according to their objective function value such that x^1 is the best solution and x^b the worst. Make *NewSolutions* = TRUE.
3. Apply the improvement method to the $b/2$ best solutions in *RefSet*.

while (*NewSolutions*) **do**
4. *Pool* = Ø
5. Generate *NewSubsets* with the subset generation method. Make *NewSolutions* = FALSE.

while (*NewSubsets* ≠ ∅) **do**
6. Select the next subset s in *NewSubsets*.
7. Apply the solution combination method to s to obtain one new trial solution x.
8. If $x \notin Pool$ then add x to *Pool* (i.e., $Pool = Pool \cup \{x\}$)
9. Delete s from *NewSubsets*.

end while
10. Apply the improvement method to the $b/2$ best solutions in *Pool*. Replace the original solutions in *Pool* with the improved ones.
11. Apply the reference set update method. Update the *RefSet* with the b best solutions in $RefSet \cup Pool$

if (*RefSet* has changed) **then**
12. Make *NewSolutions* = TRUE.

else
13. Apply the reference set update method. Rebuild the *RefSet* replacing the worst $b/2$ solutions with new diverse solutions from P.
14. Make *NewSolutions* = TRUE.

end if

if (*ExecutionTime* > MAXTIME) **then**
15. STOP. Return the best solution in *RefSet* as the output of the method.

end if

end while

Fig. 1. Outline of the scatter search procedure.

that have not been combined before. Here again, we limit the application of the improvement method to promising solutions. Specifically, we store the trial solutions resulting from the application of the combination method in a temporary *Pool*, and we apply the improvement method to the $b/2$ best solutions in *Pool*. The reference set is then updated (see steps 10 and 11 in Fig. 1) by selecting the best b solutions from the union of the *RefSet* and *Pool* (where the improved solutions replaced the original ones in *Pool*). If the *RefSet* changes (i.e. a new trial solution in *Pool* improves upon the worst solution in the *RefSet*), then step 12 sets the Boolean variable *NewSolutions* equal to TRUE and performs a new iteration in the outer while loop, applying again the subset generation method. Alternatively, if the combination method is incapable of creating solutions that can be admitted to the *RefSet*, the reference set is rebuilt in step 14. The rebuilding consists of keeping the best $b/2$ solutions intact and replacing the worst $b/2$ solutions in the *RefSet* with new diverse solutions from *P*. The method stops after a pre-established *MaxTime* CPU seconds.

2.3. Improvement method

A global iteration of the improvement method consists of three steps. First, we construct the candidate list CL of elements (variables) to be changed, which simply consists of all the variables in the problem. In the second step we order the elements (variables) in CL according to their *score* value (where those with the largest score are located first) and then, in the third step, we scan them in this order in search for improving moves. We perform several iterations in the third step alternating *flip* (changing the value of a single variable from 1 to 0 or from 0 to 1) and *swap* (exchanging the values of two different variables) moves. In the first iteration we try *flip* moves with all the elements in CL (examined in order), changing their value if it improves the objective function. The CL is reconstructed at this point. Then, in the second iteration we consider *swap* moves with all the elements in CL (examined in order), in which we try to exchange the value of each variable with the value of another variable. To do this we implement a *first* strategy, which scans the list of variables in search for the first one whose movement results in a strictly positive change of the objective function value. After this process, the CL is reconstructed again. Further iterations are performed alternating between *flip* and *swap* moves. The method stops after *MaxImpIter* iterations or before, if no improvements are achieved after trying all *flips* and *swaps*.

It must be noted that in constrained problems these moves may lead to an infeasible solution. However, we filter the moves and only consider those that lead to feasible solutions. Specifically, when a move produces an infeasible solution, we discard the move and examine the next move in the exploration.

2.4. Reactive combination method

As mentioned earlier, we propose seven different combination methods and a reactive mechanism that probabilistically selects among them according to their ability to produce high quality solutions during the current search. This method was successfully applied by Campos et al. [6] in the context of permutation problems.

Solutions in the *RefSet* are ordered according to their objective-function value (where the best solution occupies the first place). When a solution obtained with a combination method CM_i qualifies to be the jth member of the current *RefSet*, we add $b-j+1$ to *success*(CM_i). Therefore, combination methods that generate good solutions accumulate higher *success* values and increase proportionally their probability of being selected. To avoid initial biases, this mechanism is activated after the first

InitIter combinations, and before this, selections are made completely at random. A description of the seven combination methods follows. These methods generate the new trial solution z from the combination of two reference solutions x and y. It should be mentioned that the *score* value, initially computed with the solutions in P, is updated during the entire search process (i.e. computed considering not only the solutions in P but all the solutions examined during the search so far), thus providing an estimation of the contribution of each variable (when it takes on the value of 1) to the objective function value.

The CM_1 combination method first computes the partial solution z formed with the "union" of x and y, that is

$z_i = 1$ if $x_i = 1$ or $y_i = 1$

$z_i = 0$ otherwise

It then performs a series of steps switching some variables from 1 to 0 in a similar way as the G3 generator. Specifically, at each step, the method uses the score values to select probabilistically one variable with the value of 1 to switch it to 0. For unconstrained problems, it continues in this fashion while the solution improves; for multiple-choice problems the method performs steps until the number of variables set to 1 matches k. For budget problems, it performs steps until the solution becomes feasible. Note that for this type of problems, the infeasibility is always reduced by switching variable values from 1 to 0. The CM_2 combination method is similar to CM_1 with the only difference that the variable selection (to switch the value from 1 to 0 in z) is performed completely at random, that is, without considering the *score* values.

The CM_3 combination method is an adaptation of the one proposed by Laguna and Martí [7] in the context of the knapsack problem. The method calculates a *weight* for each variable, based on the objective function value of the two reference solutions being combined. The *weight* for variable i that corresponds to the combination of reference solutions x and y is calculated with the following formula:

$$weight(i) = \frac{f(x)x_i + f(y)y_i}{f(x) + f(y)} \tag{6}$$

where $f(x)$ is the objective function value of solution x and x_i is the value of the ith variable. Then, the trial solution z is constructed by using the *weight* as the probability for setting each variable to one, i.e., $\text{Prob}(z_i = 1) = weight(i)$. Note that this combination method assumes that the objective function is being maximized.

In constrained problems a test is included to ensure the feasibility of the resulting solution. In the case of *multiple-choice problems*, the method stops when k variables in z have been set to 1. If less than k variables are set to 1, the resulting infeasible solution is discarded. In the case of the *budget problems* the method stops when the resource limit B is reached (and the selection of an extra variable would violate it).

The CM_4 combination method first computes the partial solution z formed with the "intersection" of x and y:

$z_i = 1$ if $x_i = 1$ and $y_i = 1$

$z_i = 0$ otherwise

A number of steps are then performed in which a variable with the value of zero is chosen. The probability of selecting variable i is proportional to $weight(i)$. (Note that the larger the weight the more attractive it is, at least from the history of the search, to set the value of the variable to one.) The difference between CM_3 and CM_4 is that CM_3 starts the process of switching variable values from 0 to 1 with all the variables set to 0 while CM_4 starts with the partial solution z that represents the intersection of the reference solutions being combined. CM_4 stops in the same way as CM_1, depending on the type of problem.

The CM_5 combination method is very similar to CM_4. Starting from the partial solution constructed with the intersection of the reference solutions, variables that are set to 0 are chosen to change their value to 1. The only difference with CM_4 is that variables with a value of 0 are selected completely at random, that is, ignoring their *weight* values. This method is the "fixed crossover" developed by Dolezal et al. [11] for their genetic algorithm designed to tackle max-cut problems.

The CM_6 combination method starts with all the variable values set to 0. Then, it applies the G2 constructive method with the restriction that the z_i variables that are candidates to be switched to 1 are those with a value of 1 in x or y (i.e., those for which $x_i + y_i \geq 1$).

The CM_7 combination method is based on the path relinking methodology adapted for GRASP [12]. It generates new solutions by exploring trajectories that connect high-quality solutions—by starting from one of these solutions, called an *initiating solution*, and generating a path in the neighborhood space that leads toward the other solutions, called *guiding solutions*. The CM_7 combination method explores the path between two solutions x and y in the *RefSet*. It starts with the first solution x, and gradually transforms it into the guiding solution y, by changing the value of the variables in x with their value in y. If the value is the same in both solutions, then no change is made and the procedure moves to the next variable. Our procedure examines the variables in lexicographical order and in at most n steps it reaches y. The procedure uses a first-improving strategy, meaning that if during the relinking process it finds an intermediate solution that is better than either x or y, then the procedure stops. If no better solution is found, the solution that is most distant from x and y is the combined solution resulting from the application of this method. In addition to exploring the path from x to y, the procedure also constructs the path from y to x and chooses the best solution found during both processes to be the outcome of CM_7.

3. Optimization problems used for testing

We have used four combinatorial optimization problems to test our procedure. Solutions to these problems are naturally represented as binary vectors:

1. the max-cut problem
2. the maximum diversity problem
3. the knapsack problem
4. the multi-demand multi-dimensional knapsack problem

We target these problems because they are well known, they are different in nature, and optimal (or high-quality) solutions to several problem instances are readily available. Existing methods to solve these problems range from construction heuristics and metaheuristics to exact procedures. We now provide a brief description of each problem class.

The *max-cut* problem consists of finding a partition of the nodes of a weighted graph into two subsets such that the sum of the weights on the edges connecting the two subsets is maximized. A solution to this problem can be represented as a binary vector with cardinality equal to the number of nodes in the graph (where the value 0 or 1 indicates that the associated node belongs to one or other subset). The max-cut problem falls within the class of binary unconstrained problems.

Beginning with the simple approach introduced by Sahni and Gonzales [13] different heuristics and metaheuristics have been proposed for this problem. Recently Festa et al. [14] developed six different algorithms based on the variable neighborhood search, GRASP and Path Relinking (PR) methodologies. We will compare

our method with the SS algorithm [15] that was shown to outperform existing methods.

The *maximum diversity* problem (MDP) consists of selecting a subset of k elements from a set of n elements in such a way that the sum of the distances between the chosen elements is maximized. Clearly, a solution to this problem can be represented as a binary string x, where variable x_i takes on the value of 1 if element i is selected and 0 otherwise, $i=1,\ldots,n$. There is only one constraint in this problem that forces that exactly k variables in the string are assigned the value of 1. This problem belongs to the class of multiple choice problems. Many different methods have been proposed to search for solutions to the MDP. Recently, Silva et al. [16] presented two GRASP approaches, Duarte and Martí [10] introduced a tabu search and Gallego et al. [17] a scatter search method. We will use the latter method in our comparison since it has been shown to outperform previous approaches.

Knapsack problems are well known in the operations research literature. The problem consists of choosing, from a set of items, the subset that maximizes the value of the objective function subject to a capacity constraint. Mathematically, the problem can be expressed as follows:

$$\begin{aligned} &\textit{Maximize} \quad \sum_i c_i x_i \\ &\textit{Subject to} \quad \sum_i a_i x_i \le B \\ &\quad x_i \in \{0,1\} \ \ \forall i \end{aligned}$$

This is a budget-constrained problem that attempts to maximize the benefit associated with selecting a subset of n available objects. For a comprehensive examination of the knapsack and other related problems we refer the reader to Martello and Toth [18]. Pisinger [19] considered a three step method based on the well known ratios c_i/a_i and forward and backward computations. We will use this specialized method in our experiments and compare results against optimal solutions.

The 0/1 *multi-demand multi-dimensional knapsack* problem (MDMKP) represents a class of practical problems, including portfolio selection, capital budgeting and some facility location problems. Mathematically, the problem can be formulated as follows, where the m knapsack constraints are followed by the q demand constraints:

$$\begin{aligned} &\textit{Maximize} \quad \sum_i c_i x_i \\ &\textit{Subject to} \quad \sum_j a_{ij} x_j \le b_i \quad \forall i \in \{1,\ldots,m\} \\ &\quad \sum_j a_{ij} x_j \ge b_i \quad \forall i \in \{m+1,\ldots,m+q\} \quad x_i \in \{0,1\} \quad \forall i \end{aligned}$$

It must be noted that the MDMKP represents a class of binary problems with general constraints for which finding feasible solutions may not be trivial. Arntzen et al. [20] proposed an adaptive memory programming algorithm for this problem. Their method is based on a short-term tabu search coupled with strategic oscillation around the feasibility boundaries. This method outperforms previous heuristics due to Cappanera and Trubian [21] and therefore we will include it in our computational testing. Cappanera and Trubian [21] noted that the intriguing combinatorial structure of the MDMKP makes it a challenging problem for commercial integer linear programming solvers.

4. Experimental results

This section describes the computational experiments that we have performed to test the efficiency of our black box scatter search procedure as well as comparing it with various methods from the literature. We have implemented the methods in Java SE 6 and all the experiments were conducted on a Pentium 4 computer at 3 GHz with 2 GB of RAM. We have employed four sets of instances in our experimentation (available at http://heur.uv.es/optsicom):

Max-cut This data set consists of 94 instances from two sources. The *Hartmann* problems are 10 instances with 125 nodes and 375 edges all with weight values equal to -1 or 1. The second set consists of 84 instances ($n=50$–300) generated with *rudy*, a machine independent graph generator by Giovanni Rinaldi. The toroidal, planar and random graphs have weights taking the values of -1, 0, or 1.

MDP This data set consists of 92 instances from two sources. The first one with 40 instances, referred to as *Glover*, for which the values are calculated as the Euclidean distances from randomly generated points with coordinates in the 0–100 range. The second one with 52 instances, referred to as *Silva*, for which the values are integer numbers randomly generated between 50 and 100. The value of n ranges from 50 to 300 and the value of k ranges from $0.1n$ to $0.3n$.

Knapsack This data set consists of 96 instances from four types, as documented in Pisinger [22]: *strongly correlated, subset sum, uncorrelated* and *weakly correlated* instances. We consider 24 instances in each group with $n=100$, 300, 1000 and 3000 (6 instance for each n value).

MDMKP This data set consists of 94 instances from Cappanera and Trubian [21] in which basic multi-knapsack instances were modified by adding demand constraints and also allowing for negative cost coefficients. We consider 47 instances with positive coefficients and 47 with both positive and negative coefficient values. The value of n ranges from 50 to 250, the value of m ranges from 5 to 15 and the value of q ranges from $m/2$ to m.

We first compute, for each instance, the overall best solution value, *BestValue*. The *BestValue* may be the optimal value if, as in the case of the knapsack problem instances, it is known. Otherwise, *BestValue* represents the best-known as found by the execution of all methods under consideration. Then, in each experiment, we compute for each method the relative percent deviation (RPD) between the best solution value (*Value*) obtained with a particualr method and *BestValue* for that instance. That is,

$$RPD = \frac{(BestValue - Value)}{BestValue} \cdot 100\%$$

We report the average *RPD* across all the instances considered in each particular experiment. We also report, for each method, the number of instances (*#Best*) for which the value of the best solution obtained with this method matches *BestValue.* In the preliminary experiments, however, *#Best* refers to the solutions found within the experiment since simplified versions of the method are not expected to match best-known solutions. In addition, we calculate the *Rank* statistic—proposed by Ribeiro et al. [23]—associated with each method. For each instance, the *n_rank* of a method M is defined as the number of methods that found a better solution than the one found by M. In case of ties, all the methods receive the same *n_rank*, equal to the number of methods strictly better than all of them. The value of *Rank* is the sum of the *n_rank* values for all the instances in the experiment, thus, the lower the *Rank* the better the method.

In our preliminary experimentation we employed a set of 138 representative instances that included 42 *Max-cut* (*rudy* instances with $n \geq 200$), 48 *MaxDiv* (*Glover* and *Silva* instances with $n \geq 125$) and 48 *Knapsack* instances (from the four sources with $n \geq 1000$). We did not include *MDMKP* instances in this preliminary experimentation because we report the results obtained with our method when solving this problem without parameter tuning, which is relevant information regarding expected performance of a general purpose solver.

In the first preliminary experiment, we study the impact of changes in the value of the α parameter, which is used to update the *score* in the construction procedures within the diversification generation method. We also test the contribution of the *score* and the smoothed score, *sscore*, to the final solution. We run three versions of our SS algorithm (as it appears in Fig. 1) for 5 global iterations with only one combination method, CM_1, instead of the reactive mechanism. The first version, SS-random, does not incorporate the score and therefore probabilistic selections in the diversification and improvement methods are replaced here with completely random selections. The second version, SS-score, incorporates the score strategy, without the smoothness factor. The third version, SS-sscore(α), adds the smoothness parameter α for the updating of the score values. Note that by virtue of Eq. (5), SS-score is equivalent to SS-sscore(0). Table 1 reports the *RPD*, *#Best* and *Rank* values for the set of 138 instances and each version (with two values of α for SS-sscore(α)) as well as CPU time in seconds (Time).

Given the results summarized in Table 1, we determined that the best choice for the value of α is 0.3, which yields the lowest relative deviation and the best *Rank* value. The table also reveals the contribution of probabilistic selection based on smoothed scores over totally random choices.

In this preliminary experiment we also examined the contribution of each diversification generation method (G1, G2 and G3) to the solution quality. Specifically, we employed G1 to generate the *PSize* solutions in *P* and obtained a RPD value of 3.23%, instead of the 2.72% presented in Table 1 in which the three methods are applied (each one generating *PSize*/3 solutions). Similarly, we applied G2 or G3 to populate *P* and obtained RPD values of 3.09% and 2.88%, respectively. This reinforced our conjecture that the combined use of three different generation methods is more effective than the application of a single method in isolation.

In the second preliminary experiment we consider the contribution of the improvement method to the overall scatter search procedure. Table 2 reports the average of the RPD values of two procedures, the SS without the improvement method and the SS with the improvement method. These were both run with the *PSize* parameter set to 100. The results in Table 2 are quite conclusive about the contribution of the improvement method to the quality of the best solutions found during the search.

In our third preliminary experiment we search for an effective value for the *MaxImpIter* parameter in the improvement method. This parameter determines the amount of search effort invested in improving solutions. A large value emphasizes search intensification while a small value favors diversification (by allowing the SS to spend more time combining solutions than improving them). Clearly, intensification and diversification are induced by a number of factors in the scatter search method and here we are only controlling one of them as it relates to the number of iterations in the improvement method. Our experimentation showed that a value of 30 for *MaxImpIter* achieves a good balance between exploration and exploitation and therefore we chose this value to complete the setting of the search parameters.

One of the key elements in the way solutions are combined within our implementation of scatter search is the self-adapting nature of the combination methods. Therefore, as part of the performance analysis of our SS implementation, we wanted to know whether some combination methods are more effective than others for particular problems. With this goal in mind, we performed two additional preliminary experiments. In the first one we recorded the number of times that each combination method was used throughout the search when the procedure was applied to three problem classes. Table 3 reports the relative frequency of the use of each combination method by problem class. For this experiment, the full SS procedure was used with a time limit of 5 s. The full SS includes the improvement method and the seven combination methods.

In the second experiment relative to the contribution of the combination methods, we consider seven different scatter search versions, SS-CM_i for i=1–7, in which only one combination method, CM*i*, is applied. In the last row of Table 4, we include the complete SS version, BinarySS, in which the reactive mechanism with the seven combination methods is activated.

Tables 3 and 4 reveal the importance of embedding several mechanisms for combining reference solutions within the scatter search framework. Consider, for instance, CM_5 in Table 3. This method is rarely used for *MDP* instances (only on 2% of the cases) while it is the fourth most used for *Max-cut* instances (on 13% of the cases). Similar examples of varying performance are found in

Table 1
Analysis of the *score* parameter.

Method	RPD (%)	#Best	Rank	Time (s)
SS-random	4.43	37	368	38
SS-score	3.09	40	311	54
SS-sscore(0.3)	2.72	52	295	55
SS-sscore(0.5)	2.87	51	296	55

Table 2
RPD values of 10 runs of the improvement method.

	Max-cut (%)	MaxDiv (%)	Knapsack (%)
SS without Imp.	46.40	8.98	10.43
SS with Imp.	7.16	1.00	1.42

Table 3
Frequency of use of the combination methods.

Problem	CM_1	CM_2	CM_3	CM_4	CM_5	CM_6	CM_7
Max-cut	170	244	170	185	171	172	169
MDP	49824	5209	2221	4092	2272	23800	49461
Knapsack	89	93	96	102	99	70	161

Table 4
Contribution of combination methods.

Method	RPD (%)	#Best	Rank	Time (s)
SS-CM1	3.52	31	611	14
SS-CM2	3.50	26	593	15
SS-CM3	3.07	42	404	12
SS-CM4	3.44	23	639	15
SS-CM5	3.76	29	593	12
SS-CM6	3.22	30	533	12
SS-CM7	3.31	38	475	12
BinarySS	2.77	48	390	11

Table 3, with CM_1 and CM_7 as the possible exceptions. Both methods seem to perform well across all three problem types. Table 4 presents the advantage (both in terms of solution quality and speed) of employing several combination methods and the reactive mechanism over versions with a single combination method.

We now compare our scatter search implementation (BinarySS)—employing the parameter values that resulted from the experiments above—against three well known commercial solvers:

- the Evolutionary Premium Solver by Frontline Systems (http://www.frontsys.com) in the Solver Software Development Kit (version 7.2). This solver manages the constraints with the *fcnconstraint* function, defining an *upper bound* in the case of budget problems and *lower and upper bounds* for multiple-choice problems.
- OptQuest (version 6.2) by OptTek Systems (http://www.opttek.com). This engine allows us to define *upper requirements* to handle constraints in Budget problems and *dual requirements* for multiple-choice problems.
- Evolver by Palisade Corporation (http://www.palisade.com) in the Evolver Development Kit (version 4.1.2). This method manages constraints with the *evconstraintadd* function for both budget and multiple-choice problems.

We also consider state-of-the-art specialized methods for each problem class, all of which are expected to outperform general context-independent solvers. We are only using them as a baseline for comparison. Specifically, for the *Max-cut* instances, we consider the SS method by Martí et al. [15], for the *MDP* instances we use the SS method by Gallego et al. [17], and for the *Knapsack* instances we use Expknap by Pisinger [19]. In this experiment, we add the *MDMKP* instances and the Almha procedure by Arntzen et al. [20]. None of the *MDMKP* instances were used for fine tuning and therefore they will be a good test for the generalization attributes of the solver that we have created. For *Max-cut* and *MDP* we compare against the best known solutions. For the *Knapsack* and *MDMKP* instances we compare against the optimal solutions. The termination limit was set to 30 s, however, some methods finished earlier, as triggered by their internal logic. Tables 5–8 present the RPD, *#Best*, *Rank*, and the CPU time for each problem class, respectively. We also report in Tables 6–8 the number of instances for which each method is able to find a feasible solution, given that it is not unusual that black-box solvers fail to find feasible solutions for highly constrained instances. A RPD value of 100% is assigned when a feasible solution is not found. In this way, the average RPD value reflects the method's ability to find feasible solutions.

Table 5
Max-cut problem (94 instances).

Method	RPD (%)	#Best	Rank	Time (s)
OptQuest	19.09	6	302	25.14
Evolver	10.34	10	204	16.16
Solver	8.99	3	213	14.84
BinarySS	5.20	13	114	12.33
SS [15]	0.00	91	0	13.17

Table 6
Maximum diversity problem (92 instances).

Method	RPD (%)	#Best	Rank	Time (s)	#Feasible
OptQuest	2.48	7	169	30.51	92
Evolver	88.93	1	305	23.84	9
Solver	2.40	2	207	10.85	92
BinarySS	0.22	31	59	1.97	92
SS [17]	0.07	71	16	30.02	92

Table 7
Knapsack problem (96 instances).

Method	RPD (%)	#Best	Rank	Time (s)	#Feasible
OptQuest	2.64	62	36	5.24	96
Evolver	59.03	3	320	7.95	40
Solver	51.76	7	290	6.56	47
BinarySS	1.68	30	124	45.02	96
Expknap [19]	0.75	41	76	0.11	96

Table 8
Multi-demand multi-dimensional knapsack problem (94 instances).

Method	RPD (%)	#Best	Rank	Time (s)	#Feasible
OptQuest	31.85	0	189	24.40	65
Evolver	90.63	0	367	10.95	11
Solver	39.87	0	289	12.75	65
BinarySS	29.46	0	207	20.47	71
Almha [20]	1.86	28	11	59.97	94

Table 9
Standard deviations of RPD across 10 replications (376 instances).

Method	**Max-cut**	**MDP**	**Knapsack**	**MDMKP**
OptQuest	0.00011	0.00007	0.00008	0.00128
Evolver	0.00302	0.00195	0.00033	0.00687
Solver	0.00327	0.00123	0.00056	0.00448
BinarySS	0.00082	0.00007	0.00017	0.00517

To handle multiple constraints in the *MDMKP* problem we consider the static penalty approach described in Yeniay [8] in which the penalization of an infeasible solution x is computed as $K(1-s/m)$ where K is a large positive constant set to 10^9, m is the number of constraints and s is the number of satisfied constraints. We used this penalty function when executing all the methods in the comparison set.

The behavior of BinarySS is fairly consistent across problem classes. The method delivers low relative percent deviations with respect to the best-known (or optimal) solutions and always ranks best among the black-box solvers, which are also based on evolutionary strategies. As mentioned before, the RPD values in Tables 6–8 are computed considering a value of 100% for infeasible solutions. Anyway, one must take into consideration the success rate in finding feasible solutions when assessing RPD values.

The performance of BinarySS on the *MDMKP* set is more than acceptable. This is a set of instances to which BinarySS has not been exposed before (e.g., during the tuning process) and, as presented in Table 8, the procedure is able to find 71 out of 94 feasible solutions. This compares well with OptQuest, Evolver, and Solver, which found 65, 11, and 65 feasible solutions, respectively. The RPD values in Table 8 also support the conclusion that the BinarySS is able to perform at a higher level than the competing black-box approaches.

In order to test the robustness of the results presented in Tables 5–8, we replicate 10 times each method on the entire set of 376 instances. Table 9 presents the standard deviation of the RPD values for each method and each problem type. We can observe that all the methods exhibit very small standard deviations of the RPD values in the four problems (smaller than 0.01 in all the

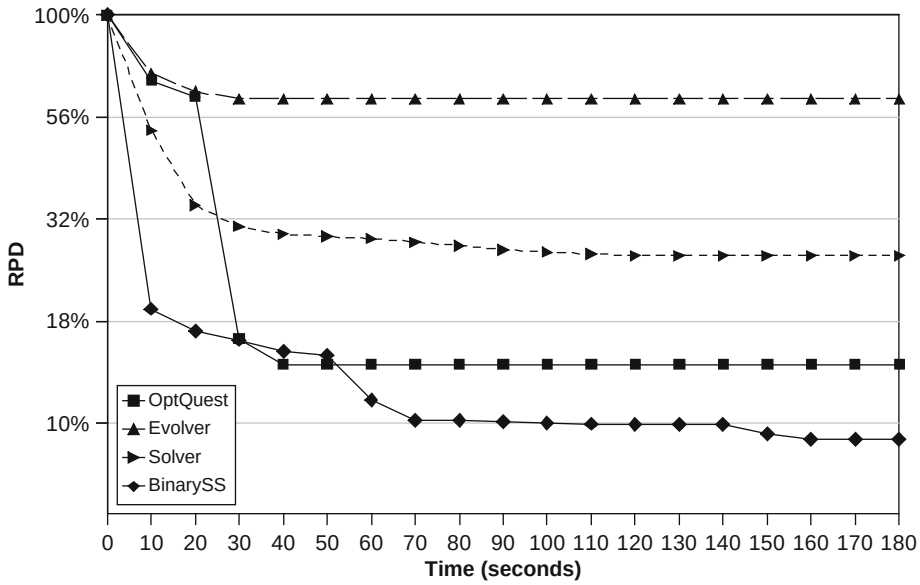

Fig. 2. Performance profile for a 180-s run.

cases), and thus we may conclude that the four black-box solvers exhibit a high level of robustness.

Our final experiment is designed to show how the average value of the best solution found improves over time. We use the reduced set of problems that were employed for the fine-tuning experiments and run the four context-independent procedures for 180 s while recording the average deviation values every second. Fig. 2 shows the results of this experiment which does not include the *MDMKP* instances because the deviation is only informative when the method is able to obtain a feasible solution, and as presented in Table 8, this happens only in a small fraction of the instances for two of the methods.

The profile in Fig. 2 indicates that our method is capable of finding high-quality solutions to all the instances in the three classes from the early stages of the search. At no time during the span of the search, with the exception of two OptQuest points, the BinarySS produces a RPD value that is worse than any of the competing methods. In this figure, the RPD values are calculated with the best solution found up to that point in the search. Finding high quality solutions early in the search is particularly important in the context of simulation–optimization, where the search horizon (defined as the number of calls to the objective function evaluator) may be particularly limited.

5. Conclusions

We have described the development of a scatter search application to optimization of problems whose solution representation is given by a binary vector. We have discussed the notion of context-independence and how solver designers attempt to use some limited, but important, information to improve the quality of the outcomes. In our particular procedure, we disclose two types of constraints to the solution method. This gives our solver some advantages when constructing and combining solutions because feasibility can be achieved easier than when dealing with general constraints (linear or nonlinear) that are handled with penalty functions.

The results of our experiments with 376 problem instances are quite conclusive regarding the effectiveness of the method that we have developed. In the process of developing this method, we are able to show the advantages of using multiple combination methods. The first-improving strategy when applied to the complete neighborhood resulted in an effective use of the allotted searching time. We expect that our experience will help software developers to improve upon the commercial solvers that are based on evolutionary processes.

Acknowledgments

This research has been partially supported by the *Ministerio de Ciencia e Innovación* of Spain (Grant Refs. TIN2006-02696, TIN2009-07516 and SEJ2005-08923/ECON), by the Comunidad de Madrid—Universidad Rey Juan Carlos project (Ref. URJC-CM-2008-CET-3731 and by the Government of Castilla y León (Consejeria de Educación Project BU008A06). The authors would like to thank OptTek Systems for running the OptQuest tests and providing the solutions that we have used for comparison purposes.

References

[1] Rosen LR, Harmonosky CM. An improved simulated annealing simulation optimization method for discrete parameter stochastic systems. Computer and Operations Research 2005;32:343–58.
[2] Guikema SD, Davidson RA, Çagnan Z. Efficient simulation-based discrete optimization. In: Ingalls RG, Rossetti MD, Smith JS, Peters BA, editors, Proceedings of the 2004 winter simulation conference, 2004. p. 536–44.
[3] Holland J. Adaptation in natural and artificial systems. Ann Arbor: University of Michigan Press; 1975.
[4] Rubinstein RY, Kroese DP. The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation, and machine learning. New York: Springer; 2004.
[5] Laguna M, Duarte A, Martí R. Hybridizing the cross entropy method: an application to the max-cut problem. Computers and Operations Research 2009;36(2):487–98.
[6] Campos V, Laguna M, Martí R. Context-independent scatter and tabu search for permutation problems. INFORMS Journal on Computing 2005;17(1):111–22.
[7] Laguna M, Martí R. Scatter search: methodology and implementations in C. Boston: Kluwer Academic Publishers; 2003.
[8] Yeniay Ö. Penalty function methods for constrained optimization with genetic algorithms. Mathematical and Computational Applications 2005;10(1):45–56.
[9] Glover F. A template for scatter search and path relinking. Lecture Notes in Computer Science, vol. 1363, 1998. p. 13–54.
[10] Duarte A, Martí R. Tabu search for the maximum diversity problem. European Journal of Operational Research 2007;178:71–84.
[11] Dolezal O, Hofmeister T, Hanno Lefmann Y. A comparison of approximation algorithms for the Maxcut-problem. Technical Report, Universitat Dortmund, Lehrstuhl Informatik; 1999.

[12] Laguna M, Martí R. GRASP and path relinking for 2-layer straight line crossing minimization. INFORMS Journal on Computing 1999;11:44–52.
[13] Sahni S, Gonzales T. P-complete approximation problem. Journal of the Association for Computing Machinery 1976;46:48–61.
[14] Festa P, Pardalos PM, Resende MGC, Ribeiro CC. Randomized heuristics for the Max-cut problem. Optimization Methods and Software 2002;7:1033–58.
[15] Martí R, Duarte A, Laguna M. Advanced scatter search for the Max-Cut problem. INFORMS Journal on Computing 2009;21(1).
[16] Silva GC, Ochi LS, Martins SL. Experimental comparison of greedy randomized adaptive search procedures for the maximum diversity problem. Lecture Notes in Computer Science, vol. 3059, 2004. p. 498–512.
[17] Gallego M, Duarte A, Laguna M, Martí R. Heuristics algorithm for the maximum diversity problem. Computational Optimization and Applications 2009;44(3):411–26.
[18] Martello S, Toth P. Knapsack problems: algorithms and computer implementations. Chichester–New York: Wiley; 1990.
[19] Pisinger D. An expanding-core algorithm for the exact 0-1 knapsack problem. European Journal of Operational Research 1995;87:175–87.
[20] Arntzen H, Hvattum LM, Lokketangen A. Adaptive memory search for multidemand multidimensional knapsack problems. Computers and Operations Research 2006;33:2508–25.
[21] Cappanera P, Trubian M. A local search based heuristic for the demand constrained multidimensional knapsack problem. INFORMS Journal on Computing 2005;17(1):82–98.
[22] Pisinger D. Core problems in knapsack algorithms. Operations Research 1999;47(4):570–5.
[23] Ribeiro CC, Uchoa E, Werneck RF. A Hybrid GRASP with perturbations for the Steiner problem in graphs. INFORMS Journal on Computing 2002;14:228–46.