OR Spektrum (1995) 17:67–86

OR Spektrum
© Springer-Verlag 1995

Local search with memory: benchmarking RTS

Roberto Battiti[1], **Giampietro Tecchiolli**[2]

[1] Dipartimento di Matematica, Università di Trento, I-38050 Povo (Trento), Italy (e-mail:battiti@science.unitn.it)
[2] Istituto per la Ricerca Scientifica e Tecnologica, I-38050 Povo (Trento), Italy

Received: 19 October 1993 / Accepted: 29 November 1994

Abstract. The purpose of this work is that of presenting a version of the Reactive Tabu Search method (RTS) that is suitable for constrained problems, and that of testing RTS on a series of constrained and unconstrained Combinatorial Optimization tasks. The benchmark suite consists of many instances of the N-K model and of the Multiknapsack problem with various sizes and difficulties, defined with portable random number generators. The performance of RTS is compared with that of Repeated Local Minima Search, Simulated Annealing, Genetic Algorithms, and Neural Networks. In addition, the effects of different *hashing* schemes and of the presence of a simple "aspiration" criterion in the RTS algorithm are investigated.

Zusammenfassung. Diese Arbeit entwickelt eine Variante der sogenannten „Reactive Tabu Search“ Methode (RTS), die auch für Optimierungsprobleme mit Nebenbedingungen geeignet ist. Das Verhalten dieser RTS Variante wird anhand einer Reihe von kombinatorischen Optimierungsproblemen mit und ohne Nebenbedingungen ausgetestet. Die Benchmark besteht aus einigen Beispielen des N-K Modells und des Multiknapsack Problems mit verschiedenen Größen und Schwierigkeitsstufen, die mit portablen Zufallszahlgeneratoren definiert werden. Ein Vergleich zwischen der Leistung der RTS Variante und der Leistung von Repeated Local Minima Search, Simulated Annealing, genetischen Algorithmen und neuronalen Netzen wird durchgeführt. Anschließend werden die Auswirkungen verschiedener Hashingschemata und eines ‚aspiration‘ Kriteriums im RTS Algorithmus untersucht.

Key words: Heuristic algorithms, local search, reactive tabu search, N-K model, multi-knapsack problem

Schlüsselwörter: Heuristics, local search, reactive tabu search, N-K Modell, Multiknapsack Problem

Correspondence to: R. Battiti

1. Introduction

While the offer of heuristic strategies for optimization is widening (Tabu Search, Simulated Annealing, Genetic Algorithms, and Neural Networks are some examples), the task of choosing the most appropriate algorithm for a given problem is difficult. A suitable benchmark must at least consider the properties of efficacy (ability to find a suitable solution), efficiency (time and memory requirements), simplicity of implementation and "stability" (robustness with respect to parameter changes) of the various strategies.

We are aware that the performance of different algorithms varies greatly with the difficult of problems so that the search for the "best algorithm" for all problems is a sterile exercise. On the other hand, there are many applicative cases when the user knows that the task possesses a local structure (a property that can be measured with the correlation between neighboring points, see for example Sect. 3) and is therefore motivated to try heuristic strategies based on local search, but he does not know the detailed form of the function to be optimized or specific properties that suggest problem-dependent heuristics. For example, if a user knows that the local minimum points have a regular structure on a grid parallel to the coordinate axes, or that a function of N variables is a linear combination of simpler functions of subsets of variables, it is not difficult to predict the success of "cross-over" based genetic schemes. But, in this case, even simpler *ad hoc* techniques can be sufficient. In many cases where the structure of the "fitness landscape" is complex some methods require a careful tuning of the algorithms and parameters that adds a large amount of computing time to the total spent on the task.

To cover a wide spectrum of problem sizes and difficulties, we focus our attention onto a set of benchmark tasks designed with the following characteristics:

1. Partially random generation. This is a requirement of "fairness" with respect to different algorithms and of convenient construction via portable pseudo-random genera-

tors. Clearly the randomness has to be partial, otherwise random search will be the winning algorithm for a completely random function.
2. Local structure with abundance of sub-optimal local minimizers. The neighborhood (or set of basic moves) is designed so that there is a non-zero correlation between the function values of neighboring points, while the abundance of local minimizers avoids the trivial "local search" solution.
3. Mixed (tunable) difficulty and size. The problems are constructed in a parametric way so that the hardness of the task (or the "roughness" of the "fitness landscape") can be varied and the performance can be measured on a wide difficulty spectrum.

In detail, we present two sets of benchmark tasks for unconstrained and constrained optimization, respectively. The first set is based on the N-K model introduced in [20], the second set is based on the Multiknapsack problem. These two sets are further separated into "small" tasks, used to measure the efficacy in reaching the global optimum on a large set of problems for which the exact solution is known, and "large" tasks, for which the global optimum is unknown and that permit the benchmark of a limited number of problems.

The memory-based optimization algorithm *RTS* is presented and discussed in Sect. 2, while the other algorithms used for the comparisons are briefly sketched in the experimental sections, mainly to define precisely the versions of the algorithms chosen from the existing literature.

The ensuing structure of the paper closely reflects the above subdivision of benchmark tasks. The first part of the benchmark (Sect. 3) is dedicated to a description of the task construction for the N-K model. Then, the algorithms that have been tested are discussed in Sect. 4, and the results on the "small" tasks (Sect. 5), and on the "large" tasks (Sect. 6) are analyzed.

The second part of the benchmark repeats the above sections for the Multiknapsack problem. First the 0-1 multidimensional knapsack tasks are illustrated (Sect. 7), then the changes in the applied algorithms to deal with the constraints are described (Sect. 8). Finally, the results on the small (Sect. 9), large (Sect. 10), and "strongly correlated" (Sect. 11) problems are presented.

2. Memory-based optimization: RTS

In this section we summarize the version of *Tabu Search* proposed by the authors in [4] with the term *Reactive Tabu Search* (RTS). The parallel properties of the RTS scheme have been investigated in [3]. Because this paper is dedicated to benchmarking RTS we completely describe the algorithm but omit the detailed analysis given in the cited papers because it is not necessary for the scope of this work. The Tabu Search technique (TS) represents a *meta-strategy* that permits to complement a local search heuristic based on a specific neighborhood with memory-based mechanisms designed to avoid cycles, see [13] and [14] for two seminal papers and [34] for a tutorial.

Let us define the notation. Given an instance of a Combinatorial Optimization (CO) problem, i.e., a pair (F, E) where F is a set of feasible points with finite cardinality and E is the cost function to be *minimized*: $E{:}F \rightarrow R^1$, the *neighborhood* $N(f)$ associates to each point f a subset of F: $N{:}F \rightarrow 2^F$. A point f is *locally optimal with respect to* N, or a *local minimizer* if: $E(f) \leq E(g)$ for all $g \in N(f)$. For the following discussion the *neighborhood* $N(f)$ is defined as the set of points that can be obtained by applying to f a set of *elementary moves* $\mathcal{M}$

$$N(f) = \{g \in F \text{ such that } g = \mu(f) \text{ for } \mu \in \mathcal{M}\}$$

We limit our consideration to the case where F is the set of all binary strings with a finite length L: $F = \{0,1\}^L$ and the *elementary moves* $\mu_i (i = 1,\ldots,L)$ change the i-th bit of the string $f = [f_1,\ldots f_i,\ldots f_L]$

$$\mu_i([f_1,\ldots,f_i,\ldots,f_L]) = [f_1,\ldots,\bar{f}_i,\ldots,f_L] \tag{1}$$

where $\bar{f}_i$ is the negation of the i-th bit: $\bar{f}_i \equiv (1 - f_i)$. Obviously, two moves commute and μ_i is idempotent (i.e. $\mu_i^2 = 1$, the identity move) and therefore its inverse is $\mu_i^{-1} = \mu_i$.

The TS scheme uses an iterative greedy search algorithm (like "steepest descent") to bias the search toward points with low E values. In addition, the TS incorporates strategies to avoid the occurrence of cycles. The two goals are attained by using the following design principles:

Modified greedy search. At each step of the iterative process, the best move is selected from a set of admissible elementary moves that lead to points in the *neighborhood* of the current state. The best move is the one that produces the lowest value of the cost function E. Note that the best move is executed even if E increases with respect to the value at the current point, while the standard greedy search technique stops if the best move increases E. Increases are allowed because they are necessary to exit from local minimizers of E.

Cycle avoidance. The *inverses* of the moves executed in the most recent part of the search are prohibited (the names "tabu" or "taboo" derive from this prohibition).

In detail, at a given iteration t of the search, the set of moves $\mathcal{M}$ is partitioned into the set $\mathcal{T}^{(t)}$ of the *tabu* moves, and the set $\mathcal{A}^{(t)}$ of the admissible moves, i.e., of the moves that can be applied to the current point. We use superscripts with parenthesis for quantities that depend on the iteration.

At the beginning, the search starts from the initial configuration $f^{(0)}$, that can be generated randomly, and all moves are admissible: $\mathcal{A}^{(0)} = \mathcal{M}$, $\mathcal{T}^{(0)} = \emptyset$. At a given iteration t, the successor of the current point is obtained by selecting the best move $\mu^{(t)}$ from the set $\mathcal{A}^{(t)}$:

$$f^{(t+1)} = \mu^{(t)}\left(f^{(t)}\right) \text{ where } \mu^{(t)} = \arg \min_{\nu \in \mathcal{A}^{(t)}} E\left(\nu\left(f^{(t)}\right)\right)$$

The rules of the algorithm must assure that the set of admissible moves does not become empty: $\mathcal{A}^{(t)} \neq \emptyset$. If the admissible moves are expensive to evaluate, for example if $\mathcal{A}^{(t)}$ is very large, one can sample $\mathcal{A}^{(t)}$ randomly and take the best out of a subset of $\mathcal{A}^{(t)}$, this strategy has been used with success in [5]. The set of points $f^{(t)}$ obtained by the above discrete dynamical process is called a *trajectory*.

Let us now motivate the introduction of prohibited moves. In isolation, the "modified greedy search" principle *can* generate cycles. Let us suppose that the current point $f^{(t)}$ is a strict local minimizer: the cost function at the next point must increase: $E(f^{(t+1)})=E(\mu^{(t)}(f^{(t)}))>E(f^{(t)})$, and there is the possibility that the move at the next step will be its *inverse* ($\mu^{(t+1)}=\mu^{(t)^{-1}}$) so that the state after two steps will come back to the starting configuration

$$f^{(t+2)}=\mu^{(t+1)}\left(f^{(t+1)}\right)=\mu^{(t)^{-1}}\circ\mu^{(t)}\left(f^{(t)}\right)=f^{(t)}$$

At this point, if the set of admissible moves is the same, the system will be "trapped" forever in a cycle of length 2. One direct way to discourage cycles is to modify the set $\mathcal{A}^{(t)}$ by prohibiting the inverses of the moves executed in the most recent part of the search. The prohibition must be canceled after a certain number of iterations T because the tabu moves can be necessary to reach the optimum in a later phase of the search.

A second reason for allowing the transfer of moves from the set $\mathcal{T}^{(t)}$ to the set $\mathcal{A}^{(t)}$ is to avoid $\mathcal{A}^{(t)}$ becoming empty, in the case of a set $\mathcal{M}$ with finite cardinality. In the above example, after the execution of movement $\mu^{(t)}\in\mathcal{A}^{(t)}$, one updates the $\mathcal{A}^{(t)}$ and $\mathcal{T}^{(t)}$ sets as follows:

$$\mathcal{T}^{(t+1)}\leftarrow\mathcal{T}^{(t)}\cup\left\{\mu^{(t)^{-1}}\right\}\setminus \left\{\mu^{(\tau)^{-1}}\in\mathcal{T}^{(t)}\text{for }\tau\text{ such that }\tau\le(t-T)\right\} \tag{2}$$

$$\mathcal{A}^{(t+1)}\leftarrow\mathcal{A}^{(t)}\setminus\left\{\mu^{(t)^{-1}}\right\}\cup \left\{\mu^{(\tau)^{-1}}\in\mathcal{T}^{(t)}\text{for }\tau\text{ such that }\tau\le(t-T)\right\} \tag{3}$$

The prohibition period T, i.e. the number of iterations that a move remains in the $\mathcal{T}^{(t)}$ set, is also called *list size* in the original terminology, a term referring to a realization of the scheme in which the forbidden moves are inserted into a first-in-first-out list (i.e., a queue of length T where a move enters as soon as it has been executed and exits after T steps).

The value of T is related to the minimum number of iterations that must separate the repetition of the same configuration on a trajectory. In fact, for a constant T such that the set of admissible moves does not become empty, it is straightforward to derive the following requirements (see [5] for the proof):

- T must be large to avoid cycles. In detail cycles of length R are impossible if T is larger than $(R/2)-1$ (note that R is even for binary strings).
- T must be sufficiently small to avoid over-constraining the trajectory, and in any case it must be smaller than or equal to $L-2$, so that at least *two* moves are admissible. If the actual value of T is close to $L-1$, the admissible moves are severely reduced and the search trajectory is too constrained to permit effective searches. In particular, if $T=L-1$, after the first T iterations only one move is available in $\mathcal{A}$, so that from this point on the trajectory does not depend on the E values anymore: the sequence given by the first $T+1$ moves will be replicated forever and a cycle of length $R=2L$ is generated.

For the following discussion, it is usefull to introduce a time-dependent prohibition period $T^{(t)}$, and to characterize in an equivalent way the sets of the prohibited and admissible moves as:

$$\mathcal{T}^{(t)}=\{\mu\in\mathcal{M}\text{ such that its most recent use has been at time }\tau\ge(t-T^{(t)})\} \tag{4}$$

$$\mathcal{A}^{(t)}=\mathcal{M}\setminus\mathcal{T}^{(t)} \tag{5}$$

2.1. The reactive tabu search (RTS) algorithm

The basic Tabu Search mechanism illustrated in Sect. 2 cannot guarantee the absence of cycles. In fact the longest possible cycle in the search space $F=\{0,1\}^L$ has length $R=2^L$, and therefore R is much larger than $2L$, for large L values.

In addition, the choice of a fixed T without *a priori* knowledge about the possible search trajectories that can be generated in a given (F, E) problem is difficult. If the search space is inhomogeneous, a size T that is appropriate in a region of F may be inappropriate in other regions. For example, T can be insufficient to avoid cycles (if T is small), or too large, so that only a small fraction of the movements are admissible and the search is inefficient. Even if a constant T is suitable, its value depends on the specific problem.

The RTS scheme [4] proposes an additional mechanism to deal with cycles that are not avoided by using the basic Tabu scheme and a way to change T during the search so that the value $T^{(t)}$ is appropriate to the local structure of the problem. In the RTS scheme, the most recent iteration when each move μ_i has been applied is recorded and each configuration $f^{(t)}$ touched by the search trajectory is stored in memory with the most recent time when it was encountered. Let us introduce the functions:

- $\Lambda(\mu)$: the last iteration when μ has been used ($\Lambda(\mu)=-\infty$ if μ has never been used)
- $\Pi(f)$: the last iteration when f has been encountered ($\Pi(f)=-\infty$ if f has not been encountered or it is not in the memory).
- $\Phi(f)$: the number of repetitions of configuration f in the search trajectory ("repetition counter"). At the beginning $\Phi(f)=0$ for all configurations.

We admit the possibility that f has been encountered but is not stored in the memory (in this case $\Pi(f)=-\infty$ and $\Phi(f)=0$). In fact the allowed memory size can be insufficient or the algorithm can cancel the memory content at specific times (in particular see the function **diversify_ search** of Fig. 3).

The prohibition period $T^{(t)}$ is initialized with a small value (e.g., $T^{(0)}\leftarrow 1$), and then adapted by *reacting* to the occurrence of repetitions. Note that checking the *tabu* status of a move requires only a few CPU cycles if the function $\Lambda(\mu)$ is realized with an array in memory.

The structure of the *Reactive Tabu Search* for the case of a CO problem on the set of fixed-length binary strings is described in Figs. 1–3. The algorithm is illustrated with simple selection (conditional) and iteration keywords (**if**

```
procedure reactive_tabu_search
(Initialize the data structures for tabu:)
$t \leftarrow 0$                         (iteration counter)
$T^{(0)} \leftarrow 1$                   (prohibition period)
$t_T \leftarrow 0$                       (last time T was changed)
$\mathscr{C} \leftarrow \emptyset$       (set of often-repeated configurations)
$R_{ave} \leftarrow 1$                   (moving average of repetition interval)
$f^{(0)} \leftarrow$ random $f \in F$    (initial configuration)
$f_b \leftarrow f^{(0)}$                 (best so far f)
$E_b \leftarrow E(f^{(0)})$              (best so for E)

repeat
    (See whether the current configuration is a repetition:)
    escape $\leftarrow$ memory_based_reaction $(f^{(t)})$   (see Fig. 2)
    if escape = DO_NOT_ESCAPE then
        $\mu \leftarrow$ best_move   (see Fig. 3)
        $f^{(t+1)} = \mu\ (f^{(t)})$
        $\Lambda(\mu) \leftarrow t$
        ($\mathscr{A}^{(t)}$ and $\mathscr{T}^{(t)}$ are therefore implicitly changed, see (4) and (5))
        (Update time, and best_so_far:)
        $t \leftarrow (t+1)$
        if $E(f^{(t)}) < E_b$ then
            $E_b \leftarrow E(f^{(t)})$
            $f_b \leftarrow f^{(t)}$
    else
        diversify_search   (see Fig. 3)
until $E_b$ is acceptable or maximum no. of iterations reached
```

Fig. 1. *RTS* algorithm: main structure

```
function memory_based_reaction (f)
comment: The function returns ESCAPE when an escape is to be executed, DO_NOT_ESCAPE otherwise.
    Search for configuration f in the memory:
    if $\Pi(f) > -\infty$ then   (if f was visited previously)
        Find the cycle length, update last_time and repetitions:
        $R \leftarrow t - \Pi(f)$   ($R$ = repetition interval)
        $\Pi(f) \leftarrow t$
        $\Phi(f) \leftarrow \Phi(f) + 1$   ($\Phi(f)$ = repetitions of f)
        if $\Phi(f) >$ REP then
            $\mathscr{C} \leftarrow \mathscr{C} \cup f$
            (f is added to the set of often-repeated configurations)
            if $|\mathscr{C}| >$ CHAOS then
                $\mathscr{C} \leftarrow \emptyset$
                return ESCAPE (reaction III)
        if $R < 2\ (L-1)$ then
        (if $R \geq 2\ (L-1)$ the cycle is not avoidable)
            $R_{ave} \leftarrow 0.1 \times R + 0.9 \times R_{ave}$
            $T^{(t+1)} \leftarrow Min\ (T^{(t)} \times$ INCREASE, $L-2)$   (reaction I)
            $t_T \leftarrow t$
    else
        If the configuration is not found, install it and set:
        $\Pi(f) \leftarrow t$
        $\Phi(f) \leftarrow 1$

    if $(t - t_T) > R_{ave}$ then
        $T^{(t+1)} \leftarrow Max\ (T^{(t)} \times$ DECREASE, 1)   (reaction II)
        $t_T \leftarrow t$
    return DO_NOT_ESCAPE
```

Fig. 2. *RTS* algorithm: the function **memory_based_reaction**

... then ... else, repeat ... until, repeat for ... to...), the assignment operator ($X \leftarrow Y$ means that the value of variable X is overwritten with the value of Y) and functions that can return values to the calling routine. Compound

```
function best_move
comment: The function returns the move to be applied to the current configuration.
    if $|\mathscr{A}^{(t)}| < 2$ then
        $T^{(t)} \leftarrow (L-2)$
        (so that at least two moves are admissible, reaction IV)
        (when $T^{(t)}$ changes, $\mathscr{A}^{(t)}$ and $\mathscr{T}^{(t)}$ are implicitly changed,
        see (4))
        $t_T \leftarrow t$
    $\mu \leftarrow \arg\min_{\nu \in \mathscr{A}^{(t)}} E(\nu(f^{(t)}))$
    if ASPIRATION then (note: inactive if ASPIRATION is FALSE)
        $\gamma \leftarrow \arg\min_{\nu \in \mathscr{T}^{(t)}} E(\nu(f^{(t)}))$
        if $E(\gamma(f^{(t)})) < E_b$ and $E(\gamma(f^{(t)})) < E(\mu(f^{(t)}))$ then $\mu \leftarrow \gamma$
    return $\mu$

function diversify_search
comment: A sequence of random steps, that become tabu as soon as they are applied.
    Clean the memory structures $\Pi$ and $\Phi$
    $S \leftarrow \{Min(1 + R_{ave}/2, |\mathscr{M}|)$ moves randomly sampled out of $\mathscr{M}\}$
    repeat for $\sigma \in S$
        $f^{(t+1)} \leftarrow \sigma(f^{(t)})$
        $\Lambda(\sigma) \leftarrow t$
        ($\mathscr{A}^{(t)}$ and $\mathscr{T}^{(t)}$ are therefore changed, see (4) and (5))
        (Update time and best_so_far:)
        $t \leftarrow (t+1)$
        if $E(f^{(t)}) < E_b$ then
            $E_b \leftarrow E(f^{(t)})$
            $f_b \leftarrow f^{(t)}$
```

Fig. 3. *RTS* algorithm: the function **best_move** and **diversify_search**

statements are indented, function names are in boldface, comments and descriptions are in italics.

Figure 1 describes the main structure of the *Reactive Tabu Search* algorithm: The initialization part is followed by the main loop, that continues to be executed until a satisfactory solution is found or a limiting number of iterations is reached. In the first statement of the loop, the current configuration is compared with the previously visited points stored in the memory by calling the function **memory based_reaction** (Fig. 2), that returns two possible values (DO_NOT_ESCAPE or ESCAPE). In the first case the next move is selected by calling **best_move** (Fig. 3), in the other case the algorithm enters a diversification phase based on a short random walk, see the function **diversify_search** (Fig. 3). Each new configuration on the trajectory with a lower E value is saved with the associated configuration f, because otherwise this point could be lost when the trajectory escapes from a local minimizer. The couple (f_b, E_b) is the sub-optimal solution provided by the algorithm when it terminates.

When RTS is applied to "difficult" tasks (like NP-complete problems [12], for which no polynomial algorithms have been designed) one must settle for a sub-optimal solution within the allotted amount of CPU time. The avoidance of cycles and confinements assures that the available time is spent in an efficient exploration of the search space, and specific termination conditions (apart from the expiration of the given CPU time) are not necessary. Naturally, specific task-dependent termination criteria can be introduced, for example by setting a threshold on the quality of the solution.

The reactive mechanisms of the algorithm modify the discrete dynamical system that defines the trajectory so that *limit cycles* and confinements in limited portions of the search space are discouraged. The reaction is based on the past history of the search and it causes possible changes of $T^{(t)}$ or the activation of a diversifying phase. Short limit cycles are avoided by modifiying $T^{(t)}$ in the appropriate way. In particular, see the function **memory_based_reaction** defined in Fig. 2, the current configuration f is compared with the configurations visited previously and stored in memory. If f is found, its last visit time $\Pi(f)$ and repetition counter $\Phi(f)$ are updated. Then, if its repetitions are greater than the threshold REP, f is included into the set $\mathscr{C}$, and if the size $|\mathscr{C}|$ is greater than the threshold CHAOS, the function returns immediately with the value ESCAPE (CHAOS = REP = 3 for all the presented tests). If the repetition interval R is sufficiently short (if $R < 2(L-1)$), one can discourage cycles by increasing $T^{(t)}$ in the following way: $T^{(t+1)} \leftarrow T^{(t)} \times$ INCREASE. If f is not found, it is stored in memory, the most recent time when it was encountered is set to the current time ($\Pi(f) \leftarrow t$) and its repetition counter is set to one ($\Phi(f) \leftarrow 1$).

If T is not allowed to decrease, there is the danger that its value will remain large after a phase of the search with many repetitions, even in later phases, when a smaller value is sufficient to avoid short cycles. Therefore, the statement labeled *reaction II* in Fig. 2 executes a reduction by the factor DECREASE < 1 if $T^{(t)}$ remained constant for a number of iterations greater than the moving average of repetition intervals R_{ave}. In our tests the reaction parameters are INCREASE = 1.1, DECREASE = 0.9. Although the choice of these parameters can influence the average convergence time, the "strategic" success of the algorithm is very robust with respect to their choice. After the robustness tests were completed, the same parameters have been used with success for problems ranging from combinatorial optimization, minimization of continuous functions [4], and sub-symbolic learning tasks [5].

If $A^{(t)}$ does not contain at least two moves, $T^{(t)}$ is decreased immediately so that at least two moves become admissible, see the statement labeled *reaction IV* in Fig. 3. If the "aspiration" criterion (see [13]) is absent (if ASPIRATION = FALSE) the best move is selected by testing only the admissible moves. Otherwise, a tabu move can be chosen as the current move if and only if it leads to a function value better than the best found during the previous search and better than the values reachable with admissible moves. This relaxation of its tabu status is allowed because one is certain that repetitions are avoided (this follows trivially from the definition of E_b as the "best so far" value). This simple aspiration criterion has been added to the RTS scheme with the explicit purpose of evaluating its effect in our benchmarks.

When the reaction that modifies $T^{(t)}$ (*reaction I and II* in Fig. 2) is not sufficient to guarantee that the trajectory is not confined in a limited portion of the search space, the search dynamics enter a phase of "random walk" (*reaction III* in Fig. 2), that is specified by the statements in the function **diversify_search** of Fig. 3. The number of random steps is proportional to the moving average R_{ave}, the rationale being that more steps are necessary to escape from a region that causes long cycles. Note that the execution time of the random steps is registered ($\Lambda(\sigma) \leftarrow t$), so that they become tabu, see (4) and (5). When the "random walk" phase begins, the memory structure is cleaned but this is not equivalent to a random restart because R_{ave} and $T^{(t)}$ are not changed. In addition, after the "random walk" phase terminates, the prohibition of the most recent random steps is crucial to discourage the dynamical system from returning into the old region.

2.2. *RTS for constrained tasks*

Simple modifications of the basic RTS scheme are needed so that the constraints of the Multiknapsack problem are taken into account. In particular the method used is to *confine* the trajectory in the admissible space. Another possibility can be that of adding a suitable *penalty term* to E, see Sect. 8.2, with the problem of selecting an appropriate balance between the two terms. We did not consider this option because the simple confinement was completely satisfactory.

The modifications are applicable to other constrained problems, provided that the admissible space is *connected* (two points can always be connected by a path where successive points differ by only one bit), and that an admissible point remains admissible if any bit is set to zero (this is used in the diversification phase). These properties are trivially satisfied for the Multiknapsack problem: two points can be connected by a path passing through the zero string $[0, 0, \ldots, 0]$, and taking items out from an admissible knapsack will never violate constraints.

The modifications are illustrated in Fig. 4. The best move applied at each point is such that the state f remains admissible, and the minimal requirement on the list size T is that it leaves at least one move that is admissible and does not violate constraints. Reducing T so that this requirement is met has the effect of "freeing" the least-recently applied tabu moves (see the function **constrained_best_move**).

The diversification moves are setting to zero with probability 0.5 the bits of the current f, therefore emptying the knapsack in a stochastic way. The "best so far" value E_b does not change during these steps (because E decreases) so that the check can be omitted (see the function **constrained_diversify_search**).

2.3. *Space-time complexity of RTS*

The RTS scheme has been designed under the constraints that the overhead (additional CPU time and memory) introduced in the search process by the reactive mechanisms had to be of small number of CPU cycles and bytes, approximately constant for each step in the search process. We now argue that this is indeed the case: in a simplified version of RTS the additional memory required can be reduced to one *bit* per iteration, while the time is reduced to that needed to calculate a memory address from the cur-

```
function constrained_best_move

comment: The best move returned must not violate constraints.
The sets $\bar{\mathscr{A}}^{(t)}$ and $\bar{\mathscr{T}}^{(t)}$ contain the admissible and tabu moves that
do not violate constraints, respectively.
  $\bar{\mathscr{A}}^{(t)} \leftarrow \mathscr{A}^{(t)} \setminus \{\mu \in \mathscr{A}^{(t)}$ that would cause a constraint violation$\}$
  $\bar{\mathscr{T}}^{(t)} \leftarrow \mathscr{T}^{(t)} \setminus \{\mu \in \mathscr{T}^{(t)}$ that would cause a constraint violation$\}$
  if $|\bar{\mathscr{A}}^{(t)}| < 1$ then
    (decrease $T^{(t)}$ so that at least one move is admissible and does
    not violate constraints, reaction IV:)
    $T^{(t)} \leftarrow \min\{T$ such that $|\bar{\mathscr{A}}^{(t)}| \geq 1\}$
    $t_T \leftarrow t$
  $\mu \leftarrow \arg\min_{\nu \in \bar{\mathscr{A}}^{(t)}} E(\nu(f^{(t)}))$
  if ASPIRATION then
    $\gamma \leftarrow \arg\min_{\nu \in \bar{\mathscr{T}}^{(t)}} E(\nu(f^{(t)}))$
    if $E(\gamma(f^{(t)})) < E_b$ and $E(\gamma(f^{(t)})) < E(\mu(f^{(t)}))$ then $\mu \leftarrow \gamma$
  return $\mu$

function constrained_diversify_search

comment: The "on" bits are set to zero with probability 0.5.
  Clean the memory structure $\prod$ and $\Phi$
  repeat for $i = 1$ to $L$ ($L$ is the length of the string)
    if $f_i = 1$ then
      with probability 0.5 set the i-th bit to zero:
      $f_i^{(t+1)} \leftarrow 0$
      $\Lambda(\mu_i) \leftarrow t$   (the move becomes tabu)
      (otherwise, with probability 0.5, leave $f_i$ unchanged)
    else $f_i^{(t+1)} \leftarrow f_i^{(t)}$
    (Update time only, best_so_far does not change in the
    Knapsack problem:)
    $t \leftarrow (t + 1)$
```

Fig. 4. *RTS* algorithm: modifications for constrained problems

rent configuration and to execute a small number of comparisons and updates of variables.

The space and time complexity of the reaction scheme are reduced to their lower limits of some bytes and to a small and approximately constant number of machine cycles per iteration, provided that a compressed version of the configuration is stored and that the *open hashing* mechanism is used for storing and retrieving the values $\Pi(f)$ and $\Phi(f)$. In the *hashing* scheme f and its associated items Π and Φ are stored in a memory location whose address is a function *address* = *hash*(f). The number of possible addresses N_a must be larger than the maximum number of items to store N_i (say $N_a > 2\ N_i$), and the *hash*() function must "scatter" the addresses of different f's so that the probability that two of them obtain the same address is small. The *open hashing* scheme, see [2], solves the "collision" problem by associating to each *address* a list of items: when a colliding address is found, the list is enlarged by one item.

Let us discuss a minor technical point about the *hashing* scheme. The *hashing* function can be applied either to the current binary configuration, or to the current E value (the two implementations are called RTS-C and RTS-E, respectively). If the probability that two different configurations have the same E value is small, RTS-E will produce a trajectory very similar to that of RTS-C, with the bonus that *hashing* applied to E can be faster (but see [31] for fast hasing schemes that can be applied to the configuration itself, and [23] for a detailed study of *hashing* functions suitable for different problems). Obviously, the

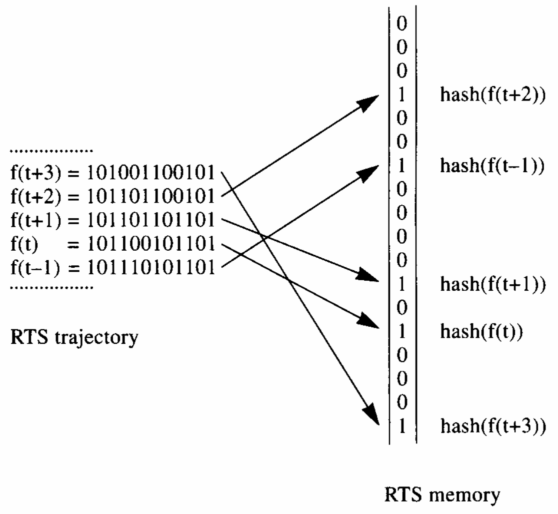


Fig. 5. Use of the RTS memory (one bit per iteration)

stored data can be different from the hashing value: in particular, the collision problems ("false alarms") disappear if the entire configuration is stored in a *bucket* at the hashing address. In general, one has a tradeoff between the probability of false alarms and the number of bits stored.

A more radical approach consists in using the *hashing* scheme with no collision treatment, and in storing a single bit for each location in the hashing structure (0 at the beginning, 1 if a configuration with the given address has been encountered). In this case the RTS algorithm becomes simpler because the part where Π and Φ are used are omitted. The "one-bit" check will always detect if the current configuration (or E value) is equal to one encountered in the previous search, and the probability of a "false alarm" (i.e. of detecting a repetition when the configuration is new) can be made negligible by selecting a sufficiently large *hashing* memory. Let us note that rare "false alarms" have a very limited impact on the search. The drawback of the simplified version is that the robustness of the method for complex E functions decreases, see Sect. 2.1. The memory use in this minimal approach is illustrated in Fig. 5.

The theoretical possibility of using a single bit per iteration has not been used in our runs because accessing integer variables turns out to be faster for a standard workstation architecture, and because the available RAM memory of some MBytes was sufficient for storing the associated Π and Φ values.

3. N-K model: Benchmark tasks and exhaustive results

The N-K model of [20] is a convenient benchmark to analyze functions to be optimized with varying local and global properties. In biological terms one speaks about "tunably rugged fitness landscapes", where "fitness" is the value of the function, about "individuals" given by specific binary strings, and about "genes" given by the single bits. The fitness E corresponding to an N-bit binary string f is obtained as:

$$E = \sum_{i=1}^{N} E_i\left(f_i, f_{j1}, f_{j2}, \ldots f_{jK}\right) \tag{6}$$

where the functions $E_i: \{0,1\}^{K+1} \rightarrow R^1$ depend on the i-th gene f_i and on K other genes that are selected in a random way when the model is defined (the indices $j1, j2, \ldots, jK$ are randomly extracted for each i from the set $\{1,2,\ldots,N\} \backslash \{i\}$). The E_i values are obtained with lookup tables that are filled with randomly generated values in the range [0,1]. The detailed generation is described in Fig. 6, while the fitness computation is described in Fig. 7.

For the portability of our benchmark tasks, let us briefly describe the family of functions (standard in the Unix© environment) that are used in this work for the generation of pseudo-random numbers. The function **srand48**() is used to initialize the sequence with a suitable *seed*, **drand48**() returns non-negative 64-bit floating-point values uniformly distributed over the interval [0.0,1.0), **lrand48**() returns non-negative 32-bit integers uniformly distributed over the interval $[0, 2^{31})$. The routines work by generating a sequence of 48-bit signed integer values, X_i, according to the linear congruential algorithm of [22]:

$$X_{n+1} = (aX_n + c) \bmod m \quad n \geq 0$$

The parameters are $m = 2^{48}$, $a = 5DEECE66D_{16}$ (hexadecimal notation), $c = B_{16}$. **lrand48**() returns the 32 high-order (leftmost) bits of X_i. In **drand48**() X_i is converted to a double precision floating point value and divided by 2^{48}. The initializer function **srand48**(*seed*) sets the high-order 32 bits of X_0 to the 32 bits of its argument. The low-order 16 bits of X_0 are always set to $330E_{16}$.

The integer K tunes the difficulty of the optimization task: when $K = 0$ E is a linear function of its bits: by changing a single bit, only one E_i value changes (no "espistasis"), when $K = N - 1$ the ruggedness of the function is maximal: by changing a single bit, all E_i values are modified. The properties of the "fitness landscape" associated to the N-K model are studied in [32] by using the theory of stationary random processes under the assumption of statistical isotropy. One considers the sequence of fitness values obtained by starting at a random chosen configuration $f^{(0)}$ and moving at each step to a randomly-chosen neighbor. The sequence of values $E^{(t)} \equiv E(f^{(t)})$ can be statistically described as a *first-order autoregressive process*, defined by the equation:

$$E^{(t)} = z\, E^{(t-1)} + R^{(t)} \tag{7}$$

where $R^{(t)}$ is a stationary sequence of uncorrelated random variables.

The autocovariance function of the above time series behaves like a decaying exponential:

$$<E^{(t)}E^{(t+h)}> - <E^{(t)}><E^{(t+h)}> \approx c\, z^h = c \exp(h/\tau) \tag{8}$$

where < > is the average and τ is the effective correlation length: $\tau = -1/\log(z)$. The shorter the correlation length, the shorter is the number of random steps that are sufficient to de-correlate the function values along the trajectory. The correlation length as a function of the parameters N and K is given by:

$$\tau \approx \frac{-1}{\log\left(\dfrac{N-(K+1)}{N}\right)} \tag{9}$$

The approximation fails for very small K values. Note that when K tends to $N-1$, τ tends to zero: a move to a random point in the neighborhood will produce an E value almost completely uncorrelated. For the two N-K cases that we consider (24-10 and 100-15) we obtain: $\tau_{24-10} = 1.63$ and $\tau_{100-15} = 5.73$. The globally optimal results for the "small" N-K tasks can be read in Table 1, under the "Best" column of Repeated Local Minima Search.

```
procedure generate_NK_task (seed)

comment: Generates an instance of the N-K model with a portable
random number generator.
The matrix $l[N][K+1]$ determines the $j_1, j_2, \ldots, j_K$ indices for each
$E_i$, see (6)
The matrix $val[N][2^K+1]$ is the lookup table for the $E_i$ values see (6)
  srand48(seed)   (random number generator startup)
  repeat for $i = 1$ to $N$
    $l[i][1] \leftarrow i$
    repeat for $k = 2$ to $K + 1$
      repeat
        $r \leftarrow 1 + ($lrand48() mod $N)$
      until $r$ is not present in $l[i][1], \ldots, l[i][k-1]$
      $l[i][k] \leftarrow r$
    sort $l[i][1], \ldots, l[i][k+1]$ so that
    $l[i][1] < l[i][2] < \ldots < l[i][K+1]$
  repeat for $i = 1$ to $N$
    repeat for $h = 1$ to $2^{K+1}$
      $val[i][h] \leftarrow$ drand48()$/N$
```

Fig. 6. N-K model: task generation

```
function NK_fitness (f)

comment: Returns the fitness of the N-bit binary string f.
  $tmp \leftarrow 0$
  repeat for $i = 1$ to $N$
    $index \leftarrow 0$   (index will contain the binary number with digits
    $f_i, f_{j1}, f_{j2}, \ldots, f_{jK}$)
    repeat for $k = 1$ to $K + 1$
      if $f_{l[i][k]} = 1$ set $index \leftarrow index | 2^{k-1}$
      ("|" is the bitwise "or" operator)
    $tmp \leftarrow tmp + val[i][index]$
  return tmp
```

Fig. 7. N-K model: fitness computation

4. N-K model: algorithms applied

In this section the Combinatorial Optimization techniques tested on the N-K model are briefly presented. The *Reactive Tabu Search* technique has already been illustrated in Section 2. In detail, the checks are always executed on the E values (RTS-E) and the *open hashing* scheme is used, so that possible effects caused by "collisions" are completely eliminated [2].

```
procedure repeated_local_minima_search

repeat
    Generate a random binary string $f \in \{0, 1\}^N$

    greedy_search ($f$)
    $E_b$ is updated as soon as a local minimum is encountered, and the
    greedy search terminates
until $E_b$ is acceptable or maximum no. of iterations reached
```

Fig. 8. Skeleton for RLMS

4.1. Repeated local minima search (RLMS)

This simple version of local search, see Fig. 8, is based on the repeated generation of random strings that are used as starting points for a pure "greedy" search. At each point the complete neighborhood is evaluated; if the best neighbor reduces E, the move is executed, otherwise the "greedy" search is stopped at a local minimizer. As usual, the best value E_b found during the repeated greedy phases is saved with the corresponding configuration. The RLMS technique has been chosen in this work to obtain a "unit of measure" for the performance of methods based on local search. In fact it is useless to consider more sophisticated techniques if their performance is not clearly superior to RLMS. It is unfortunate that many experimental tests of heuristic algorithms do not compare the obtained performance with such simple technique, so that it is difficult to judge about the relative quality of the different proposals.

4.2. Simulated annealing (SA)

The Simulated Annealing technique of [21] has generated a significant attention as a general-purpose way of solving difficult optimization tasks. Although successful in solving some relevant problems (see [1] for a review of some applications), SA has been criticized recently in [10], where it is demonstrated that the *asymptotic* performance of SA is worse than that of the simple RLMS previously described (called "randomized local search" in [10]). Precisely, the following theorem holds:

Theorem. *For any given instance of a global optimization problem, there exists a constant t_0 such that, if $t>t_0$ the probability that RLMS finds an optimal solution is larger than the probability then SA finds an optimal solution.*

This result, that holds for any annealing schedule, considerably reduces the excitement about the SA technique. While the proof is given in [10], a qualitative explanation can be given by considering the "fitness surface" of a problem. Let h be the minimum height of the "barrier" surrounding the current search point. Now, when the temperature becomes much smaller than h, either the point is in the basin leading to the global optimum, or it will "never" be able to escape (and in this case SA will eventually be defeated by RLMS). The main drawback of SA is related to its Markovian nature (see [1]): the move at each step

```
procedure simulated_annealing

comment: T is the "temperature", that is progressively decreased
by DECR $\in (0, 1)$.

Generate a random binary string $f^{(0)} \in \{0, 1\}^N$
$t \leftarrow 0$
$T \leftarrow T_0$   (initial temperature)
repeat
    repeat for $m_T$ steps   (Markov chain)
        $\mu \leftarrow$ random $\in \mathcal{M}$   (generate a random move)
        if $E(\mu f^{(t)}) < E(f^{(t)})$ then
            $f^{(t+1)} \leftarrow \mu f^{(t)}$   (descending moves always accepted)
        else
            $f^{(t+1)} \leftarrow \mu f^{(t)}$ with probability
            $p = \exp\left(-\frac{E(\mu f^{(t)}) - E(f^{(t)})}{T}\right)$
            (otherwise current $f$ unchanged: $f^{(t+1)} \leftarrow f^{(t)}$)
        $t \leftarrow (t+1)$
    $T \leftarrow T \times$ DECR
until $T < T_f$   (crystallization)
```

Fig. 9. Skeleton for SA

does not depend on the previous moves and there is no method for the search strategy to detect that it is trapped in the basin of attraction of a suboptimal local minimum. RLMS solves the "trapping" problem by brute force, i.e. by restarting the search when a local minimum is encountered, while RTS solve the problem by using a memory-based search where diversification is encouraged as soon as there is evidence of cycles or, in general, of the repetition of previously visited configurations.

The version of the SA algorithm that we used is described in Fig. 9. The parameters are: T_0 (initial temperature), T_f final temperature, m_T (length of the Markov chain at temperature T), and DECR (temperature-reduction factor). For the application to the N-K model, the initial temperature T_0 was chosen so that almost all proposed changes are accepted at the beginning. A total of 1024 points are generated for each task, and the maximum difference ΔE with all points in their neighborhoods is calculated. Then T_0 is initialized with two times this maximum difference. The remaining parameters are $T_f = 0.01$ and $m_T = N$, while DECR has been chosen so that the total number of function evaluations is 960,000 (for the 24-10 task) and 10,000,000 for the 100-15 task.

4.3. Genetic algorithms (GA)

Genetic Algorithms optimize functions by mimicking the natural selection mechanisms [18]. GA operate on a population of strings and use genetic-like operators to produce new individuals from the best individuals of the current population. While the *exploitation* of the fitter individuals is effected with higher selection probabilities, random mutations assure that all points of the search space can be reached (*exploration*).

A wide variety of genetic schemes is being considered for Combinatorial Optimization. The algorithm that we use

```
procedure genetic_algorithm
comment: M is the size of the "population" P, with strings $s_1, \ldots, s_M$ as "members"

Generate the initial population P with random binary strings $s_i \in \{0,1\}^N$
repeat
    $\forall_i fit_i \leftarrow E(s_i)$ (evaluate the fitness for all members)
    Compute the rescaled fitness:
    $fit_{min} = \min_i(fit_i); fit_{max} = \max_i(fit_i); \forall i\, fit_i \leftarrow \frac{fit_i - fit_{min}}{fit_{max} - fit_{min}}$

    Selection of a new population Q of "mating partners" $q_i$:
    repeat for $i = 1$ to $N$
        $q_i \leftarrow$ random extraction out of P with probability
        $p(s_i) = \frac{fit_i}{\sum_{j=1}^{N} fit_j}$

    if ELITE_SIZE > 0 Extract the best ELITE_SIZE individuals from P and install them into $\bar{Q}$

    Reproduction with uniform cross-over to complete the new population $\bar{Q}$:
        Choose randomly (N – ELITE_SIZE)/2 distinct pairs $(q_i, q_j)$ using the N members of Q
        For each pair, with prob. $p_{cross}$ produce two offsprings by mixing the parent genes,
        otherwise copy the parents to the offsprings

    Random mutation:
        Change (complement) each bit of all $\bar{q}_i$, apart from the "elite", with probability $p_{mut}$

    Replacement:
        $P \leftarrow \bar{Q}$ (P is replaced with the newly computed $\bar{Q}$)
until $E_b$ is acceptable or maximum no. of iterations is reached
```

Fig. 10. Skeleton for GA

is a version of "GA with stochastic selection and replacement, uniform crossover and mutation". It has been derived from [15], [28] (that discusses the multi-point crossover methods), [8], [16] (that contains a general discussion about the control parameters), and [27] (that analyses the elitistic strategy).

The structure of the algorithm is described in Fig. 10. Our population contained 16 individuals, a number that was found experimentally to be appropriate for the task, see [3] that studies a parallel implementation of Genetic Algorithms on the Connection Machine©.

When the elitistic strategy is activated (we tried both GA with and without it), we set ELITE_SIZE $= (|P| + 5)/10$. The crossover and mutation probabilities are $p_{cross} = 1.0$ and $p_{mut} = 1/N$.

5. N-K model: results on "small" tasks

A total of eight different tasks (with seed $= 1, 2, \ldots, 8$) have been considered and for each task 32 optimization runs have been executed for each algorithm, after starting from different random initial points. The results are collected in Table 1, where in each line we show the exact global optimum of the task, the number of times (out of 32) that the algorithm found the global optimum, and the largest percent difference with respect to the optimum $(100 \times (sub\text{-}optimal\_value - global\_optimum)/global\_optimum)$. The range is zero, and it is not shown, if all 32 runs terminated with the global optimum. Then we list the average number of function evaluations needed to reach the best value of each run, whatever it was. The maximum number of function evaluations for each run is 960,000. For space reasons we do not report the actual CPU time, that is dependent on the specific computer, language, and compiler. Estimates of the CPU time can easily be derived from the skeletons of the algorithms presented in the previous sections.

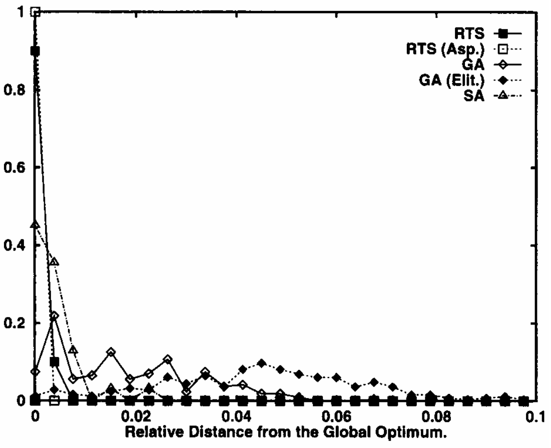


Fig. 11. N-K model: distribution of values found during the different runs for the 24-10 task. Values are normalized differences with respect to the global optimum

It can be observed that RLMS always reaches the global optimum, this is not surprising because approximately 5% of the 2^{24} admissible points are evaluated during the search.

RTS with aspiration exactly solves almost all instances, with the exception of the task with *seed* = 6, that turns out to be the most difficult one (also for RLMS, if one considers the larger average number of steps required). The average number of function evaluations for RTS is almost always considerably reduced with respect to that of RLMS. If the aspiration criterion is absent the average number of steps increases, there are more sub-optimal solutions for problem 6, and one for problem 5, out of the 32 runs.

The performance is not completely satisfactory for SA, that require more evaluations and do not always produce the correct optimum, and much worse for GA, that fails in most of the cases and that shows a large spread of the final values obtained.

Figure 11 illustrates with a coarse 32-bin histogram the distribution of the sub-optimal values found for all instances and all algorithms, apart from RLMS (whose distribution is single spike at the origin). In order to compare the values corresponding to different problem instances, they have been normalized to percent differences with respect to the global optimum before collecting the statistics.

In detail, the plots show the experimental probability distribution function for obtaining final values at different relative distances with respect to the optimum. The

Table 1. N-K model: results for 24-10 task

Seed	Best	(No.)	% Range	Fevals
Repeated Local Minima Search				
1	0.80502790	(32)	–	195759
2	0.78208940	(32)	–	209703
3	0.81098140	(32)	–	186294
4	0.80621170	(32)	–	189391
5	0.79743650	(32)	–	104060
6	0.79207170	(32)	–	317490
7	0.79329850	(32)	–	128104
8	0.79861770	(32)	–	171779
Simulated Annealing				
1	0.80502790	(29)	1.670	596585
2	0.78208940	(31)	0.022	569596
3	0.81098140	(31)	2.494	520038
4	0.80621170	(30)	1.109	597866
5	0.79743650	(31)	0.414	537572
6	0.79207170	(22)	0.678	521948
7	0.79329850	(27)	0.154	625761
8	0.79861770	(24)	0.937	556301
Genetic Algorithm				
1	0.80502790	(7)	4.938	331485
2	0.78208940	(7)	1.963	322004
3	0.81098140	(5)	6.788	390330
4	0.80621170	(7)	5.532	390616
5	0.79743650	(5)	2.410	377320
6	0.79207170	(3)	2.805	359719
7	0.79329850	(5)	2.828	381359
8	0.79861770	(2)	5.515	424069
Genetic Algorithm (with Elitistic Strategy)				
1	0.78987400	(0)	5.817	131996
2	0.78208940	(1)	5.863	174060
3	0.81098140	(1)	10.014	129905
4	0.80621170	(3)	9.593	200838
5	0.79413160	(0)	7.092	196255
6	0.79207170	(2)	7.889	189183
7	0.79329850	(1)	7.280	135071
8	0.79861770	(1)	8.091	201626
Reactive Tabu Search				
1	0.80502794	(32)	–	159944
2	0.78208940	(32)	–	271784
3	0.81098138	(32)	–	80500
4	0.80621169	(32)	–	62308
5	0.79743652	(31)	0.414	201427
6	0.79207169	(23)	0.264	321453
7	0.79329847	(32)	–	178885
8	0.79861770	(32)	–	123124
Reactive Tabu Search (with Aspiration)				
1	0.80502794	(32)	–	99645
2	0.78208941	(32)	–	139473
3	0.81098137	(32)	–	49959
4	0.80621169	(32)	–	95375
5	0.79743652	(32)	–	156833
6	0.79207169	(27)	0.264	313727
7	0.79329847	(32)	–	111247
8	0.79861770	(32)	–	120732

"spread" is large for GA, and even larger if the elitistic version is used, while both RTS and RTS with aspiration concentrate almost all probability within 0.5% percent of the optimum. SA probability is negligible after about 1% of the optimum.

Table 2. N-K model: results for 100-15 task. A asterisk identifies the best value obtained

Seed	Best	% Range	Fevals
Repeated Local Minima Search			
1	0.76008930	3.265	2764731
2	0.75343020	2.091	3027133
3	0.76021670	3.013	2748891
4	0.75610320	2.600	2748945
Simulated Annealing			
1	0.76507090	2.089	8096559
2	0.76706800	2.632	7866815
3	0.77519750	3.422	8048150
4	0.78355840 *	4.234	8270039
Genetic Algorithm			
1	0.67781460	5.993	2140343
2	0.65979400	3.243	1934909
3	0.66758630	4.682	2300765
4	0.66011640	3.102	2132896
Genetic Algorithm (with Elitistic Strategy)			
1	0.74665040	6.820	1333181
2	0.73404180	5.821	873087
3	0.73455880	6.193	964984
4	0.73825480	7.319	1027518
Reactive Tabu Search			
1	0.76962269 *	2.304	1811279
2	0.76870986 *	2.102	1860857
3	0.77596915 *	2.858	2155485
4	0.77433969	3.015	2153135
Reactive Tabu Search (with Aspiration)			
1	0.76742340	2.007	1928867
2	0.76658336	2.078	2017479
3	0.77408461	2.754	2068859
4	0.77227939	2.389	2106795

6. N-K model: results on "large" tasks

The best values found by the different algorithms on the 100-15 N-K model are listed in Table 2, after collecting the results of 32 runs for each seed, with a maximum number of 10 million function evaluations each. Note that the exact optimum for the tasks cannot be calculated with the available CPU time. Given this, an experimentally *estimated optimum* is obtained as the best values found for a given task considering all algorithms and all runs. This value is identified with an asterisk in Table 2. Note that the percent range is relative to the best value found for a given algorithm. This value indicates the spread of E_b at the end of the different runs.

The results are substantially different with respect to those of the smaller problems. RLMS performance is worse: it does not find any of the estimated optima (while SA finds one and RTS three) and the average results are approximately between 1% and 5% relative distance from the estimated optimum.

The elitistic version of GA is now better that the non-elitistic version (while the opposite was true for the smaller tasks), but the best values found during the various runs

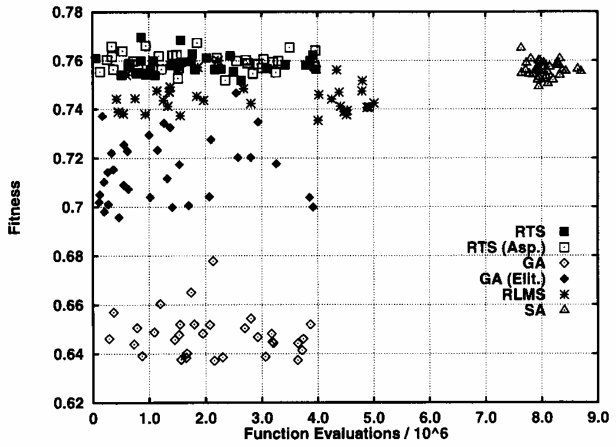


Fig. 12. N-K model: results for "large" task ($N=100$, $K=15$), seed = 1

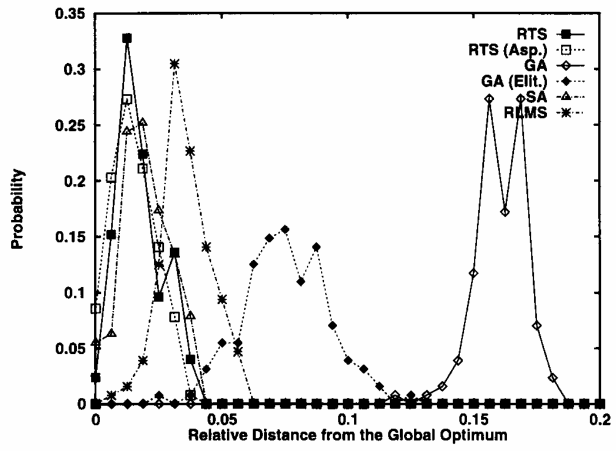


Fig. 13. *N-K* model: distribution of values found during the different runs for the 100-15 task. Values are normalized differences with respect to the estimated optimum

of the elitistic GA fall between 5% and 10% of the best ever found, while those of GA are between 12% and 18%.

While Table 2 lists only the maximum relative range, Fig. 12 shows a scatter plot of the E_b values versus the number of function evaluations when they were first found during the various runs, for all runs on the task with seed = 1. It can be noted that many of the 64 RTS runs (for RTS with and without aspiration) determine their best values during the first millions of function evaluations. On the contrary, most SA runs fix their E_b value in the last "crystallization" phase. Faster cooling schedules tended to produce final results similar to those of the RLMS technique.

The experimental probability of falling within a given relative distance from the estimated optimum at the end of a run is derived by collecting the normalized values for all runs and all tasks ((E_b-*estimated_optimum*)/*estimated_optimum*). For example, the data of Fig. 12 are projected onto the fitness axis and normalized with respect to $E=0.77380420$, the best value for seed = 1, and the same procedure is repeated for the other seeds. Finally the experimental probability has been derived by counting the number of points in a series of bins. The experimental global probability distribution is shown in Fig. 13. Note that the normalization is now with respect to the best value over all algorithms and not the best of the individual ones, that was reported in Table 2.

The two versions of RTS-E, with or without the aspiration criterion, do not show statistically significant differences. The relative performance is such that most of the best values for the runs fall within 2–3% relative distance from the estimated optimum.

7. Multiknapsack: benchmark tasks and exhaustive results

In the Multiknapsack problem [29] the goal is to fill a "knapsack" with a subset of N items with associated utilities c_i and loads a_{ki}, such that their total utility

$$\mathcal{U} = \sum_{i=1}^{N} c_i f_i \tag{10}$$

is maximized, subject to a set of M load constraints:

$$\sum_{i=1}^{N} a_{ki} f_i \le b_k \quad \text{for } k = 1, \ldots, M \tag{11}$$

The binary decision variables f_i specify whether or not item i is included into the knapsack.

Our benchmark problems are derived from [25], where the parameters a_{ki} and c_i are independent uniformly distributed random numbers, and b_k is equal to $b=N/4$ for all constraints (this choice of the "most difficult case" for b_k implies that about $N/2$ items will be contained in the optimal solution, so that the exact solution becomes inaccessible for large N). Unfortunately, floating point parameters cause cancellation errors when the constraints are calculated (by updating values along the search trajectory), with the danger that small errors can activate a constraint that would otherwise be inactive. In order to assure the complete portability and reproducibility of our benchmarks we decided to use 32-bit integer calculations. All values (including b) are therefore multiplied by 2^{20} and transformed into integers. Let us note that this multiplication does not have any effect on the performance of the different algorithms.

The difficulty of the task is tuned by a scale parameter for the c_i, that regulates the spread of their values. In de-

```
function generate_knapsack_task (seed)

comment: Generates an instance of knapsack with a portable random number generator.
    srand48 (seed)  (generator startup)
    repeat for $k$ = 1 to $M$
        $b_k \leftarrow$ floor ($2^{20} \times N \times 0.25$)
    repeat for $i$ = 1 to $N$
        $c_i \leftarrow$ floor ($2^{20} \times (0.5 - scale/2.0 + scale \times$ drand48()))
        repeat for $k$ = 1 to $M$
            $a_{ki} \leftarrow$ floor ($2^{20} \times$ drand48())
```

Fig. 14. Multiknapsack: task generation

Table 3. Multiknapsack: global optima for the benchmark suite

scale = 1.0				*scale = 0.1*				*scale = 0.0*			
(seed)	opt.	(seed)	opt.	(seed)	opt.	(seed)	opt.	(seed)	opt.	(seed)	opt.
(0)	8096841	(5000)	9559743	(0)	6894577	(5000)	6996238	(0)	6815744	(5000)	6815744
(100)	9935710	(5100)	8871375	(100)	7105187	(5100)	6997877	(100)	6815744	(5100)	6815744
(200)	9532213	(5200)	10004250	(200)	7087385	(5200)	7571251	(200)	6815744	(5200)	7340032
(300)	9995299	(5300)	8208537	(300)	7133694	(5300)	6876110	(300)	6815744	(5300)	6815744
(400)	9172232	(5400)	9803660	(400)	7888781	(5400)	7408195	(400)	7864320	(5400)	7340032
(500)	10097437	(5500)	9929182	(500)	7143907	(5500)	7077626	(500)	6815744	(5500)	6815744
(600)	8987822	(5600)	7687492	(600)	7312791	(5600)	7161770	(600)	7340032	(5600)	7340032
(700)	8791962	(5700)	9964390	(700)	6997131	(5700)	7399480	(700)	6815744	(5700)	7340032
(800)	10190037	(5800)	10312523	(800)	7625026	(5800)	7614129	(800)	7340032	(5800)	7340032
(900)	8987021	(5900)	10789096	(900)	7016708	(5900)	7552482	(900)	6815744	(5900)	7340032
(1000)	9006259	(6000)	10813020	(1000)	7020362	(6000)	7215467	(1000)	6815744	(6000)	6815744
(1100)	9982896	(6100)	9602396	(1100)	7447176	(6100)	6999926	(1100)	7340032	(6100)	6815744
(1200)	10421361	(6200)	9515261	(1200)	7173511	(6200)	7029976	(1200)	6815744	(6200)	6815744
(1300)	9707497	(6300)	10523479	(1300)	7054636	(6300)	7519213	(1300)	6815744	(6300)	7340032
(1400)	8533579	(6400)	10107718	(1400)	7175034	(6400)	7504045	(1400)	7340032	(6400)	7340032
(1500)	10602380	(6500)	10566487	(1500)	7026071	(6500)	7112556	(1500)	6815744	(6500)	6815744
(1600)	9206076	(6600)	10047873	(1600)	7054773	(6600)	7472601	(1600)	6815744	(6600)	7340032
(1700)	9640036	(6700)	9754974	(1700)	7050308	(6700)	7372999	(1700)	6815744	(6700)	7340043
(1800)	10744791	(6800)	8770738	(1800)	7560656	(6800)	7445782	(1800)	7340032	(6800)	7340032
(1900)	10022224	(6900)	9703796	(1900)	7399727	(6900)	7539346	(1900)	7340032	(6900)	7340032
(2000)	10391629	(7000)	9771832	(2000)	7516386	(7000)	7043289	(2000)	7340032	(7000)	6815744
(2100)	9493061	(7100)	9853233	(2100)	7046412	(7100)	7094255	(2100)	6815744	(7100)	6815744
(2200)	10229116	(7200)	10160495	(2200)	7096540	(7200)	7536984	(2200)	6815744	(7200)	7340032
(2300)	8815764	(7300)	10394273	(2300)	6982108	(7300)	7173593	(2300)	6815744	(7300)	6815744
(2400)	9746691	(7400)	8992537	(2400)	7072038	(7400)	7478725	(2400)	6815744	(7400)	7340032
(2500)	11281082	(7500)	9984298	(2500)	8018266	(7500)	7095254	(2500)	7864320	(7500)	6815744
(2600)	9992374	(7600)	8327137	(2600)	7105019	(7600)	7337258	(2600)	6815744	(7600)	7340032
(2700)	8768444	(7700)	10343871	(2700)	6933077	(7700)	7168552	(2700)	6815744	(7700)	6815744
(2800)	10158953	(7800)	9953541	(2800)	7122071	(7800)	7335459	(2800)	6815744	(7800)	7340032
(2900)	10684987	(7900)	8330769	(2900)	7540243	(7900)	6955701	(2900)	7340032	(7900)	6815744
(3000)	10611792	(8000)	10739713	(3000)	7474446	(8000)	7595414	(3000)	7340032	(8000)	7340032
(3100)	10065181	(8100)	9618150	(3100)	7344148	(8100)	7487662	(3100)	7340032	(8100)	7340032
(3200)	11105464	(8200)	10279164	(3200)	7995728	(8200)	7540681	(3200)	7864320	(8200)	7340032
(3300)	9374663	(8300)	9189848	(3300)	7476762	(8300)	7019622	(3300)	7340032	(8300)	6815744
(3400)	9261063	(8400)	10492361	(3400)	7015022	(8400)	7579702	(3400)	6815744	(8400)	7340032
(3500)	10545871	(8500)	9925327	(3500)	7551012	(8500)	7489176	(3500)	7340032	(8500)	7340032
(3600)	10429416	(8600)	10652557	(3600)	7425815	(8600)	7513297	(3600)	7340032	(8600)	7340032
(3700)	11165473	(8700)	9989923	(3700)	7560291	(8700)	7411294	(3700)	7340032	(8700)	7340032
(3800)	10334211	(8800)	10201305	(3800)	7579402	(8800)	7541800	(3800)	7340032	(8800)	7340032
(3900)	9657424	(8900)	10532368	(3900)	7007524	(8900)	7185724	(3900)	6815744	(8900)	6815744
(4000)	10302427	(9000)	9415567	(4000)	7590054	(9000)	7297599	(4000)	7340032	(9000)	7340032
(4100)	9724600	(9100)	9255847	(4100)	7790767	(9100)	7448462	(4100)	7864320	(9100)	7340032
(4200)	9527637	(9200)	10101531	(4200)	7457310	(9200)	7094236	(4200)	7340032	(9200)	6815744
(4300)	10415813	(9300)	9695850	(4300)	7502640	(9300)	7562339	(4300)	7340032	(9300)	7340032
(4400)	9675861	(9400)	9445948	(4400)	7375677	(9400)	7321332	(4400)	7340032	(9400)	7340032
(4500)	9920987	(9500)	9848810	(4500)	7083528	(9500)	7063763	(4500)	6815744	(9500)	6815744
(4600)	10497199	(9600)	9182444	(4600)	7470194	(9600)	7407745	(4600)	7340032	(9600)	7340032
(4700)	10686611	(9700)	10072803	(4700)	7420587	(9700)	7006392	(4700)	7340032	(9700)	6815744
(4800)	9193480	(9800)	10412722	(4800)	7386477	(9800)	7138711	(4800)	7340032	(9800)	6815744
(4900)	9134199	(9900)	11783840	(4900)	6982462	(9900)	7732176	(4900)	6815744	(9900)	7340032

tail, the parameters for a task identified by a given *seed* are generated in the way presented in Fig. 14.

The first three sets of "small" benchmark tasks ($N=M=30$, the largest values that permit a conveniently fast exact solution) correspond to three different values of the scale parameter (scale = 1.0, 0.1, 0.0) and to different seeds for the random number generators (100 seeds for each set). The 300 tasks were solved exactly by a straightforward implicit enumeration technique (the function value is calculated by updating the current value when a choice for a single bit is executed, a subtree is cut as soon as the constraints are violated) and in Table 3 we list the optimal $\mathcal{U}$ values for the different scales and seeds. For brevity, we will talk about problem sets Knapsack-1.0, Knapsack-0.1, Knapsack-0.0, where the number refers to the *scale* value.

The difficulty of Multiknapsack tasks is greatly affected by the correlation between utilities and loads [29], [11].

Therefore a second series on *strongly correlated* problems is presented in Sect. 11. Only the a_{ki} parameters are generated in a random way. On the contrary, each utility c_i is equal to the average load of the i-th item. The task-generation algorithm is obtained by canceling the c_i-setting line in Fig. 14, and by generating the utilities at the end, as follows:

$$c_i \leftarrow \textbf{floor}\left(\frac{1}{M}\sum_{k=1}^{M} a_{ki}\right) \tag{12}$$

8. Multiknapsack: algorithms applied

In our tests we consider the Multiknapsack task only as an example of a "fitness surface" of applicative interest but we decided not to adopt problem-specific heuristics (see for example [29], or [30] for the use of some complex moves that can be applied with the REM tabu search). The only information available to the various algorithms is the E value, and the differences between the current and the maximum admissible loads $\left(\sum_{i=1}^{N} a_{ki} f_i - b_k\right)$.

The previously cited algorithms need to be modified to take care of the constraints of the Multiknapsack problem. While SA and NN adopt a "penalty" term technique with heuristics suggested in [25], so that the points in search trajectory can violate constraints, RTS, RLMS and GA are modified so that either the search points are confined in the admissible portion of the search space, or they are immediately "corrected" to eliminate constraint violations. This second approach, although effective, is based on the assumption that the admissible portion is *connected* (all points can be visited by starting from an arbitrary admissible point and moving with the allowed basic moves), and that the constraints can always be satisfied by setting to zero some bits of the string.

RTS is initialized with the configuration corresponding to no items in the knapsack. Clearly, in this way the constraints are not violated, although some iterations are spent before reaching the border of the admissible region.

8.1. RLMS with stochastic projection

Each initial point for the greedy search is generated randomly with a uniform probability. If it violates constraints, the "on" bits are randomly selected and set to zero until an admissible point is reached, see the routine described in Fig. 15. Then the greedy search is executed until all moves producing better E values violate the constraints.

The "correction" operation is called *stochastic projection* because its effect is analogous to a projection of a point in R^N onto the nearest point in the admissible region (where all inequality constraints are satisfied). For the Multiknapsack tasks ($N=30$) the probability of generating points outside the admissible region was about 0.89, and only two steps of greedy search were executed before the "border" was encountered, in the average. This algorithm is therefore close to a random search on the border of the admissible region: the greedy search is used only as a sort of fine tuning.

```
procedure stochastic_projector (f)
comment: Maps a constraint-violating f into an admissible one
    repeat
        choose randomly a bit $f_i = 1$ of $f$
        $f_i \leftarrow 0$
    until f is admissible
```

Fig. 15. Multiknapsack: stochastic projector

8.2. SA with time-dependent penalty

In order to apply the SA algorithm to the Multiknapsack task, where inequality constraints are present, the "penalty function" method is used: the problem becomes unconstrained but a suitable penalty is added to favor admissible points. Following the suggestion of [25], it is useful to consider a time-dependent penalty function that is proportional to the degree of violation, so that the quantity to be minimized becomes:

$$E = -\sum_{i=1}^{N} c_i f_i + \alpha(T)\sum_{k=1}^{M}\Theta\left(\sum_{i=1}^{N} a_{ki} f_i - b_k\right). \left(\sum_{i=1}^{N} a_{ki} f_i - b_k\right) \tag{13}$$

where $\Theta(x)=1$ if $x>0$, 0 otherwise, and $\alpha(T)=\text{ALPHA}/T$, with $\text{ALPHA}=0.1$. The parameters of SA are $T_0=15$, $T_f=0.01$, $\text{DECR}=0.995$. In the last phase of the SA algorithm the size of penalty term when a constraint is violated is so large that the search is in practice confined in the admissible portion of the space.

8.3. GA with stochastic projection (GA-P)

The operator described in Fig. 15 is used to correct the "offspring" obtained with the reproductive scheme described in Fig. 10: if a new individual in the new population $\bar{Q}$ is not admissible it is immediately projected. The analogy is that of a "genetic surgery" applied to defective individuals.

8.4. Neural networks

The use of "neural networks" to solve combinatorial optimization task was pioneered by [19], where it is shown that certain highly-interconnected networks of non-linear analogue units can be effective in solving CO problems like the Traveling Salesman Problem. Unfortunately, although the interest of the method for analog VLSI implementations remains, its practical applicability for large scale problems is still the subject of a continuing debate. In particular, the "stability" of the TSP algorithm of [19] results to be poor in subsequent studies by [33]. Our interest in testing a "neural" solution for the Multiknapsack problem was generated by [25] (and [17], although for small-size tasks). We now summarize their formulation.

Equation (13) is minimized with the mean field approximation (MFT) proposed in [26]. In this approximation,

```
function neural_network_optimize
comment: Mean Field Theory approximation, see [25]
$\forall i \quad v_i^{(0)} \leftarrow 0.5$
$T \leftarrow T_0$
$\alpha \leftarrow \frac{\text{ALPHA}}{T}$
$t \leftarrow 0$
repeat
    repeat for $i = 1$ to $N$
        Calculate $\partial E/\partial v_i$ using (15)
        Calculate $v_i^{(t+1)}$ using (14)
    $t \leftarrow (t+1)$
    $\Sigma \leftarrow \frac{4}{N}\sum_{i=1}^{N}(v_i - 0.5)^2$   (saturation)
    $\Delta \leftarrow \frac{1}{N}\sum_{i=1}^{N}(\Delta v_i)^2$   where $\Delta v_i = v_i^{(t+1)} - v_i^{(t)}$   (evolution rate)
    if $0.1 < \Sigma < (N-1)/N$ then DECR $\leftarrow 0.985$
    else DECR $\leftarrow 0.95$
    $T \leftarrow T \times$ DECR
    $\alpha \leftarrow \frac{\text{ALPHA}}{T}$
until $\Sigma > 0.999$ and $\Delta < 0.00001$

Transform analogue variables to digital values:
    $\forall i$ set $f_i \leftarrow \Theta(v_i^{(t)} - 0.5)$

Check that f does not violate constraints
if all constraints are satisfied for configuration f then
    return $E \leftarrow \sum_{i=1}^{N} c_i f_i$
else the binarized is not admissible
```

Fig. 16. Skeleton for NN

the binary variables f_i are replaced with "mean field" variables at temperature T: $v_i = \langle f_i \rangle_T$, and the MFT equations are:

$$v_i = \frac{1}{2}\left[1 + \tanh\left(\frac{-\frac{\partial E}{\partial v_i}}{T}\right)\right] \tag{14}$$

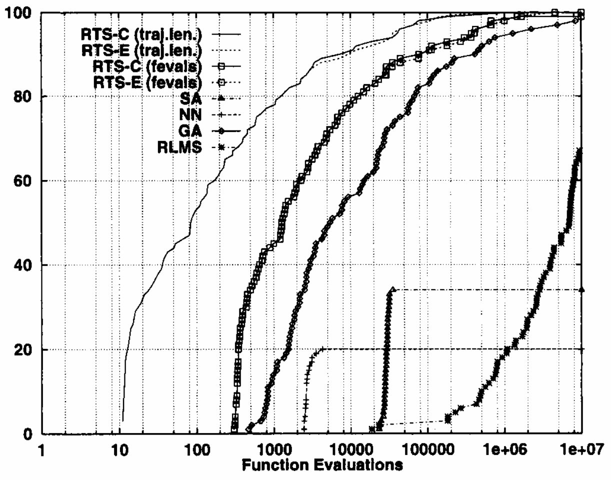
Self-couplings are avoided by replacing $\frac{\vartheta E}{\vartheta v_i}$ with the difference in E computed at $v_i = 1$ and $v_i = 0$, respectively:

$$-\partial E/\partial v_i \leftarrow c_i - \alpha \sum_{k=1}^{M} \left[\Phi\left(\sum_{i=1}^{N} a_{ki} f_i - b_k\right)\Bigg|_{v_i=1} - \Phi\left(\sum_{i=1}^{N} a_{ki} f_i - b_k\right)\Bigg|_{v_i=0}\right] \tag{15}$$

where $\Phi(x) \equiv x \cdot \Theta(x)$. Equations (14) and (15) are solved iteratively by annealing in T. The constraint α is proportional to $1/T$, so that the confinement induced by the penalty term is stronger at the end of the search. The complete algorithm is illustrated in Fig. 16. The parameters of [25] are: $T_0 = 10$, ALPHA $= 0.1$.

9. Multiknapsack: results on "small" tasks

We were interested in measuring how much information about the function to be optimized is used during the search



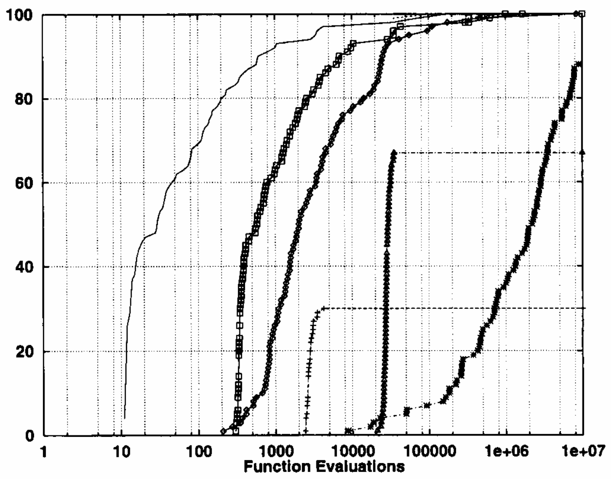


Fig. 17. Knapsack-1.0, N=30: fraction of problems (out of 100) reaching the optimal value (*top*) and 99% of the optimal value (*bottom*)

before finding the global optimum (found by implicit enumeration), or a good approximation of it (i.e. 99% of this value). In other words, we wanted an answer to the following question: what is the (experimental) probability that algorithm X solves an instance of the given problem after a given number of function evaluations? In order to abstract from the properties of individual instances, we report the collective results after running 100 different instances of Knapsack-1.0, Knapsack-0.1, and Knapsack-0.0, described in Sect. 7, for a maximum of 10^7 function evaluations.

In detail, Figs. 17–19 show the total number of instances solved as a function of the number of function evaluations.

For each figure, the top plots are related to the number of instances solved optimally, the bottom ones to the number of instances solved approximately (i.e., by reaching 99% of the optimal value for scales 1.0 and 0.2, and by reaching the second best value for scale 0.0, when the values are "quantized" with a fixed interval between the dif-

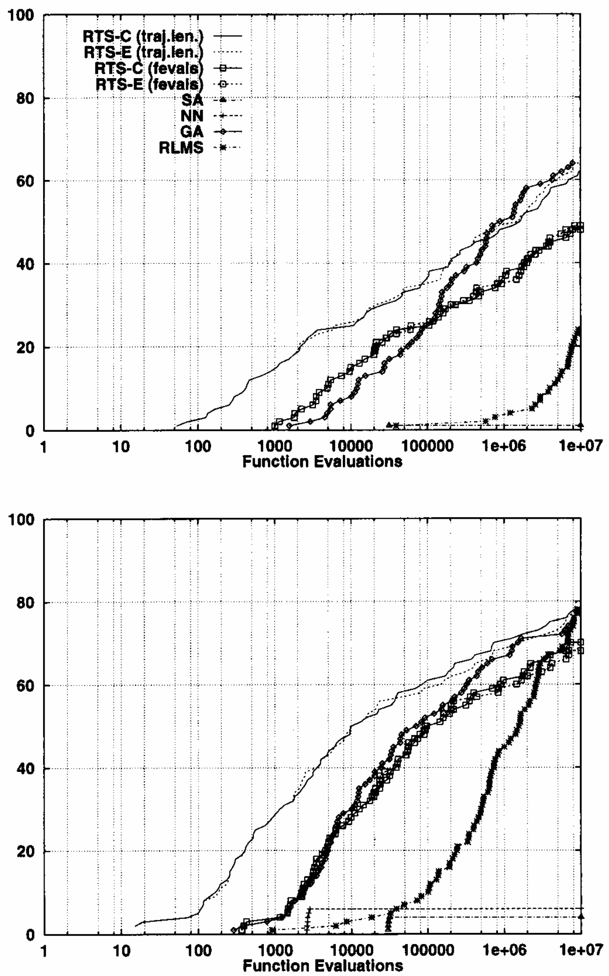


Fig. 18. Knapsack-0.1, N=30: fraction of problems (out of 100) reaching the optimal value (*top*) and 99% of the optimal value (*bottom*)

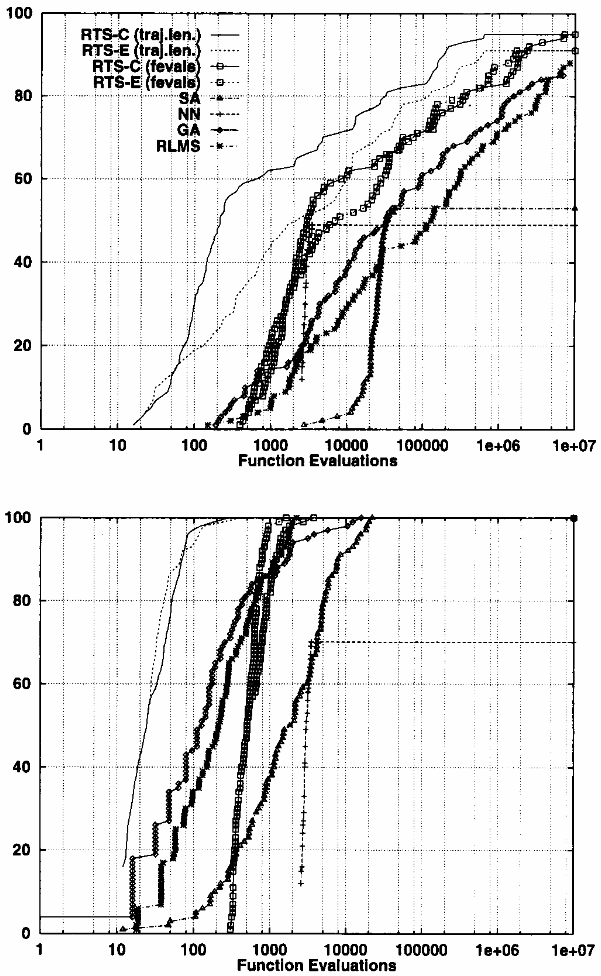


Fig. 19. Knapsack-0.0, N=30: fraction of problems (out of 100) reaching the optimal value (*top*) and the second best (*bottom*)

ferent values). RLMS results are given as a lower performance bound for a local search with the given neighborhood.

Let us briefly comment on the performance for finding the globally optimal solution.

SA performance varies greatly with the problem. At the end of the annealing runs, SA exactly solves 34 instances of Knapsack-1.0, only one instance of Knapsack-0.1, 53 of Knapsack-0.0. After considering the results and the small number of function evaluations, it appeared clearly that the annealing schedule suggested in [25] is too fast to reach acceptable results, especially for the hardest problem given by Knapsack-0.1. We therefore decided to repeat the tests with slower schedules. When DECR was changed from 0.995 to 0.999, the numbers of correctly solved instances rose to 77, 7, 70 (for scale = 1.0, 0.1, 0.0, respectively). With DECR = 0.9998 the number of correctly solved instances were 93, 9, 87, for the three scales. Nonetheless, at a given number of functions evaluated, the number of solved instances remains substantially lower with respect to the number solved by RTS.

GA-P exactly solves 98 of Knapsack-1.0 instances, 64 of Knapsack-0.1, and 85 of Knapsack-0.0. For a given threshold on the number of solved instances, the number of function evaluations is substantially larger than that of RTS of Knapsack-1.0 and Knapsack-0.0, while, for Knapsack-0.1, GA-P is more effective than RTS for large number of function evaluations, although RTS wins at low numbers of function evaluations.

Neural Networks tend to produce binarized final configurations that violate constraints, so that only 20, zero, and 49 instances are solved for Knapsack-1.0, Knapsack-0.1, and Knapsack-0.0, respectively. When a given instance is solved, the solution is very efficient, but unfortunately, in the absence of modifications capable of producing a larger effectiveness, the NN method appears to be of limited practical interest.

RTS solves all instances of Knapsack-1.0, 49 instances of Knapsack-0.1, and 95 of Knapsack-0.0. It is interesting to note that RTS-C (checks on the configurations) and RTS-E (checks on E values) produce almost undistinguishable performances for all tasks but the Knapsack-0.0. In this case the E values are quantized so that a very limited set of values is reached during the search: a wide number of different configurations have the same E values, so that by checking E, the tabu list size will rapidly pass the minimum level that is required to diversify the search. Nonetheless, RTS-E shows a remarkable degree of robustness even in this extreme situation.

The relative performance of the different schemes for reaching suboptimal values is qualitatively similar to that for reaching the optimal ones. Let us note that the difficulty of the task decreases rapidly: already at the 99% level (or to find the second best value for Knapsack-0.0) the performance of RLMS is comparable so that of more sophisticated algorithms (SA and RTS), especially for Knapsack-0.1 (although GA-P and RTS are superior at low numbers of function evaluations) and for Knapsack-0.0.

To avoid cluttering the plots we show the number of solved instances as a function of the trajectory length (or of the number of *neighborhood* evaluations) only for RTS-E and RTS-C. It is interesting to note that the number of function evaluations per *neighborhood* evaluation is decreasing as time progresses (as can be observed from the x-logarithmic plot) because the trajectory is passing near the "border" of the admissible space so that many neighbors violate constraints and do not need to be evaluated. Finally, let us note that, if the *neighborhood* is evaluated in parallel, the real-time is approximately proportional to the trajectory length. The trajectory-length plots for RTS-E and RTS-C are clearly distinguishable only for Knapsack-0.0, see the previous explanation.

10. Multiknapsack: results on "large" tasks

A total of eight Knapsack-1.0 tasks with $N=M=500$ were tested, for seeds equal to 1111, 2222, ..., 8888. Because of the limits on the available CPU time, only one run was tried for each optimization scheme. Given the problem size, the global optimum is not available, the *estimated optimum* is defined as the best values found for a given task considering all algorithms and all runs. The parameters for the NN and SA techniques have been derived from [25] (apart from the scaling by 2^{20} implied by our scaling of E values, that is executed during the computation). For SA (see Fig. 9) the temperatures are $T_0=15$, so that the candidate moves are accepted with a probability close to one at the beginning, $T_f=0.01$, $m_T=N$. The annealing factor DECR$=0.995$ of the cited paper was chosen to obtain a CPU time similar to that of NN. Given the results obtained with the smaller problems, we tried a series of annealing schedules (DECR$=\{.995, .999, .9998\}$). We did not change m_T because using larger m_T's is practically equivalent to using a slower cooling, given that DECR is very close to one. As it was expected, better results are obtained with slower schedules, see Figure 20, for the task with seed$=1111$.

For the NN scheme (see Fig. 16), we used $T_0=10$, a value derived from a statistical analysis so that v_i remains close to 0.5 at the beginning. The suggested value for the penalty multiplier is ALPHA$=0.1$ (to be increased if the

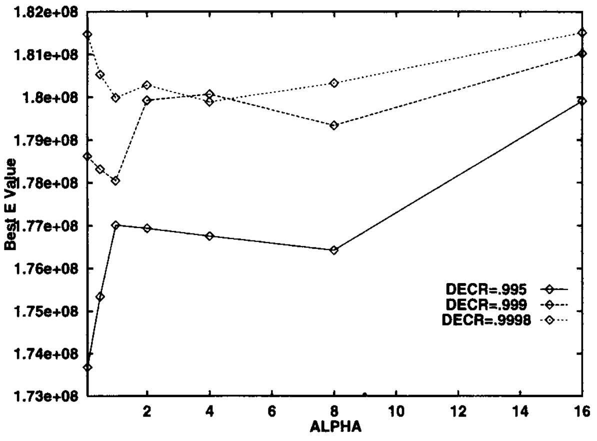


Fig. 20. Knapsack-1.0: best value found in runs with various DECR and Alpha parameters (N=M=500, seed=1111)

obtained solution does not satisfy the contraints). To give NN and SA more chances, for each task we tried the following series of values: ALPHA$=\{0.1, 0.5, 1.0, 2.0, 4.0, 8.0, 16.0\}$. RTS checks E values and uses the aspiration criterion.

The list of the suboptimal values obtained is presented in Table 4. Let us note that the number of function evaluations is fixed by the algorithm in the NN and SA algorithm, while a threshold of one million was put on the maximum length for the RTS trajectory. The RLMS results are not reported because the CPU time required to reach large numbers of function evaluations permitted only a couple of runs (the greedy search trajectories are short and the evaluation of new points is very costly). On the first three instances, RLMS could not reach more than 80% of the *estimated optimum* after 400,000 starts of the greedy search (corresponding to about 250 million function evaluations).

Although a total of 21 parameter combinations were tried for SA, it never reached the best value for the task (in most cases the value is about 0.5–1% less). The suboptimal value reached by SA is always reached by RTS in a lower number of iterations (note that this does not appear in the table, where the number of function evaluations needed by RTS to reach *its* best value is listed, but this appears in Fig. 21).

NN finds the best value in two cases (beating RTS by 0.1–0.2%), with a very small number of "function evaluations", although a direct comparison with SA and RTS is inappropriate: during the search the function is evaluated at points with analog values (and not on admissible strings in $\{0,1\}^N$). It can be that the large value of N involved and the homogeneity of the task (for *scale*$=1.0$) make the "mean field" approximation of statistical mechanics particularly effective. Unfortunately, as it happens with the TSP applications, the method presents a substantial lack of stability: in many cases the "binarized" string obtained at the end of the search is not admissible because it violates the constraints (see Table 4). No particular structure

Table 4. Multiknapsack: suboptimal values for the "large" tasks ($N=M=500$). An asterisk identifies the best value obtained

Seed	SA	(Fevals)	NN	(Fevals)	[Admissible]	RTS	(Fevals)
1111	181516586	(13617506)	181752176	(42000)	[2/7]	183080332*	(232744665)
2222	186924707	(14094661)	187810878	(47000)	[6/7]	188238221*	(236656425)
3333	181541379	(13769646)	182435114	(42000)	[5/7]	182661134*	(225516962)
4444	179067314	(14744236)	180417755*	(52000)	[4/7]	179955821	(132098137)
5555	181429636	(14323357)	181823046	(43000)	[4/7]	182804875*	(109535539)
6666	184692412	(14363896)	184550223	(63000)	[4/7]	185811309*	(232631822)
7777	179154340	(14272739)	180420804	(46500)	[4/7]	181219988*	(239515115)
8888	182275858	(13209735)	182927211*	(42000)	[3/7]	182799696	(196986875)

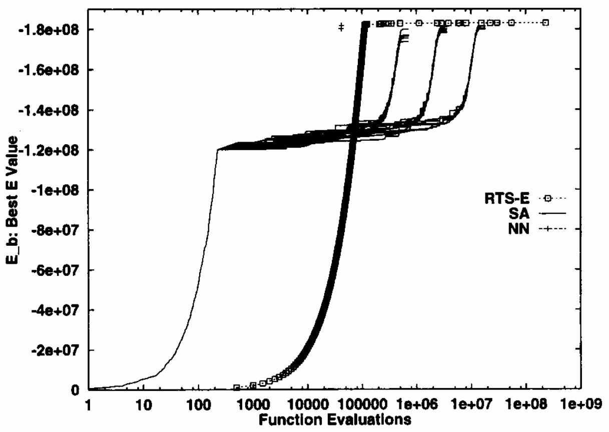


Fig. 21. Knapsack-1.0: evolution for a "large" task ($N=M=500$), seed = 1111

in the distribution of successful cases with respect to the values of ALPHA was detected: unsuccessful cases are present for all values tested. In addition, the spread of the results, when the string is admissible, is around 1%. In spite of its speed, that is comparable to that of RTS, the successful applicability of NN is therefore a matter of chance and more detailed studies are needed before the technique can be suggested for a general-purpose use.

It is interesting to observe the evolution of the different optimization schemes as a function of time. Because the curves for the eight tasks are qualitatively similar (as it was expected because the statistical properties of the "fitness surface" are the same) in Fig. 21 we show the evolution of E_b (the "best so far value") only for the first task, with seed = 1111.

The three "bunches" of curves for SA correspond to the different annealing schedules. Each curve in the bunch (for a different value of ALPHA) shows a first "ascent" phase similar to that of a random walk, a "plateau" region with a very slow progress and the convergence to the sub-optimal minimizer when the system is frozen. It is interesting to observe that the final part of each curve is almost flat: this is consistent with the view that the point is trapped in a basin of attraction (T does not allow an escape). Note that larger final values tend to be associated with slower schedules, as is espected from the theory of SA. The behavior of SA with respect to ALPHA is complex and different for different cooling schedules, see Fig. 20 that reports the best values found for the first task. In the absence of specific prescriptions the user is bound to waste a sizable amount of CPU time to find a suitable value.

The two "+" signs in the top central fort of Fig. 21 correspond to the only two admissible solutions (out of seven) obtained with NN at approximately the same number of steps, with values separated by about 1%. Note that intermediate values do not exist because the tentative solution is binarized only at the end.

RTS is penalized in the first steps (with the complete evaluation of the neighborhood 500 points have to be evaluated for each move at the beginning, a number that is subsequently reduced by the constraints) but shows an aggressively-growing curve that rapidly (at about 100,000 function evaluations) reaches points close to the best obtained. The best value reached by SA ($E \approx 181.7 \times 10^6$ with 42×10^3 function evaluations) is reached after 113,627 function evaluations. An analogous behavior has been observed for the other instances.

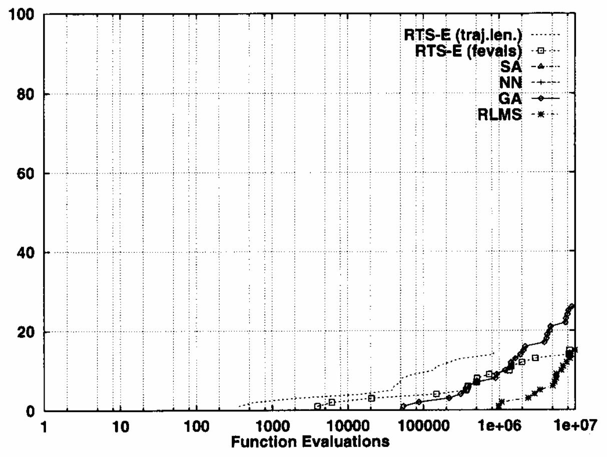
11. Multiknapsack: results on "strongly correlated" tasks

In the previous Sections the difficulty of the Multiknapsack tasks depended on the spread of the utilities [25]. A different parametrization of the "hardness" of a given task can be obtained by changing the amount of correlation between utilities and loads [29], [11]. In particular, *strongly correlated* tasks are obtained by setting each utility c_i equal to the average load of the i-th item (see also Section 7). In this Section, in order to benchmark the algorithms on a wider spectrum of Multiknapsack tasks, the same algorithms are run on a series of *strongly correlated* problems, in the experimental conditions described in Section 9 (small tasks) and Section 10 (large tasks).

11.1. Small tasks. The results on 100 small tasks as a function of the number of function evaluations are illustrated in Fig. 22 (the seeds for the series of 100 small tasks are 0, 100, ... 9900, the list of global optima is available from the authors).

SA performance (with DECR = 0.995, ALPHA = 0.1) is poor: no global optimum is found, only one task is solved within 99%, 76 are solved within 95%. To study the effect of parameter tuning, values of DECR equal to 0.995, 0.999 and 0.9998 were tested. For each of these, ALPHA values equal to 16, 8, 4, 2, 1, 0.5, 0.1 were tried. With the slowest and most successful schedule (DECR = 0.9998) the maximum number of global optima detected was 4 (ALPHA = 16), the maximum number of problems solved within 99% is 22 (ALPHA = 2), all problems are solved within 95%.

GA-P with stochastic projection solves 26 problems to optimality, 69 within 99%, all tasks within 95%. If only the number of function evaluations is considered, GA-P is the most successful algorithm for these tasks.



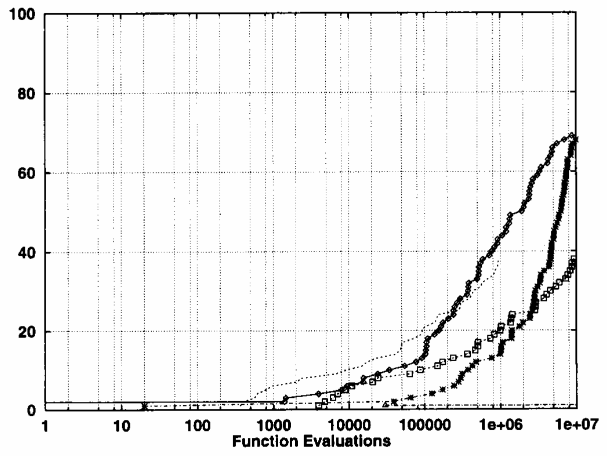


Fig. 22. Multiknapsack, strongly correlated tasks, $N=30$: fraction of problems (out of 100) reaching the optimal value (*top*) and 99% of the optimal value (*bottom*)

RLMS solves 15 problems to optimality, 68 within 99%, all tasks within 95%.

Neural Networks produce illegal configurations (because of constraint violation) in 76 out of 100 cases, only 16 problems are solved within 95%, none is solved within 99%.

RTS shows a good performance in the first phase, especially when the optimal solution is searched, but it is then overtaken by GA-P and by RLMS. At 10^7 function evaluations RTS solves 15 tasks to optimality, 38 within 99% and all tasks within 95%.

The fact that RLMS shows a good performance on these tasks shows that an intense form of diversification is appropriate. The local structure of the problem is probably such that the global optimum is "hidden", among a wealth of suboptimal points. It is not surprising that a neighborhood evaluation based only on the utility changes does not perform particularly well in this case: the utility is in fact proportional to the average load for our set of *strongly correlated* tasks: on average, larger increases of the utility correspond to larger loads. The average length of a local search in RLMS is of 1.9 (after the initial random string is projected to make it admissible). As it happened for the tasks considered in the previous Sections, the local minimum is very close to the "projected" points and RLMS is in practice a random search on the boundary followed by a "fine tuning".

The CPU time of the algorithms divided by the total number of function evaluations gives a scale factor that has to be applied when deciding the practical application of the different approaches. Clearly, the CPU time depends on many factors, like programming language, compiler, operating system, machine, etc. As an example, for our implementations, (C language, native cc compiler, HP 747i 100 MHz) the averages are of 11 μs per function evaluation for RTS, 13 μs for SA, 90 μs for RLMS, 520 μs for GA. Neural Nets are not directly comparable because the trajectory assumes continuous values and is "binarized" only at the end. If the CPU time is considered, RTS is the most successful algorithm in our software implementation.

11.2. Large tasks. The relative ranking of the algorithms is different for the large tasks. The list of the suboptimal values obtained is presented in Table 5. The results of RLMS and GA-P are not listed because the CPU times are too large to be competitive. The CPU times are of 58 μs per function evaluation for RTS, 231 μs for RLMS, 385 μs

Table 5. Multiknapsack: suboptimal values for the large, strongly correlated tasks ($N=M=500$). An asterisk identifies the best value obtained

Seed	SA	(Fevals)	NN	(Fevals)	[Admissible]	RTS	(Fevals)
1111	123678726	(12988698)	124360221*	(45500)	[4/7]	123419276	(174175253)
2222	123893309	(13756833)	124438908*	(49500)	[4/7]	122951776	(45986421)
3333	123989984	(13968899)	124581358*	(43000)	[4/7]	123256860	(97748271)
4444	124016052	(12921989)	124107064*	(46000)	[5/7]	123390394	(144372374)
5555	124189841	(12868875)	124764889*	(52500)	[5/7]	123538344	(16651006)
6666	124302268	(13363570)	124824255*	(50500)	[6/7]	123519648	(135164400)
7777	124080003	(13555633)	124988318*	(55000)	[4/8]	122874614	(160803959)
8888	123749624	(12845482)	124765898*	(57500)	[4/8]	123052141	(165503341)

for SA, 130,000 μs for GA-P. Let us only note that the actual CPU time is approximately proportional to the number of function evaluations and that the constant of proportionality is smaller for algorithms that evaluate the entire neighborhood, like RTS and RLMS. In this case the cost is of $O(M)$ operations per point evaluated, and not of $O(M\ N)$, the cost for evaluating a random point and for checking the constraints. More precise evaluations can be derived from the structure of the different algorithms previously illustrated.

The good performance of both SA and NN on the large tasks could be related to the asymptotic properties of randomly-generated tasks. Some results related to the asymptotic properties of combinatorial optimization problems (especially QAP) are discussed in [7].

While both SA and NN appear to produce strikingly different results in passing from small to large tasks, RTS maintains an acceptable performance and is competitive in both cases if the CPU time is considered. Better performances for the different algorithms could be obtained with problem-specific versions, possibly based on different neighborhood evaluations. We did not consider them in this work because of the limited scope of this paper and because of the assumption introduced in Section 8.

12. Concluding remarks

The Reactive Tabu Search is a memory-based local search method based on simple "reactive" mechanisms that are activated when repetitions of configurations are detected along the search trajectory. The results of the tests presented in this work show that the small overhead required (some CPU cycles and a few bytes per iteration) is paid off by the efficacy and efficiency of the search of tasks of various sizes and difficulties.

RTS demonstrates a satisfactory performance (especially if the CPU time is considered) and a high degree of robustness while maintaining the same algorithmic structure that has been used in [4] and [5], apart from the changes required by the presence of constraints. A similar conclusion was found in [6] for the QAP problem. As a part of the benchmark, various technical changes have been tested (like the use of E values or configurations for the *hashing* function, and the presence or absence of the "aspiration" criterion). The use of E values for *hashing* is almost undistinguishable from the use of configurations (provided that a large set of E values is given). The simple aspiration criterion considered could reduce the average solution time for small tasks (see the 24-10 N-K model), but was not significantly useful for the large tasks.

Neural Nets (based on "mean field" theory) are particularly effective in solving some large-scale Multiknapsack tasks, although the obtained configuration is not always legal and the performance is poor for the small tasks.

Given the growing variety of Genetic Algorithm and Simulated Annealing schemes, our comparison had to be limited to a particular (although widely used) choice of the methods, both for space and knowledge reasons. Although either GA (and GA-P) or SA were successful in some cases, finding the appropriate parameters for specific tasks often is a time-consuming effort. In addition, different parameter combinations tend to be better for small and large tasks. In some cases the simple RLMS technique provided comparable or even better results. We therefore suggest that this straightforward technique should be included when benchmarking advanced optimization algorithms.

Given the variety of problems and algorithms, benchmarking is by its nature a never-ending activity. To ease the use of our benchmark suite in other research environments, the task-generating procedures and the list of the global optima for the exactly solved tasks are available by electronic mail from the authors.

Acknowledgements. We thank one of the anonymous referees for useful remarks and for suggesting the tests in Section 11.

References

1. Aarts E, Korst J (1989) Simulated annealing and Boltzmann machines. John Wiley & Sons, New York
2. Aho AV, Hopcroft JE, Ullman JD (1985) Data structures and algorithms. Addison-Wesley, Reading, MA
3. Battiti R, Tecchiolli G (1992) Parallel biased search for combinatorial optimization: genetic algorithms and TABU. Microprocessors and Microsystems 16:351–367
4. Battiti R, Tecchiolli G (1994) The Reactive Tabu Search. ORSA J Comput (2):126–140
5. Battiti R, Tecchiolli G (1993) Training and nets with the reactive tabu search. Preprint Univ. of Trento. IEEE Trans. Neural Networks (to appear)
6. Battiti R, Tecchiolli G (1994) Simulated Annealing and Tabu Search in the Long Run: a Comparison on QAP Tasks. Comput Math Appl 28(6):1–8
7. Burkard E, Fincke U (1985) Probabilistic asymptotic properties of some combinatorial optimization problems. Discr Appl. Math 12:21–29
8. De Jong KA (1975) An analysis of the behavior of a class of genetic adaptive systems. Dissert Abstr Int 36:(10)5140B
9. Elliot DF, Rao KR (1982) Fast Transforms, Algorithms, Analyses Applications. Academic Press, Orlando, Florida
10. Ferreira AG, Zerovnik J (1993) Bounding the probability of success of stochastic methods for global optimization. Comput Math Appl 25:1–8
11. Fréville A, Plateau G (1993) FPBK 92: an Implicit Enumeration Code for the Solution of the 0-1 Bidimensional Knapsack Problem. Research Report 93-2, Univ. of Valenciennes, J Comput (submitted for publication)
12. Garey MR, Johnson DS (1979) Computers and Intractability: a Guide to the Theory of NP-Completeness, Freeman, San Francisco
13. Glover F (1989) Tabu Search – part I. ORSA J Comput 1:190–206
14. Glover F (1990) Tabu Search – part II. ORSA J Comput 2:4–32
15. Goldberg D (1989) Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley, Reading, Mass
16. Grefenstette JJ (1986) Optimization of control parameters for genetic algorithms. IEEE Trans. Syst. man Cybern 16:122–128
17. Hellstrom BJ, Kanal LN (1992) Knapsack packing networks. IEEE Transact Neural Networks 3:302–307
18. Holland JH (1975) Adaptation in natural and artificial systems, an introductory analysis with applications to biology, control, and artificial intelligence. University of Michigan Press, Ann Arbor
19. Hopfield JJ, Tank DW (1985) "Neural" Computation of Decisions in Optimization Problems. Biol Cybern 52:141–152
20. Kauffman S, Levin S (1987) Towards a general theory of adaptive walks on rugged landscapes. J Theor Biol 128:11–45

21. Kirkpatrick S, Gelatt CD, Vecchi MP (1983) Optimization by simulated annealing. Science 220:671–680
22. Knuth DE (1973) The art of computer programming. Vol 2 – Seminumerical algorithms. Addison Wesley, Reading, Mass
23. Knuth DE (1973) The art of computer programming. Vol 3 – Searching and sorting. Addison Wesley, Reading, Mass
24. Mühlenbein H (1991) The Parallel Genetic Algorithm as Function Optimizer. Parallel Comput 3:619–632
25. Ohlsson M, Peterson C, Söderberg B (1993) Neural networks for optimization problems with inequality constraints: the knapsack problem. Neural Comput 5:331–339
26. Peterson C, Söderberg B (1989) A new method for mapping optimization problems onto neural networks. Int J Neural Syst 1:3
27. Schaffer JD, Eshelman LJ, Offutt D (1991) Spurious correlations and premature convergence in genetic algorithms. In: Foundations of Genetic Algorithms. Morgan Kaufmann, San Mateo, CA
28. Spears WM, De Jong KA (1991) An analysis of multi-point crossover. In: Foundations of Genetic Algorithms. Morgan Kaufmann, San Mateo, CA
29. Martello S, Toth P (1990) Knapsack Problems: Algorithms and Computer Implementations. Wiley-Interscience, Chichester, UK
30. Dammeyer F, Voß S (1993) Dynamic tabu list management using the reverse elimination method. Ann Oper Res 41:123–138
31. Woodruff DL, Zemel E (1993) Hashing vectors for tabu search. Ann Oper Res 41:31–46
32. Weinberger E (1990) Correlated and uncorrelated fitness landscapes and how to tell the difference. Biol Cybern 63:325–336
33. Wilson GV, Pawley GS (1988) On the stability of the traveling salesmann problem algorithm of Hopfield and Tank. Biol Cybern 58:63–70
34. de Werra D, Hertz H (1989) Tabu Search Techniques: A Tutorial and an Application to Neural Networks. OR Spektrum 11:131–141