Tabu search-based metaheuristic algorithm for software system reliability problems

M. Caserta\textsuperscript{a,}\textsuperscript{*}, A. Márquez Uribe\textsuperscript{b}

\textsuperscript{a}Institute für Wirtschaftsinformatik, Universität Hamburg, Von-Melle-Park 5, 20146 Hamburg, Germany
\textsuperscript{b}Lamar University, Cherry Engineering 2203, PO Box 10032, Beaumont, TX 77710-0032, USA

Abstract

This paper presents a new metaheuristic-based algorithm for complex reliability problems. The algorithm effectively uses features of the Tabu Search paradigm, with special emphasis on the exploitation of memory-based mechanisms. It balances intensification with diversification via the use of short-term and long-term memory. The algorithm has been thoroughly tested on benchmark problems from the literature as well as on a pool of random generated instances of very large scale software systems. The proposed algorithm proves to be robust with respect to its parameters and it is especially suited for very large scale instances of the reliability problem, when exact approaches are doomed to fail.

© 2007 Elsevier Ltd. All rights reserved.

Keywords: Tabu search; Reliability; Metaheuristics; Combinatorial optimization

1. Introduction

Many modern systems, both hardware and software, are characterized by a high degree of complexity. To enhance the reliability of such systems, it is vital to define techniques and models aimed at optimizing the design of the system itself. This paper presents a new metaheuristic-based algorithm aimed at tackling the general software system reliability problem, where one wants to identify the system configuration that maximizes the overall system reliability, while taking into account a set of resource constraints.

One of the most relevant applications of reliability optimization is the software reliability problem. Forty percent of software development cost is spent on testing to remove errors and to ensure high quality \cite{1}. One traditional way to increase the reliability of a software system is by increasing the time of debugging. However, due to the high debugging costs, more often an alternative solution is pursued, which is the use of redundant software modules. In software development, redundancies are programs developed by different teams based on the same specifications. Often, in order to ensure the independence of the different modules, these programs are written using different languages, platforms, development methodologies and testing strategies.

A possible definition of “reliability” for a software system, provided by Musa \cite{2}, is the probability that the software operates without failure in a specific environment during a specific exposure period. The objective of the reliability problem is to maximize the reliability of the system via redundancy allocation, while respecting resource constraints.

\textsuperscript{*} Corresponding author. Tel.: +49 40 42838 4294; fax: +49 40 42838 5535.
E-mail addresses: caserta@econ.uni-hamburg.de (M. Caserta), alberto.marquez@lamar.edu (A.M. Uribe).

0305-0548/$ - see front matter © 2007 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cor.2007.10.028

The general software system studied in this paper is made up by several programs, each performing a different function. Every program contains a set of modules. For each module, a number of different versions exist, each characterized by a cost and a reliability level. The objective is to maximize the reliability by choosing the optimal set of versions for each module, allowing redundancy.

The mathematical formulation of a software reliability problem with \( m \) modules (each one performing a different function), \( m_i \) versions for each module \( i (\sum_{i=1}^{m} m_i = n) \) and \( K \) resource constraints is provided below:

\[
\text{(RP):} \quad \begin{align*}
\max \ R_s &= f(x) \\
\text{s.t.} \quad & g(x) \leq b \\
& c'x \geq 1 \ \forall i \in M \\
& x \in \mathbb{B}^n,
\end{align*}
\]

where \( f : \mathbb{B}^n \to \mathbb{R}^+ \), \( g = (g_1, g_2, \ldots, g_K) \) (size \( K \times 1 \)), with \( g_l : \mathbb{B}^n \to \mathbb{R}^+ \), \( b \) is the resource availability vector \( (k \times 1) \), \( c' \) is a vector \( (1 \times n) \) whose component \( j \) is 1 if \( x_j \) belongs to module \( i \), and 0 vice versa. Finally, the variable \( x \) is a binary vector, where a value 1 indicates that version \( j \) is in the system and 0 vice versa.

Constraint (C1), \( g(x) \leq b \), represents economical and financial constraints (not necessarily linear), while constraint (C2), \( c'x \geq 1 \), implies that each module \( i \) must have at least one version.

The objective function of the system \( f \) depends on the particular configuration of the software system tackled and the overall number of components in the system. The reliability of the system is computed by using conditional probability on the reliability/failure of certain parts of the system. An illustration of the use of conditional probability is provided in Caserta and Ryoo [3].

The purpose of the paper is to present a new robust algorithm for software reliability optimization, able to solve any system configuration problem and especially suited for very large scale instances. The underlying assumption required in order for the algorithm to find good quality solutions is that the reliability function \( f \) as well as the constraint functions \( g_l, l = 1, \ldots, K \), satisfy the “monotonic property” presented below in Definition 1.

We now introduce the notion of monotonic property for a boolean function (see Crama [4]). Let \( \mathbb{B}^n \) denote the \( n \)-dimensional hypercube \( \{0, 1\}^n \), let \( f(x_1, \ldots, x_n) \) be a boolean function defined on \( \mathbb{B}^n \), and let \( N = \{1, 2, \ldots, n\} \). The first derivative of \( f \) with respect to \( x_i \) is the boolean function defined as

\[
\delta_i f(x) = f_+(x) - f_0(x), \quad (1)
\]

where \( f_+(x) = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n) \), \( f_0(x) = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n) \), for all \( x \in \mathbb{B}^n \).

**Definition 1.** A boolean function \( f \) is monotonic if, for each \( i \in N \), \( \delta_i f \) has constant sign on \( \mathbb{B}^n \).

It is a well-known fact that any multi-linear function in integer variables can be transformed to an equivalent 0–1 multi-linear function (see Hammer and Rudeanu [5]). For this reason, the above definition applies to any “multi-linear” integer function and it is valid for all the problems presented in this paper.

The approach followed in this paper is similar to the one presented by Berman and Ashrafii [6], where the difference between *ex post* and *ex ante* software reliability models is highlighted. While the former has little or no impact on the design of the software system, the latter deeply affects the way the overall system is eventually designed. The basic idea of such a “prescriptive” approach is that recommendations, or guidelines, are provided in the “design” phase of the system itself, rather than once the failure occurs.

The proposed algorithm is aimed at tackling any kind of reliability problem, provided that the monotonic property of Definition 1 is ensured. However, this paper devotes much attention to the software reliability problem. We will show that the algorithm is especially suited to tackle very large scale instances of the problem, where exact approaches are doomed to fail.

The paper has the following structure. In Section 2, a brief review of the state-of-the-art with respect to software reliability models and algorithms is offered. In Section 3 we present the schema of the proposed algorithm. Section 4 illustrates some computational results on problems from the literature (Section 4.1) and on a set of randomly generated reliability problems of very large dimension (Section 4.2). Finally, Section 5 presents some concluding remarks.
2. Literature review

As illustrated in Belli and Jedrzejowicz [7], reliability optimization is a branch of general reliability theory, developed first during the 1960s. Typically, two kinds of reliability problems arise: the first one attempts to minimize the amount of resources used in the design of a system that meets the required reliability level, while the second one maximizes the overall system reliability taking into account resource availability constraints.

A number of exact and heuristic-based approaches have been proposed in recent years, each one of them addressing a specific class of problems. What emerges from the analysis of the literature is that existing models and algorithms can be used only under specific circumstances.

Belli and Jedrzejowicz [7] present two different models for optimization via redundancy allocation. The underlying assumption of the authors is that the software system is made up by independent modules connected in series. The problem is reduced to finding the optimal redundancy level for each module within the system while respecting the overall resource consumption. Each module $i$ is made up by $m_i$ parallel components. The probability of reaching component $i$ is given by $1 - \prod q_k$, with $k = 1, \ldots , i - 1$, where $q_k$ is the probability of failure of $k$th component. No solution to the problem has been offered.

Another approach is proposed by Berman and Ashrafi [6]. The authors present four different models, corresponding to four system configurations. The most complex one, with $k$ functions, $m$ modules and $m_i$ versions for each module, is aimed at identifying the optimal redundancy allocation of versions within each module. The authors tackle a problem with two functions and three modules ($k = 2, m = 3$) and one single resource constraint. No algorithm is provided to tackle such a problem and the solution to the proposed instance is obtained by using the integer programming module of an optimization tool (LINGO).

Several heuristic-based techniques have been developed in the literature for the approximate solution of the reliability problem for complex systems configuration [8–11]. For example, Shi [8] provide a heuristic algorithm based upon the computation of a sensitivity factor defined as a ratio between the increase in reliability of the subsystem, taking into account and the consumption of resources in the subsystem itself. This sensitivity factor is computed for every path from the in-node to the out-node, and for the path with the minimum sensitivity factor, a component was selected and its redundancy was incremented by one. Computational results were presented on a 2-bridge network system with five components.

Kim and Yum [9] presented a metaheuristic algorithm that allowed the search path to visit outside the feasible space (excursion) to move away from a local solution. Extensive computational results were reported on small test problems as well as randomly generated instances of 2-bridge network systems with up to 20 components.

Ravi et al. [10] propose an improving nonequilibrium simulated annealing-based algorithm to find the optimal redundant allocation in complex systems, while considering different constraint types (space, volume, etc.). However, the algorithm is tested on small instances of the problem.

The central issue in the class of problems discussed in this paper is the identification of which modules in the system should receive a greater amount of resource allocation. For this reason, a number of different metaheuristic-based algorithms could be envisioned for this class of problems. For example, methods such as Cross Entropy or Ant Colony, that progressively build a solution by adding redundant components to the system, could be used to solve the problem. However, the proposed algorithm, based upon the tabu search (TS) paradigm and relying upon the results of Caserta and Ryoo [3], presents a new effective meta-heuristic algorithm especially suited for very large scale software system reliability optimization. What makes TS particularly suited to tackle reliability problems is the combination of two main ingredients of this metaheuristic: on the one hand, TS is one of the metaheuristic schemes that allows to explore “portions” of the infeasible space and, on the other hand, TS makes good use of memory based mechanisms to bias the search trajectory. As explained later on with more details, while the first feature is used to escape from local attractors, the second feature guarantees that previously visited solutions, including local attractors, will not be visited again. This idea of exploring part of the infeasible space exploits a mathematical property of the class of problems here studied, the monotonic property of the objective function value. This property ensures that the algorithm can easily avoid to be trapped into a local optimum by temporarily “jumping” into the infeasible space, where we can find solutions that use up more resources and that, consequently, have a higher value of the objective function value. Finally, the effective use of memory-based mechanisms guarantees that, once the algorithms gets back into the feasible space, it will not visit again the same local optimum, hence exploring different regions of the feasible space.
3. Proposed algorithm

The TS paradigm keeps track of not only local information, but also of information acquired during the search space. Its main characteristic is the use of memory-based mechanisms, that record not only information about the current solution, e.g., objective function value, but also information about the trajectory followed to reach the current solution. This information is embedded in the choice of the next move within the current neighborhood. Memory is used to forbid moves, labeled as “tabu”, which might lead back to previously visited solution. Below we provide a general TS algorithm. However, we refer interested readers to Glover [12,13] and Glover and Laguna [14] for a thorough presentation of the method.

\begin{algorithm}
\textbf{algorithm} GeneralTS;
\begin{algorithmic}
\STATE \textbf{begin}
\STATE S1. generate initial solution
\STATE S2. select a “nontabu” move in the neighborhood of the current solution
\STATE S3. update best solution found
\STATE S4. update tabu lists and memory mechanisms
\STATE S5. \textbf{if} stopping criterion \textbf{then} STOP; \textbf{else} go back to S2.
\STATE \textbf{end}
\end{algorithmic}
\end{algorithm}

The proposed algorithm relies on the TS paradigm presented by Glover [12,13], and offers a specialization of the paradigm with respect to the software reliability problem. The most outstanding features of the algorithm are here presented:

1. it permits to thoroughly explore the search space by means of a simple mechanism that allows to escape from local optima;
2. it makes use of short/medium-term memory through the implementation of the Active Tabu List (working memory) and the Tabu List (short-term memory);
3. it exploits long-term memory to balance intensification and diversification through the implementation of a “path relinking” strategy, specialized for the software reliability problem.

The proposed algorithm makes use of the different features presented in Sections 3.1–3.3. In the following, let us indicate with $x \in B^n$ a solution to the reliability problem such that at least one version is chosen for each module (the vector $x$ is feasible with respect to constraints (C2) of (RP)). While we enforce that each visited solution is always feasible with respect to the set covering-type constraints, to implement feature (1) of the algorithm and escape from local attractors, we allow the algorithm to temporarily visit solutions infeasible with respect to constraints (C1), counting the number of excursions into the infeasible space (see Section 3.1 for more details). We present the TS algorithm with the following pseudo-code, where $\lambda$ is the maximum number of excursions allowed into the infeasible region:

\begin{algorithm}
\textbf{algorithm} TS;
\begin{algorithmic}
\STATE \textbf{begin}
\STATE $k \leftarrow 0$
\STATE $t \leftarrow 0$
\STATE generate $x^0$
\WHILE {($k < \lambda$)}
\STATE $j^{\bullet} \leftarrow \text{argmax}_{x \in \mathcal{N}(x^t)} \{ w \in \mathcal{N}(x^t), w \notin \mathcal{T^L} \}$
\STATE \textbf{if} ($\mathcal{N}(x^t) = \emptyset$) \textbf{then} increase counter \textbf{else} change search direction
\STATE execute move along $j^{\bullet}$
\STATE update $\mathcal{T^L}$ & $\mathcal{T}^L$ \textbf{update memory}(see Sections 3.2 and 3.3)
\STATE $x^{t+1} \leftarrow x^t$
\STATE \textbf{if} $z(x^{t+1}) > z^*$, update best result
\STATE $t \leftarrow t + 1$
\STATE \textbf{end if}
\STATE $k \leftarrow k + 1$
\STATE \textbf{end while}
\STATE \textbf{end}
\end{algorithmic}
\end{algorithm}

The computational complexity of the overall algorithm is $O(|S|n^2\kappa f(z,A))$, where $|S|$ indicates the cardinality of the “elite solutions” set, $n = \sum_{i=1}^{m} n_i$ is the total number of bits in the binary solution vectors, $\kappa$ is a measure of the complexity of evaluating the objective function value of a solution, which depends on the system configuration, and $f(z,A) = (z \min\{b_{ij}\}/\min\{c_{ij}\})A$ indicates the size of the “enhanced” feasible space in terms on number of possible moves within the region. The detailed computational complexity of each phase of the algorithm is presented in Sections 3.1–3.3.

### 3.1. General features of the TS algorithm

The basic element of any local search algorithm is the definition of a neighborhood, which is the set of points that can be reached from a given solution. In the following, let us indicate with $x \in \mathbb{B}^n$ a solution to the reliability problem such that at least one version is chosen for each module (the vector $x$ is feasible with respect to the second constraint of (RP), $c'x \geq 1$, $\forall i \in M$—we call such a vector “module-feasible”). Let $N = \{1, 2, \ldots, n\}$ be the set of indices in $x$. Furthermore, let $B^x = \{j \in N : x_j = 1\}$ be the set of elements of $x$ set to 1 and $N \setminus B^x$ the set of elements set to 0 in $x$.

We first define the feasible space as

$$ X = \{x \in \mathbb{B}^n : g(x) \leq b\} $$

and the “enhanced” feasible space as

$$ \overline{X} = \{x \in \mathbb{B}^n : g(x) \leq b(1 + \alpha)\}. $$

The region $\overline{X} \setminus X$ is called “allowed infeasible space” and is made up by the set of “module-feasible” solutions that use an $\alpha\%$ more of the total the resources available, with $\alpha \in [0,1]$.

With respect to any $x^k \in \overline{X}$, we define $\mathscr{B} = B^x \setminus B^x$ and $\overline{\mathscr{B}} = B^x \setminus B^x$. We further define two different neighborhoods, namely

$$ \mathcal{N}_+(x^k) = \{x \in \overline{X} : |\mathscr{B}| = 1 \wedge |B^x| - |B^x| = 1\}, $$

$$ \mathcal{N}_-(x^k) = \{x \in \overline{X} : |\overline{\mathscr{B}}| = 1 \wedge |B^x| - |B^x| = 1\}. $$

It can be easily seen that $\mathcal{N}_+(x^k)$ is the set of “module-feasible” solutions that can be reached from $x^k$ by setting a component of $x^k$, say $x^k_j \in N \setminus B^x$, to 1. Conversely, $\mathcal{N}_-(x^k)$ is the set of “module-feasible” solutions that can be reached from $x^k$ by setting a component of $x^k$, say $x^k_j \in B^x$, to 0.

Starting from a current solution $x^k$, at each iteration of the algorithm, a move in $\mathcal{N}_+(x^k)$ or $\mathcal{N}_-(x^k)$ is chosen according to the following three criteria: direction of the search trajectory, merit score, and tabu status. With respect to the direction of the search trajectory, the monotonic property of the resource constraint functions (and of the objective function) with respect to $x$ is exploited. The algorithm is made up by two different phases: (i) an ascending phase, during which only moves in $\mathcal{N}_+(x^k)$ are allowed. During this phase, the objective function value is monotonically increased, along with the total resources used. The ascending phase stops when, for a given solution $x^k$, its neighborhood $\mathcal{N}_+(x^k)$ is empty. Note that, during the ascending phase, feasibility with respect to the resource consumption constraints can be lost; and (ii) a descending phase, during which only moves in $\mathcal{N}_-(x^k)$ are allowed. The descending phase is characterized by a monotonic decrease of the objective function value along with a monotonic decrease of the total resources used. This phase is aimed at restoring full feasibility of the solution. The descending phase terminates when $\mathcal{N}_-(x^k) = \emptyset$.

If we indicate with $n = \sum_{i=1}^{m}$ the total number of bits of a binary vector representing a solution, an estimate of the size of each neighborhood can be given by $|\mathcal{N}_*| = n - m$, where $m$ is the number of modules in the system. Consequently, we can estimate the computational complexity of the neighborhood exploration as $O(n\kappa)$, where $O(\kappa)$
indicates the computational complexity associated to the objective function evaluation, which changes depending on the system configuration.

**Remark 1.** One of the benefits of defining the “allowed infeasible space” along with the ascending phase is that the algorithm can easily escape from a local maximum by moving to a non-feasible, though allowed, solution in $X \setminus X$. Due to the monotonic property of the objective function, solutions in $X \setminus X$ are more attractive than solutions in $X$.

**Remark 2.** Since a stopping criterion of the algorithm is established by fixing the maximum number of excursion into the $X \setminus X$ region, the algorithm always terminates. Along with the neighborhood definition, we define a “merit score” to choose, among all the possible moves allowed in the current neighborhood, the most attractive one. We define two merit scores $\sigma_j^+$ and $\sigma_j^-$, one for each of the two neighborhoods, and we select the move $j$ that maximizes its corresponding score $\sigma_j^+$ if $\bullet = +$, and $\sigma_j^-$ if $\bullet = -$. The two scores are:

$$\sigma_j^+ = \frac{f_+(x) - f_-(x)}{\sum_{k=1}^K a_{kj}},$$

$$\sigma_j^- = \frac{\sum_{k=1}^K a_{kj}}{f_+(x) - f_-(x)},$$

where $f_+(x) = f(x_1, \ldots, x_j-1, 1, x_{j+1}, \ldots, x_n)$, $f_-(x) = f(x_1, \ldots, x_j-1, 0, x_{j+1}, \ldots, x_n)$, for all $x \in B^n$.

The most attractive move, selected via (2) or (3), needs to be not-tabu, which is, the algorithm uses a short-term mechanism that prevents revisiting previously visited points. The next section provides more details about how the short-term memory mechanism is used.

### 3.2. Short-term memory and working memory

Memory plays a critical role in TS. Consequently, the way memory mechanisms are implemented is directly linked to the effectiveness of the algorithm itself.

The proposed algorithm uses two mechanisms to balance the use of short-term memory with working memory. More specifically, we implement a regular Tabu List ($\mathcal{TL}$), aimed at exploiting short-term memory, and an Active Tabu List ($\mathcal{ATL}$), which contributes to making a more effective use of working memory. Both kinds of memories are intensively used at each iteration of the algorithm. $\mathcal{ATL}$ is used to define “active chains”, which are subsets of consecutive elements of the active tabu list. Let us call $AC \subseteq \mathcal{ATL}$ any subset of consecutive elements of $\mathcal{ATL}$, and let us indicate with $\mathcal{AC}$ the set of all the active chains existing in $\mathcal{ATL}$. At each iteration of the algorithm the opposite of the current move, if present, is removed from $\mathcal{ATL}$. Consequently, some of the active chains are reduced in size. It can be proved that, as long as $|AC| \geq 1$, $\forall AC \in \mathcal{AC}$, the algorithm will not visit previously visited points.

At each iteration, both $\mathcal{ATL}$ and $\mathcal{TL}$ are updated. Below we present the updating procedure, where $\bullet j$ indicates the direction along which a move is taken with respect to the current solution $x^k$, either in $\mathcal{N}_+(x^k)$ or in $\mathcal{N}_-(x^k)$.

**procedure update $\mathcal{ATL}$ & $\mathcal{TL}$ ($\bullet j$);**

**begin**

$\mathcal{ATL} \leftarrow \mathcal{ATL} \cup \{ \bullet j \}$

if $(\bullet j) \in \mathcal{ATL}$

$\mathcal{ATL} \leftarrow \mathcal{ATL} \setminus \{ (\bullet j) \}$

createAC $(\bullet j)\quad (\bullet j)$

$\mathcal{AC} \leftarrow \mathcal{AC} \cup AC$ $\quad$ chain from $\bullet j$ to $\bullet j$

$\mathcal{AC} \leftarrow \mathcal{AC} \setminus AC$ $\quad$ add new chain to $\mathcal{AC}$

if $\exists AC^* \in \mathcal{AC} : |AC| = 1$

$\mathcal{TL} \leftarrow \mathcal{TL} \cup AC^*$ $\quad$ length 1

then $\mathcal{TL} \leftarrow \mathcal{TL} \cup \{ \bullet j \}$ $\quad$ strong tabu

**end**

After each iteration, the last move \( \bullet j \) is added in the last position of \( ATL \). In addition, starting from the end of \( ATL \), we execute a backward search of the opposite move \( \bullet j \). If such move is already into the \( ATL \), we define an active chain \( AC \), made up by all the elements in \( ATL \) included between \( \bullet j \) and \( \bullet j \). If we consider that, in the worst case, \(|ATL| = n\), which is, all the elements could be in the Active Tabu List, the computational complexity of the procedure \( update_{ATL & TL}(\bullet j) \) is \( O(n) \).

The benefit of the use of \( ATL \) is two-fold: on the one hand, it guarantees that the search path will not include the same point twice, since this would correspond to eliminating, one by one, all the elements of an \( AC \) within \( ATL \) (which is not allowed by labeling the last element of the chain as “strong tabu”); on the other hand, it allows to make the use of \( TL \) less restrictive and, consequently, to explore more thoroughly the search space, since an element is labeled tabu only when it forms a one-element chain within \( ATL \). \( ATL \) allows the implementation of an “intermediate” memory, or working memory, hence increasing the effectiveness of the memory usage.

Some of the most relevant parameters of the mechanism are the tabu tenure \( \theta \), which is the number of periods a new element is kept in \( TL \), and the strong tabu tenure \( \tau \), which is the number of periods the last element of an \( AC \) is kept in \( TL \) (usually, \( \theta < \tau \)). How to assign values to these two parameters will be discussed in Section 4.1.

3.3. Long-term memory

Long-term memory mechanisms are used to “permanently” remember some of the points visited during the search. More specifically, long-term memory keeps track of a pool of elite solutions which are then used to enforce promising trajectories within the search space.

We implement a long-term memory mechanism by means of a technique known as “path relinking” [15–17], which basically consists of identifying two points along the search and then force the algorithm to follow a trajectory path that joins these two points. Our implementation of the mechanism within the reliability problem is illustrated in the procedure presented below. In the following, \( S \) indicates the reference set, the set of all feasible points \( x^k \) visited along the path such that \( x^{k+1} \in X \setminus X^t \), \( x^t \) indicates the target point (the best solution found) and \( x^i \in S \) indicates the initial point of the trajectory. The search is biased in such a way that, starting from \( x^i \), \( \forall x^i \in S \), either \( x^t \) is reached within a maximum number of iterations, or a new starting point for the TS phase is obtained.

\[
\text{procedure } activateLTM; \\
\text{begin} \\
\text{while } |S| > 1 \text{ do} \\
\text{set } k \leftarrow 0 \text{ iterations counter} \\
x^i \leftarrow \max_{x^w \in S} \{d_{x^w} \} \text{ initial point} \\
x^k \leftarrow x^i \\
\text{while } (x^k \neq x^i \text{ & } k < d(x^t, x^i)) \text{ do} \\
\text{select } \bullet j \text{ in } \mathcal{N}_{x^t}(x^k) \text{ find next move} \\
\text{update } \mathcal{IL} \text{ & } TL \text{ update memory} \\
\text{if } z(x^k) > z^*, \text{ update best result} \\
k \leftarrow k + 1 \\
\text{end while} \\
S \leftarrow S \{x^i\} \text{ remove point from } S \\
\text{end while} \\
\text{end} \\
\text{end} \\
\text{end}
\]

The distance function \( d \) between two vectors in \( \mathbb{R}^n \) is a measure of dissimilarity and is computed as

\[
d_{x^t}^{x^i} = \sum_{j=1}^{n} h_j,
\]

where

\[
h_j = \begin{cases} 
1 & \text{if } x_j^t \neq x_j^k, \\
0 & \text{otherwise}.
\end{cases}
\]
The procedure \textsc{activateLTM} strikes the balance between intensification, by setting the best point found as target, and diversification, by choosing as starting point the furthest point from the target, as well as by “feeding” a new, different, point to TS, hence exploring a new section of the searching space.

Finally, while $\mathcal{N}_{\pm}(x^k) \equiv \mathcal{N}_{\pm}^{	ext{guid}}(x^k)$, the \textit{guided} neighborhood is defined with respect to the target vector $x^t$ as

$$
\mathcal{N}_{\pm}^{	ext{guid}}(x^k) = \{ x^w : d_{x^w}^x = 1, d_{x^w}^{x^t} \leq d_{x^w}^x \}.
$$

It is possible to evaluate the computational complexity of this phase of the algorithm as $\mathcal{O}(|\mathcal{S}|n^2 \kappa)$, since for each point in the “elite set” we need to estimate its neighborhood a number of times equal to the distance between the point itself and the best solution, indicated by $d(x^t, x^i)$, where $d(x^t, x^i) = n - m = \mathcal{O}(n)$ in the worst case.

4. Results

4.1. Problems from the literature

In this section we present the results of the algorithm on some problems drawn from the literature on software optimization. The benchmarking tests have been carried out on a Pentium 4 Workstation with 256 MB of RAM. The TS algorithm has been implemented and compiled using Microsoft C#.NET 2003. The software used to solve the benchmark problems P1–P4 and the random generated instances can be freely downloaded, for further testing, at \texttt{http://paginas.ccm.itesm.mx/~mcaserta/}. This software, written by the authors, has been developed using the Microsoft .NET 2003 platform and written in C# and is freely distributable under the GNU General Public License published by the Free Software Foundation [18] at \texttt{http://www.gnu.org/copyleft/gpl.html}.

Berman and Ashrafi [6] present four different models, corresponding to four system configurations. We solved problems P1–P4 with TS and the algorithm always found the global optimal solution in less than 0.1 CPU seconds. Table 1 reports the problem parameters, as in Berman and Ashrafi [6]. The system is made up by three modules arranged in series, and eight versions to be distributed among the modules. There is only one knapsack-type constraint, with a total budget of 10. As mentioned, we reach the optimal solution of 0.9359 $(x_{12} = x_{13} = x_{21} = x_{23} = x_{31} = 1)$ in less than 0.1 CPU seconds.

Finally, Table 2 illustrates the importance of a correct choice of the tabu tenure $\theta$. The table shows how the quality of the solution and the thoroughness of the search change as a function of the value of $\theta$, while $\tau$ is kept fixed to 5.

<table>
<thead>
<tr>
<th>Version</th>
<th>Module 1</th>
<th>Module 2</th>
<th>Module 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$r_j$</td>
<td>$e_j$</td>
<td>$r_j$</td>
</tr>
<tr>
<td>1</td>
<td>0.90</td>
<td>3</td>
<td>0.95</td>
</tr>
<tr>
<td>2</td>
<td>0.80</td>
<td>1</td>
<td>0.80</td>
</tr>
<tr>
<td>3</td>
<td>0.85</td>
<td>2</td>
<td>0.70</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\theta$</th>
<th>No. of iterations</th>
<th>Best result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
<td>0.9031</td>
</tr>
<tr>
<td>1</td>
<td>64</td>
<td>0.9031</td>
</tr>
<tr>
<td>2</td>
<td>58</td>
<td>0.9359</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>0.9359</td>
</tr>
<tr>
<td>4</td>
<td>28</td>
<td>0.9316</td>
</tr>
<tr>
<td>5</td>
<td>24</td>
<td>0.9124</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>0.9124</td>
</tr>
</tbody>
</table>

Note, first, that without the use of the short-term memory mechanism ($\theta = 0$), the algorithm does not find the global optimum, even though 64 points are visited and the working memory mechanism is enforced ($\tau = 5$). However, when the short-term memory mechanism is turned on, the algorithm finds the global optimum for $\theta = 2, 3, \text{and} 4$, visiting less points than in the previous case. However, when the tabu tenure becomes too restrictive with respect to the problem size, the performance deteriorates and, both with $\theta = 5$ and 6, TS does not find the global optimum.

### 4.2. Random generated problems

In order to further test the effectiveness of the algorithm, with special emphasis on very large instances of (SRP), we devised a mechanism to generate random instances. It takes as input the number of modules in the system ($m$), the maximum number of versions per module ($v$), and the number of resource constraints $K$, and defines $m_i$, the total number of version within each module $i$, $r_{ij}$, the reliability value of each version $j$ within module $i$, $c_{ij}^k$, the resource usage of each version $j$ within module $i$ in the constraint $k$, and $b_k$, the right-hand side value of each constraint. The random values are uniformly generated as follows:

- $m_i \in [2, v]$,
- $r_{ij} = 0.80 + (0.99 - 0.80) \omega$,
- $c_{ij}^k = 1 + 4 \varepsilon$,
- $b_k = \frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{m_i} s_{ij}^k + \max [3 \times \min_j {s_{ij}^k}, 2 \times \max_j {s_{ij}^k}]$

for $i = 1, \ldots, m$, $j = 1, \ldots, v_i$ and $k = 1, \ldots, K$, where $\omega$ and $\varepsilon$ are random numbers in $[0, 1]$.

Along with the random generation process, we implemented a branch-and-cut code to solve problems to optimality. The heuristic solution obtained by the proposed algorithm was then compared with the global optimum. The branch and bound algorithm uses the Cbc module of the COIN-OR Library [19].

To perform these tests, we chose problems of type P2, where only one function is performed within the software system and redundancy is allowed. The problem can be formulated as follows:

(P2): $\max \ R_s = \prod_{i=1}^{m} \left( 1 - \prod_{j=1}^{m_i} (1 - r_{ij})^{v_{ij}} \right)$

s.t.

\[
\begin{align*}
\sum_{i=1}^{m} \sum_{j=1}^{m_i} c_{ij}^k x_{ij} &\leq b_k, & k &= 1, \ldots, K, \\
\sum_{j=1}^{m_i} x_{ij} &\geq 1, & i &= 1, \ldots, m, \\
x &\in \mathbb{B}^m.
\end{align*}
\]

The random instances generated are of medium and large scale. We randomly generated and solved problems with the following combinations of $m$ and $v$:

- $m \in M = \{30, 50, 70\}$,
- $v \in V = \{10, 30, 50\}$,
- $k \in K = \{1, 3, 5\}$.

In addition, for each combination of parameters $(m, v)$, we randomly generated 10 different instances. This implies that we generated $10 \times 3 \times 3 \times 3 = 270$ different instances.\(^1\)

The study on randomly generated instances had a two-fold objective: on the one hand, we wanted to test the effectiveness of the algorithm in terms of solution quality by comparing the solution obtained by the algorithm on each instance with the global optimal solution given by the branch-and-cut solver as well as by our implementation of Berman and Ashrafi [6] dynamic algorithm, in terms of CPU time and solution quality. On the other hand, we aimed at testing the effect that two algorithmic parameters, $\theta$ and $\tau$, have on the solution quality.

---

\(^1\) These instances can be downloaded at http://www.ccm.itesm.mx/goal/.
Table 3
Comparison of algorithms performances for the SRP

<table>
<thead>
<tr>
<th>Problem Class</th>
<th>Problem size</th>
<th>B&amp;C Time ( a )</th>
<th>B&amp;A Time ( a )</th>
<th>( \gamma^b )</th>
<th>C&amp;M Time ( a )</th>
<th>( \gamma^b )</th>
<th>Avg. iters.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_1 )</td>
<td>( m ) = 30</td>
<td>( m_i ) = 10</td>
<td>170</td>
<td>1311</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( R_2 )</td>
<td>( m ) = 50</td>
<td>( m_i ) = 11</td>
<td>1474</td>
<td>74</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( R_3 )</td>
<td>( m ) = 70</td>
<td>( m_i ) = 9</td>
<td>4231</td>
<td>801</td>
<td>0</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( a \) CPU time on a Pentium4 1.2 GHz Linux Workstation.

\( b \) Computed as in Eq. (6).

A clear correlation between the problem size, in terms of total number of variables, and the correct value of \( \theta \) and \( \tau \) emerges from the analysis. We ran the whole set of random generated instances with different values of \( \tau \) and \( \theta \) such as

\[
\tau \in \left\{ \frac{\bar{b} \times \alpha}{2 \bar{a}}, \frac{n}{2} \frac{\bar{m}_i (m-1)}{2} \right\},
\]

where \( \bar{m}_i \) is the average number of versions per module, \( \bar{b} \) is the average resource available and \( \bar{a}_i \) the average amount of resource used by a version within the system. We found that a “good” choice of values of \( \tau \) and \( \theta \) are given by

\[
\tau = \min \left\{ \left[ \frac{\bar{b} \times \alpha}{2 \bar{a}} \right], \left[ \frac{n}{2} \frac{\bar{m}_i (m-1)}{2} \right] \right\},
\]

(4)

\[
\theta = \max \left\{ \frac{\tau}{2}, 2 \right\}.
\]

(5)

Finally, we measured the effectiveness of the algorithm in terms of solution quality by randomly generating 270 instances under the aforementioned scheme and grouped together the 270 instances in three different classes, labeled \( R_1 \)–\( R_3 \). To provide an indication of the size of the search space, we also compute the average number of versions per module. Furthermore, the results are averaged with respect to the number of constraints, which is, each combination of \( m \) and \( m_i \) gives rise to three different instances, with \( k = 1, 3, \) and \( 5 \).

Table 3 summarizes the most relevant results of this part of the experiment. In the table, the first column presents the instance class name, while columns two and three indicate how many modules and how many version per modules, as an average, are in the system. The fourth column presents the computational time of the branch and cut scheme, measured as CPU time. Under the column B&A we present the results of our implementation of Berman and Ashrafi [6] dynamic algorithm, in terms of CPU time and accuracy level of the solution (a value of NA indicates that the algorithm did not terminate within the maximum allotted time, fixed to 1500 s). Finally, the last three columns under C&M present the results of the proposed algorithm, in terms of CPU time, solution quality and number of iterations required. It is worth noting that all the measures of time are referred to the total running time of the algorithm and that our algorithm was stopped after at most 300 CPU seconds. The solution quality \( \gamma \) is computed, for each instance class, as

\[
\gamma = \frac{1}{T} \sum_{t=1}^{T} \frac{z^*_t - z^*_i}{z^*_i},
\]

(6)

where \( t = 1, 2, \ldots, T \), with \( T = 10 \), accounts for the fact that 10 instances are generated for each pair \( (m, v) \). In addition, \( z^*_t \) indicates the best solution found on instance number \( t \) by the branch-and-cut scheme. This solution is either the
Table 4
Comparison of algorithms performances on instance with different number of constraints

<table>
<thead>
<tr>
<th>Problem size</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>Results</td>
</tr>
<tr>
<td></td>
<td>Time&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
<tr>
<td>70</td>
<td>14.4</td>
</tr>
<tr>
<td>3</td>
<td>15.1</td>
</tr>
<tr>
<td>5</td>
<td>15.7</td>
</tr>
<tr>
<td>Avg</td>
<td>15.1</td>
</tr>
<tr>
<td>19</td>
<td>20.3</td>
</tr>
<tr>
<td>3</td>
<td>18.1</td>
</tr>
<tr>
<td>5</td>
<td>19.4</td>
</tr>
<tr>
<td>Avg</td>
<td>19.3</td>
</tr>
<tr>
<td>33</td>
<td>25.1</td>
</tr>
<tr>
<td>3</td>
<td>27.2</td>
</tr>
<tr>
<td>5</td>
<td>26.4</td>
</tr>
<tr>
<td>Avg</td>
<td>26.2</td>
</tr>
</tbody>
</table>

<sup>a</sup>CPU time on a Pentium4 1.2 GHz Linux Workstation.
<sup>b</sup>Computed as in Eq. (6).

global optimum, if found in less than the maximum CPU time (set to 10,000 s) or the best lower bound reached by the scheme within the allotted time. Finally, $z^*$ is the best solution found on the same instance by the heuristic algorithms, Berman and Ashrafi [6] when $\bullet = B&A$, or the proposed algorithm, when $\bullet = C&M$.

The results presented in Table 3 provide a measure of the effectiveness of the algorithm in terms of solution quality. It is worth noting that a value of $\gamma = 0$ implies that the global optimum has been found for the 10 instances of a given class. The proposed algorithm finds the global optimum solutions for medium size instances and near-optimal solutions for large scale problems. The value of the gap between the heuristic solution of the algorithm and the global optimum (or the best lower bound) is less than or equal to 13%. Furthermore, it is possible to conclude that the algorithm outperforms Berman and Ashrafi [6] algorithm, both in terms of solution quality and computational time.

Finally, in order to show that the quality of the algorithmic solution is not affected by the number of constraints, we illustrate the detailed behavior of the algorithm on problems of class $R_3$ with respect to the number of constraints in each instance. Table 4 illustrates the results collected on 90 instances with different values of $\bar{m}_i$ and $k$. For reasons of space, we present the results only with respect to problems of class $R_3$. However, a similar behavior was observed on problems of class $R_1$ and $R_2$. From the table, it is possible to see that the algorithmic performance, both in terms of computational time and solution quality, are insensitive to the value of $k$, the number of constraints in the system.

5. Conclusions

We propose a new metaheuristic algorithm for the general reliability problem. The algorithm makes effective use of short term, working and long-term memories and thoroughly explores the search space, without getting trapped into local attractors.

The algorithm has been tested on four benchmark problems from the software reliability literature [6], and for each one of the problems, it finds the global optimal solution in less than 0.1 CPU seconds.

In order to further test the robustness of the proposed algorithm, we run a thorough test on a pool of random generated instances of a software reliability problem. We were able to compare the solution obtained by the proposed algorithm with the global solution, obtained using a branch-and-cut solver, up to problems with approximately 1300 binary variables. Problems of this size were solved by the proposed algorithm in 3 CPU seconds, while the branch-and-cut code took around 2500 s. We further solved larger instances, with up to 2300 variables, and, even though we could not get the global optimum solution using the branch-and-cut code, we used the dual value provided by the solver to get
a measure of the distance between the solution found by our algorithm and the best upper bound obtained. On these very large scale instances we were able to get a solution very close to the optimum (dual gap of 13%).

The algorithm proved to be robust with respect to its parameters. However, as briefly illustrated in Section 4, a proper choice of $\theta$ and $\tau$, which govern the intensity of the memory usage, seems to enhance the performances of the algorithm.

Acknowledgments

This work was carried out while both authors worked at “Instituto Tecnologico y de Estudios Superiores de Monterrey”, Campus Mexico City, Department of Industrial and Systems Engineering, Faculty of Engineering and Architecture, Mexico.

References