ORSA Journal on Computing
Vol. 6, No. 2, Spring 1994

0899-1499/94/0602-0126 $01.25

The Reactive Tabu Search

ROBERTO BATTITI / *Dipartimento di Matematica and Istituto Nazionale di Fisica Nucleare, gruppo collegato di Trento, Università di Trento, 38050 Povo (Trento); Italy; Email:battiti@science.umitm.it*

GIAMPIETRO TECCHIOLLI / *Istituto Nazionale di Fisica Nucleare, gruppo collegato di Trento and Istituto per la Ricerca Scientifica e Tecnologica, 38050 Povo (Trento), Italy; Email:tec@irst.it*

(Received: October 1992; final revision received: July 1993; accepted: November 1993)

We propose an algorithm for combinatorial optimization where an explicit check for the repetition of configurations is added to the basic scheme of Tabu search. In our Tabu scheme the appropriate size of the list is learned in an automated way by reacting to the occurrence of cycles. In addition, if the search appears to be repeating an excessive number of solutions excessively often, then the search is diversified by making a number of random moves proportional to a moving average of the cycle length. The reactive scheme is compared to a "strict" Tabu scheme that forbids the repetition of configurations and to schemes with a fixed or randomly varying list size. From the implementation point of view we show that the Hashing or Digital Tree techniques can be used in order to search for repetitions in a time that is approximately constant. We present the results obtained for a series of computational tests on a benchmark function, on the 0-1 Knapsack Problem, and on the Quadratic Assignment Problem.

The tabu search meta-strategy has been shown to be an effective and efficient scheme for combinatorial optimization that combines a hill-climbing search strategy based on a set of elementary moves and a heuristics to avoid the stops at suboptimal points and the occurrence of cycles (see [5, 6, 7]). This goal is obtained by using a finite-size list of forbidden moves (the tabu moves) derived from the recent history of the search. The basic underlying assumption is that the suboptimal points (where the simple hill-climbing component stops) can be better starting points with respect to random restarts, provided that care is taken so that the local maxima (or minima) do not become *attractors* of the dynamics induced by the algorithm and that *limit cycles* do not arise (we borrow the terminology from the theory of dynamical systems[13]).

Some tabu search implementations are based on the fact that cycles are avoided if the repetition of previously visited configurations is prohibited. For example, in the Reverse Elimination Method,[7] the only local movements that are excluded from consideration (i.e., that become tabu) are those that would lead to previously visited solutions. REM is a method to realize what may be called *Strict Tabu* (S-TABU for short).

We argue that S-TABU can converge very slowly for problems where the suboptimal configuration is surrounded by large "basins of attractions," i.e., by large sets of points that converge to it with hill-climbing. This slow convergence is related to the "basin-filling" effect, which is illustrated in Section 2. In addition, the optimal point can become unreachable because of the creation of barriers consisting of the already-visited points. When S-TABU can be used, one can avoid the relatively slow REM technique (that at iteration n requires a computation of order $O(n)$) by using the *hashing* or *digital tree* approaches (which require a constant amount of computing per iteration).

The tabu scheme based on a fixed list size (F-TABU) is not strict and, therefore, the possibility of cycles remains. The proper choice of the size (long to avoid cycles but short in order not to constrain the search too much) is critical to the success of the algorithm, although for many interesting problems the results do not depend too much on its value (see [8] and the contained bibliography). More robust schemes are based on a randomly varying list size,[15] although one must prescribe suitable limits for its variation.

Our *Reactive Tabu* scheme (R-TABU for short) goes further in the direction of robustness by proposing a simple mechanism for adapting the list size to the properties of the optimization problem. The configurations visited during the search and the corresponding iteration numbers are stored in memory so that, after the last movement is chosen, one can check for the repetition of configurations and calculate the interval between two visits. The basic fast "reaction" mechanism increases the list size when configurations are repeated. This is accompanied by a slower reduction mechanism so that the size is reduced in regions of the search space that do not need large sizes.

An additional Long Term Memory diversification mechanism is enforced when there is evidence that the system is in a *complex attractor* of the search space (the analogy is that of chaotic attractors, see Section 1). The LTM "escape" or "diversification" mechanism can be realized with a negligible effort by exploiting the memory structure described.

If a problem requires an excessive memory space to store all configurations, one may resort to compression techniques (the use of hashing for compression in [17] is an

Subject classifications: Programming: integer, heuristic.
Other key words: Tabu search, greedy heuristics, hashing, memory-based optimization.

example). If a total of m configurations are visited, the theoretical minimum on the number of bits needed to distinguish among them is $\log_2 m$ bits per configuration. Reaching the information-theoretic minimum may require complex coding techniques, but with a small increase in memory size even simple compression techniques are effective and well within the typical memory limitations of current workstations.

In the following section, first we motivate and describe the *Reactive Tabu* scheme (Section 1), then we analyze the behavior of the algorithm in the case study of a function of two variables (Section 2) and compare the computation and memory requirements (Section 3). Finally, we apply the *Reactive Tabu* search to the quadratic assignment problem and discuss the results (Section 4).

1. The Reactive Tabu Scheme

Let us begin with an analogy between the evolution of the search process in combinatorial optimization and the theory of dynamical systems (see, for example, [12, 13]). The current configuration traces a path in the configuration space subject to the movements dictated by the search technique. Let us suppose that we are looking for the global minimum (trivially maximizing f corresponds to minimizing $-f$). Local minima are *attractors* of the system dynamics for the steepest descent strategy. They are, in fact, *fixed points* until a scheme is introduced that forces the system to exit from the local minimum and continue the search. *Limit cycles* (or *closed orbits*) are a second possibility, where the trajectory endlessly repeats a sequence of states. Cycles are discouraged by the tabu technique, and they are, in fact, strictly prohibited in the S-TABU version. But there is a third possibility that is very relevant for the case of optimization: the case in which fixed points and limit cycles are absent but the trajectory is *confined* in a limited portion of the search space. In the theory of dynamical systems this phenomenon is described by introducing the concept of *chaotic attractor*. Because, in this paper, the concept of chaotic attractor is used only as an example of a dynamic behavior that could affect the search process, we summarize the main characteristics and refer to [13] for a detailed theoretical analysis. Chaotic attractors are characterized by a "contraction of the areas," so that trajectories starting with different initial conditions will be compressed in a limited area of the configuration space, and by a "sensitive dependence upon the initial conditions," so that different trajectories will diverge.

For an analytical characterization of this sensitive dependence, it is convenient to introduce the concept of *Lyapunov exponent*. Let us consider the function g that maps the point at step n to the point at step $n + 1$, $g^k(x)$ is defined as the map obtained by iterating gk times. Starting from close initial conditions x_0 and $x_0 + \epsilon$, the Lyapunov exponent λ is defined through the relationship:

$$\epsilon e^{n\lambda} \approx \|g^n(x_0 + \epsilon) - g^n(x_0)\|$$

If the exponent λ is greater than 0, the initially close points diverge exponentially and, if the trajectories remain confined in a limited region of the space, one obtains the situation called "deterministic chaos." The motivation for this term is that the trajectory appears to be random, although the system is deterministic. In the above case, although limit cycles are absent, the search trajectory will visit only a limited part of the search space. If this part does not contain the absolute minimum (or the desired configuration), it will never be found.

The motion caused by the tabu search technique is very complex, so that a detailed analytical study of the associated discrete dynamical system is problematic (for example, most of the results in the discrete dynamical systems of "cellular automata," which have a simpler structure with respect to tabu, are based on numerical simulations; see [12] for a brief overview of the subject). Nonetheless, the main suggestion to be derived is that avoiding limit cycles or even avoiding repetitions of configurations is not sufficient for an effective and efficient search technique. The chaotic-like attractors should be discouraged too. In some computational tests we will show evidence of a trapping of the solution trajectory in a suboptimal region of the search spaces (see Section 2). Similar ideas are also present in [8] ("cycle avoidance is not an ultimate goal of the search process... the broader objective is to continue to stimulate the discovery of new high quality solutions"[8]).

The reactive tabu scheme maintains the basic building blocks of tabu search, i.e., the use of a set of temporarily forbidden moves, where the time interval for the prohibition is regulated by the tabu list-size. What we add is a fully automated way of *adapting* the size to the problem and to the current evolution of the search, and an escape strategy for diversifying the search when the first mechanism is not sufficient. Both ideas can be seen as ways to implement the learn-while-searching paradigm that is characteristic of the tabu approach.

The algorithm is summarized in words and described in detail using a pseudo-language derived from the *Pascal* language. To make the description more concise, variable declarations and trivial parts of the code have been omitted or described in words.

For concreteness reasons we will present the algorithm for an application to the Quadratic Assignment Problem (see [14] for one of the first applications of Tabu to the QAP). The space of configurations is given by the possible assignments of N units to N locations ($\phi[loc]$ is the unit that occupies location *loc*). The details about the QAP application will be described in Section 4. Understanding the following Section does not require a detailed knowledge about the QAP problem.

1.1. Basic Tabu Tools

The main tabu structures are common to various tabu implementations for the QAP (see, for example, [15]). An exchange movement is tabu if it places both units to locations that they had occupied within the latest **list_size** iterations. The *aspiration* criterion is satisfied if the function value reached after the move is better than the best previously found. The basic functions for the above operations are illustrated in Figure 1.

BASIC TABU FUNCTIONS (FOR THE QAP PROBLEM)

```
procedure make_tabu(r,s)
comment:
    Record the latest occupation for the two units that are going to be exchanged.
    The array latest_occupation[φ,r] contains the latest occupation time
    for unit φ in location r.
begin
latest_occupation[current.φ[r],r] := current.time
latest_occupation[current.φ[s],s] := current.time
end


function aspiration(r,s)
comment:
    Return the boolean value true if the function value after the movement is
    better than the best value ever found, false otherwise.
begin
if (current.fitness - move_value[r,s])<best_so_far.fitness then
    aspiration := true
else
    aspiration := false
end


function is_tabu(r,s)
comment:
    Return the boolean value true if, after the exchange, both units r and s
    occupy locations that they had already occupied within the latest list_size iterations,
    the boolean value false otherwise.
begin
if  latest_occupation[current.φ[r],s] ≥ current.time - list_size and
    latest_occupation[current.φ[s],r] ≥ current.time - list_size then
    is_tabu := true
else
    is_tabu := false
end
```

Figure 1. Basic tabu functions for the QAP problem.

1.2. Memory Structures

Before explaining our variation of the tabu scheme, let us briefly illustrate the meaning of the variables and memory structures used. An elementary exchange move is indexed by variables **r_chosen** and **s_chosen**, the two locations that will be subjected to the exchange. All visited points in the configuration space are saved in records that contain the placement of units in locations (ϕ), the most recent time when it was encountered (**last_time**), and its multiplicity (**repetitions**). When the number of repetitions for a given point is greater than REP (3 in our runs), the configuration is added to the set of often-repeated ones.

The constants INCREASE and DECREASE determine the amounts by which the **list_size** is increased in the fast reaction mechanism, or decreased in the long-term size reduction process. The variables **moving_average** (a moving average of the detected cycle length) and **steps_since_last_size_change** (the number of iterations executed after the last change of list size) are used for the long-term size reduction; the variable **chaotic** counts the number of often-repeated placements. A diversifying *escape* movement is executed when **chaotic** is greater than CHAOS (a constant equal to 3 in our runs). The status of the search is described by the record **current**; **current.ϕ** is the placement (**current.ϕ[loc]** is the unit contained in location **loc**); **current.f** is the corresponding function value, and **current.time** is the number of steps executed. Each step consists of neighborhood evaluation and move selection. A similar record **best_so_far** stores the best placement found during the search.

The target value **sub_optimum** and the maximum number of iterations **max_iterations** are used for terminating the search.

1.3. Skeleton of R-TABU

Before proceeding with the reactive tabu search, the data records for hashing and tabu are initialized, and a random starting configuration is generated. Then the search routine cycles through the following steps:

i) all possible elementary moves from the current configuration are evaluated;

ii) the latest configuration is searched in the memory structure (with a possible update of the **list_size**, see Section 1.1), and a decision is taken about a diversifying escape move. In the "default" case, i.e., with no escape;

iii-D) (Default) the best admissible move is executed with a possible reduction of the **list_size** if all movements are tabu and none satisfies the aspiration criterion, and the current status and time are updated.

In the other case, i.e., with escape:

iii-E) (Escape) the system enters a phase of random movements whose duration is regulated by a moving average of detected cycles.

The initialization and main loop of R-TABU are illustrated in Figure 2, while the details on the reaction, escape

SKELETON OF REACTIVE TABU

```
procedure initialization
begin
Initialize the data structures for hashing.
Initialize the data structures for tabu: set the latest occupation time
equal to a large negative value:
for unit :=0 to N − 1 do
    for location :=0 to N − 1 do
        latest_occupation[unit,location] := -INFINITY
list_size := 1
chaotic := 0
moving_average :=0
steps_since_last_size_change := 0
current.time := 0

Generate a starting configuration in a random way by setting current.φ[ ] equal to
a random permutation of 0,1,2,...,N-1;
Set current.f equal to the initial function value.

Initialize the best_so_far record containing the best solution ever found:
best_so_far.f := current.f
end


function reactive_tabu_search(max_iterations)
comment: Cycle until the best configuration is found or
    the maximum number of iterations is reached.
begin
while current.time < max_iterations do
    begin
    Find the decrease in function value for all possible elementary moves.
    escape := check_for_repetitions(current.φ)
    if escape = DO_NOT_ESCAPE then
        begin
        choose_best_move
        comment: when the above procedure returns,
            r_chosen and s_chosen contain the two units to be exchanged
        make_tabu(r_chosen,s_chosen)
        Swap the units contained in the locations r_chosen, s_chosen
        Update time, function value and best_so_far.
        end
    else
        escape
    if best_so_far.f ≤ sub_optimum then
        reactive_tabu_search := SUCCESSFUL
        Return from function.
    end

reactive_tabu_search := UNSUCCESSFUL
Return from function.
end
```

Figure 2. Skeleton of reactive tabu algorithm.

and move selection mechanisms will be illustrated in the following Sections.

1.4. The Reaction and Escape Mechanism

When a repetition of a previously encountered configuration occurs, there are two possible reaction mechanisms. The basic "immediate reaction" increases the **list_size** to discourage additional repetitions (**list_size** ← **list_size** × INCREASE). After a number R of immediate reactions, the geometric increase ($\propto$ INCREASE^R) is sufficient to break any limit cycle. In this case a continuous sequence of repetitions rapidly increases the size until the trajectory is forced to explore new regions, but this mechanism may not be sufficient to avoid the "chaotic trapping" of the trajectory in a limited area of the configuration space. To this end a second and slower mechanism counts the number of configurations that are repeated many times (more than REP times). When this number is greater than a threshold CHAOS the **check_for_repetitions** function returns, and diversifying escape movement is enforced.

The reaction is caused by the local properties of the solution trajectory, but, if the **list_size** increases only, it could be excessive in the later phases of the search because it would constrain the search more than necessary. Therefore, a slow process reduces the size if a number of iterations greater than **moving_average** passed from the last size change. The function that checks for the repetitions of function values is shown in Figure 3.

In addition to the immediate increase and slow reduction mechanisms, there is a third point at which the **list_size** is modified. This is the case when the list grows so much that all movements become tabu (and none satisfies the aspiration criterion). When this happens the size is reduced. Because of the geometric decrease, after small number of reductions at least some movements will lose their tabu status.

Our escape strategy is based on the execution of a series of random exchanges. Their number is random and proportional to the **moving_average**, the rationale being that longer average cycles are evidence of a larger basin and, therefore, that more escape steps are likely to be required. To avoid an immediate return into the old region of the search space, all random steps executed are made tabu. The choice of the best move with the reaction and the escape strategies is illustrated in Figure 4.

The different dynamics for the **list_size** and the escape mechanism are illustrated in Figure 5, for an application to a Quadratic Assignment Problem of size $N = 30$.

In Figure 5 (top) we show the evolution of **list_size** and the percent of repetitions of previously visited configurations. Data points are taken every 100 iterations. Note how frequent repetitions provoke a fast increase of **list_size**, while the absence of repetitions provokes a gradual decrease. In Figure 5 (bottom) we show the **list_size** evolution for a larger span of time (up to 50K iterations). Frequent spikes are superimposed to a plateau of about size 8. The bottom curve shows the counter **chaotic**. Each time the value of CHAOS is surpassed, the counter is reset to zero and an escape move is executed. Note that the escapes are automatically triggered by the evolution of the search process. In this case they are executed with a much larger time scale with respect to the frequent spikes of reaction.

1.5. Details on REM, Hashing, and Digital Tree

According to Glover and Laguna,[8] a fundamental element of tabu search is the use of *flexible memory*, which embodies the creation and exploitation of structures for taking advantage of history. In this section we present some competitive structures and algorithms for storing and retrieving the information about the history of the search process in a fast way. Items to store are, for example, the configurations, the corresponding function values, and the iteration number when they were encountered. The various schemes differ in their time-space complexity and in the amount of data stored per iteration.

In the classical Reverse Elimination Scheme (REM) proposed in [7], the visited points are not stored explicitly, but they can be derived by applying a sequence of moves that reverse the moves applied during the search. In fact, one

REACTION AND ESCAPE MECHANISM

```
function check_for_repetitions(φ)
comment: The function takes a current placement φ as argument and returns ESCAPE
    when an escape action is to be executed, DO_NOT_ESCAPE otherwise.
    CYCLE_MAX is a constant equal to 50 in our runs, the other constants and variables
    are described in the text.
begin
steps_since_last_size_change := steps_since_last_size_change+1
Search for the current configuration in the hashing structure.
Set pointer := the location of the record if it is found.
if the configuration is found then
    begin
    Find the cycle length, update last_time and repetitions:
    length := current.time - pointer↑last_time
    pointer↑last_time := current.time
    pointer↑repetitions := pointer↑repetitions +1
    if pointer↑repetitions > REP then
        begin
        Add the current placement to the set of often-repeated ones:
        chaotic := chaotic +1
        if chaotic > CHAOS then
            begin
            Reset counter and execute escape after returning.
            chaotic :=0
            check_for_repetitions := ESCAPE
            Return from function.
            end
        end
    if length < CYCLE_MAX then
        begin
        moving_average := 0.1×length +0.9×moving_average
        list_size := list_size×INCREASE
        steps_since_last_size_change :=0
        end
    end
else
    If the configuration is not found, install it.

if steps_since_last_size_change >moving_average then
    begin
    list_size := Max(list_size× DECREASE,1)
    steps_since_last_size_change :=0
    end

Do not escape in the 'default' case:
check_for_repetitions := DO_NOT_ESCAPE
end
```

Figure 3. Reaction and escape mechanism.

MOVE SELECTION AND ESCAPE FUNCTION

```
procedure choose_best_move
begin
if a move that is not tabu or that satisfies the aspiration criterion is found then
    set (r_chosen,s_chosen) := two units to be exchanged
else
    begin
    If all moves are tabu and none satisfies the aspiration requirement
    find the best of all moves, independently of their tabu status.
    Decrease list_size to decrease the number of tabu moves:
    list_size := list_size×DECREASE
    set (r_chosen,s_chosen) := two units to be exchanged
    end
end


procedure escape
begin
Clean the hashing memory structure.
Generate a random number of steps in the given range. rand returns a random no. in [0,1)
steps := 1 + (1+rand)×moving_average/2
for i=1 to steps do
    begin
    set (r_chosen,s_chosen) := random exchange of two units
    make_tabu(r_chosen,s_chosen)
    update_current_and_best
    Find the decrease in function value for all possible elementary moves.
    end
end
```

Figure 4. Move selection and escape function.

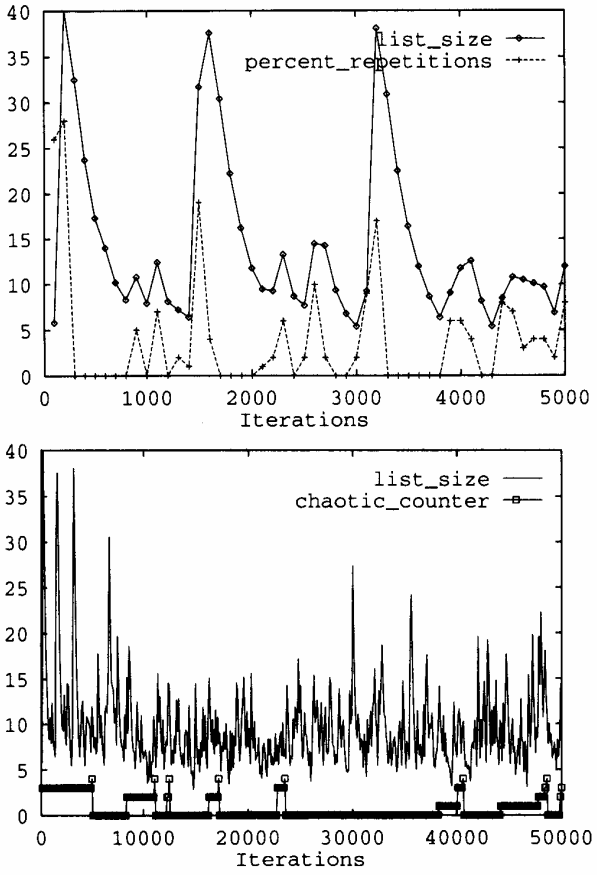

Figure 5. Dynamics of the size of the tabu list (top) and evolution of the repetition counter (bottom).

does not need to find the previous points if one is interested only in knowing which moves will lead from the current configuration to a previously visited one, i.e., the moves that will acquire a tabu status.

REM is based on a *running list* containing all the moves executed from the initial configuration. According to the *sufficiency property* stated in [7], let us assume that all moves m_i are such that a sequence $m_{j1} \circ m_{j2} \circ m_{jk}$ is the identity move only if it consists of couples of the kind $m_i \circ m_i^{-1}$, possibly in separated positions ($\circ$ is the composition operator). In addition, we assume that each move m_i has a unique inverse m_i^{-1} and that moves commute. This is trivially true for set-clear moves acting on single bits of a binary string, but not for the elementary exchanges of a permutation problem like QAP.

Before each iteration, the *residual cancellation sequence* (RCS) is constructed for all previous points, starting from the most recent ones. The RCS is the shortest sequence of moves leading to a previously encountered configuration (all couples $m_i \circ m_i^{-1}$ are canceled using the commutative property). A move m_i is tabu if its execution would repeat an old configuration, i.e., if during the backward tracing the RCS collapses to only m_i^{-1}. For additional details and modifications see [3, 7].

Let n be the number of iterations executed. Because the computational cost of each trace is proportional to the length of the running list (i.e., to the number of iterations), the total cost is proportional to n^2. An example is presented in Figure 6 for an application to the 0-1 knapsack problem labeled Weingartner8 defined in [18] and used in [16], with 105 binary variables and 2 constraints. The basic moves used in this case are the set-clear operations on individual bits.

A total of 16 runs with different random initial points has been executed. The data points are derived from actual measurements of the CPU time on a current Sun Sparc 2 Workstation. An interpolation of the points corresponding to a number of iterations greater than 100 gives the follow-

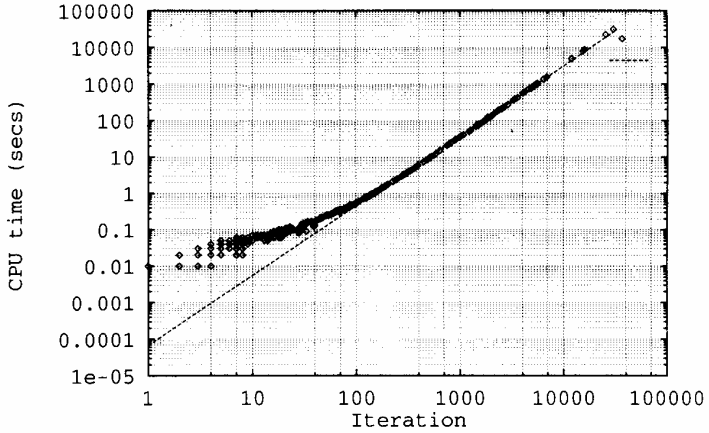

Figure 6. REM time complexity in logarithmic scale (diamonds) and interpolating curve (dashed line).

ing result:

$$CPUtime(sec) = 6.71 \times 10^{-5} n^{1.91}$$

The errors on the interpolation are 2% on the multiplicative constant and 0.1% on the exponent. The slight deviation from the quadratic form is caused by the influence of points with low iteration numbers: the quadratic term dominates only in the asymptotic limit. The code has not been optimized (a C++ programming language was used), so the multiplicative constant is not to be considered as the smallest obtainable.

The asymptotic behavior is unchanged by the modifications proposed for reducing the number of tracing steps (see the auxiliary memory structure Least used in Section 1.3 of [7]). It is obviously changed if the backward tracing is stopped after a maximum number of steps.

The *hashing* technique is standard in computer science (see, for example, [1]). The basic idea of hashing is that of storing entries in "buckets" whose index is obtained in a scattered way from the entry itself (by using the hashing function with the element as argument). The search time is approximately constant and equal to a very small number of machine cycles if the number of buckets is so large that, with a high probability, different entries end up in different buckets. In this last case, only a comparison with a single stored item is sufficient. One way of dealing with "collisions" (entries with the same bucket) is that of associating to each bucket a list of entries, which is enlarged when new elements arrive. The version that we describe (*open hashing*) is based on an array of "bucket table headers," which contain the pointer to the first entry in the associated list (**bucket[hash_val]↑ ϕ** contains the first stored placement in the list, **bucket[hash_val]↑last_time** the last time when it was encountered, **bucket[hash_val]↑repetition** the number of repetitions). Elements in the list are chained with the pointer **next** that gives the address of the next element. **nil** pointers are used for list termination. The number of buckets has to be larger in order to make collisions a rare event. A rule of thumb is to make it about two times (or larger than two times) the maximum number of stored entries, assuming that the hashing function scatters the entries in an almost uniform manner. Our use of the hashing technique is illustrated in Figure 7.

The *digital tree*[11] method stores binary strings (for example) using a binary tree structure where the decision about choosing the left or right child of a node at depth d depends on the value of the dth bit of the string. The storing time is proportional to the total number of bits (therefore, it is *constant* when the number of stored items grows). The same is true for the worst-case retrieval time, i.e., when the item is found. If it is not found, the search is terminated before reaching the deepest layer, as soon as the first **nil** pointer is encountered. In Figure 8 we show the memory configuration for the storage of strings (101111) and (100110).

We tested the digital tree technique on the 0-1 knapsack problem labeled Petersen7 (see [16]), with 50 variables, 5 constraints, and the same moves as those used for Weingartner8. We ran 60 tests with random starts, sampling the

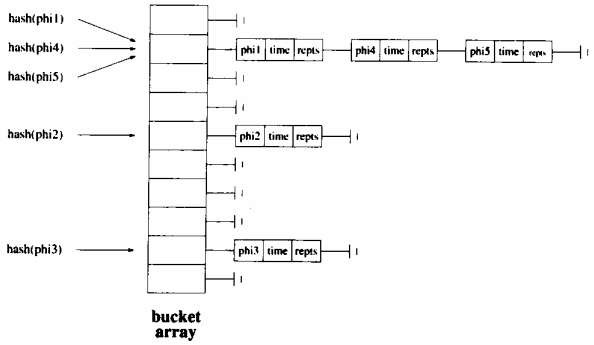

Figure 7. Memory configurations for the open hashing scheme.

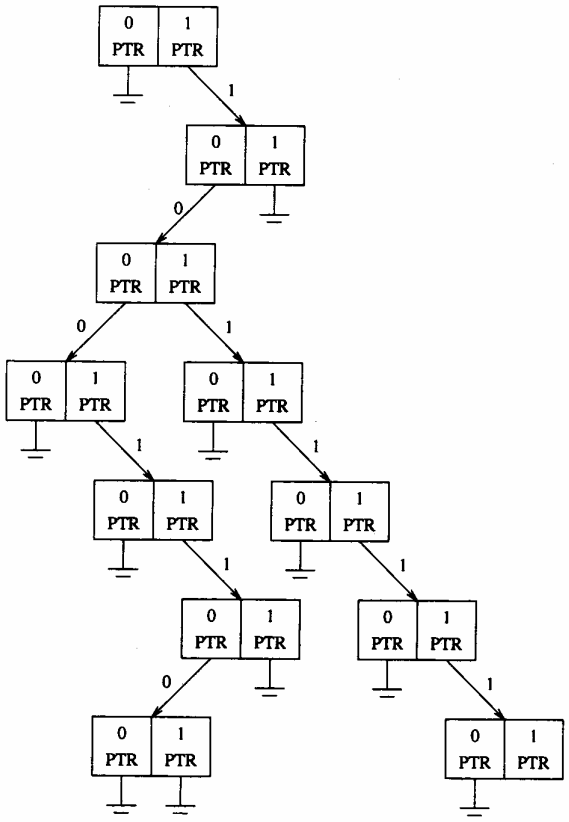

Figure 8. "Superposition" of the strings 101111 and 100110 stored in a digital tree.

memory usage (in terms of number of tree nodes used) at random times. Each node requires 8 bytes for the two pointers. The results are consistent with a linear increase (see Figure 9). The differences between the various runs are caused by the possible superpositions of the initial parts of different strings, which saves some nodes.

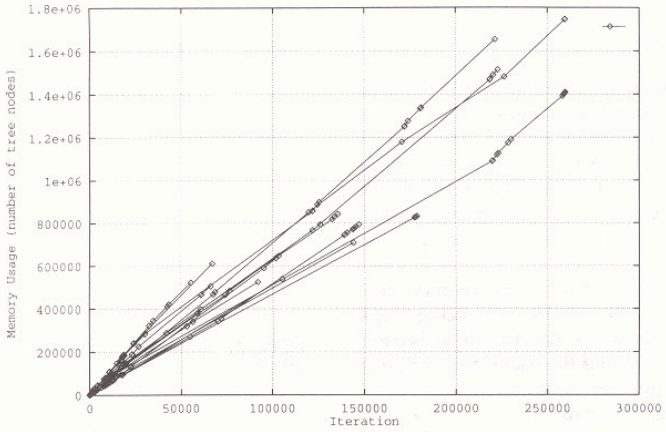

Figure 9. Observed memory growth for the digital tree scheme.

2. Efficacy and Efficiency: A Case Study

The set of basic moves for a tabu algorithm must satisfy a "completeness" criterion, i.e., the region of the search space "covered" by the algorithm starting from a randomly chosen starting point must not be too small with respect to the size of the search space. If $m_1, m_2, \ldots, m_k$ are the basic moves, X is the search space, and

$$\text{span}(x) = \{y \in X | y = m_{j1} \circ m_{j2} \circ \ldots m_{jk}\, x\},$$

i.e., the points that can be reached with chains of basic moves, the requirements that $\text{span}(x) = X$ for every $x \in X$ is a necessary condition for the efficacy of a tabu algorithm, if an exact solution is required. The basic moves often correspond to simple operations as in the case of the set-clear bit moves, but in certain applications they can be implemented by complex sequences of operations; see, for example, the DROP/ADD moves used in [16].

The efficacy of the search is obviously affected by the specific tabu strategy. In fact, the trajectory obtained by the step-by-step application of the allowed moves can show qualitatively different behaviors. To illustrate this fact, let us consider the problem of finding the global maximum of the following function:

$$F6(x, y) = 0.5 - \frac{\sin^2\left(100\sqrt{x^2 + y^2}\right) - 0.5}{(1 + 10(x^2 + y^2))^2} \qquad (1)$$

in the domain $[-1, 1] \times [-1, 1]$. The function, apart from a trivial scaling of the x and y coordinates so that the domain becomes $[-1, 1]$, is the same as the one described in [4] and used in [17].

In Figure 10 we show the central part of the F6 function. The global maximum is at the origin, with a very narrow basin around it. The function is "difficult" because there are many suboptimal solutions located on concentric circles that trap algorithms based on hill-climbing with a high probability.

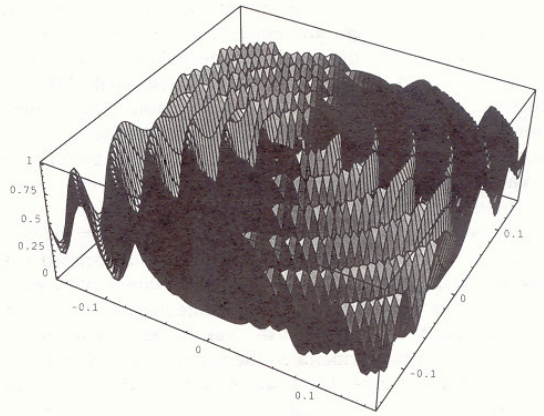

Figure 10. Function F6: Surface plot of the central part with its needle-like peak in the origin.

The problem of maximizing $F6$ becomes a combinatorial problem after choosing a discrete binary encoding of the continuous interval $[-1, 1]$. Two natural mappings between binary strings and (x, y) coordinates are obtained by discretizing each coordinate in 2^n evenly spaced points identified by integers $j_x, j_y = 0, 1, \ldots, 2^n - 1$, such that $x = (2/2^n) j_x - 1$, $y = (2/2^n) j_y - 1$, and then using the binary or Gray encoding of the integers j_x, j_y. The conversion between the binary encoding $b_n b_{n-1} \ldots b_1$ and the Gray encoding $g_n g_{n-1} \cdots g_1$ is as follows (see, for example, [10]):

$$\begin{cases} g_k = b_k & \text{if } k = n \\ g_k = b_{k+1} \oplus b_k & \text{if } k < n \end{cases}$$

$$\begin{cases} b_k = g_k & \text{if } k = n \\ b_k = b_{k+1} \oplus g_k & \text{if } k < n \end{cases}$$

where $\oplus$ is the exclusive-or operator and the second transformation must be done for decreasing values of k, starting from $k = n$.

In our test n is equal to 14, so that a total of 28 bits are used for the two coordinates. The elementary moves on the 28-bit binary string are setting-clearing individual bits. The corresponding elementary moves in $x - y$ plane depend on the encoding. The standard binary encoding leads to moves of different sizes, larger when the more significant bits are modified. Gray encoding causes a similar "multi-scale" set of moves, with the important addition that near points in the $x - y$ domain can be reached by changing a *single bit* of the string.

The efficacy and efficiency of the search for the different tabu schemes and encodings is shown in Table I which summarizes the results on 20 random starts for each version with a maximum of 16000 iterations each.

The set of moves associated to the Gray encoding allows a more effective search in every variant. Let us now discuss the main characteristics of the trajectories obtained in the different schemes.

Table I. Comparison of Different Tabu Schemes on the Problem F6

Binary Coding				
Variant	Succ. No.	Iter. No.	Unsucc. Values	Iter. No.
Fixed	2	13	0.888385	14
(List Size 7)		(11)	(0.026073)	(3)
Fixed	3	90	0.971304	92
(List Size 14)		(56)	(0.016403)	(226)
Fixed	8	3707	0.999956	208
(List Size 21)		(1876)	(4.19002e-06)	(104)
Strict	3	4448	0.989758	3282
		(4430)	(0.004806)	(1171)
Reactive	17	5055	0.999963	2435
		(1294)	(0.0)	(2211)
Gray Coding				
Variant	Succ. No.	Iter. No.	Unsucc. Values	Iter. No.
Fixed	1	4	0.911327	558
(List Size 7)		(0)	(0.035065)	(236)
Fixed	14	189	0.990284	105
(List Size 14)		(50)	(0.0)	(32)
Fixed	20	1623	0.0	0
(List Size 21)		(430)	(0.0)	(0)
Strict	20	3300	0.0	0
		(528)	(0.0)	(0)
Reactive	20	1554	0.0	0
		(391)	(0.0)	(0)

Number of successes, mean iterations number in the successful cases, mean best value and number of iterations to obtain it in the unsuccessful cases. Standard deviations in parentheses.

In Strict-Tabu the next point of the trajectory is the one with the highest function value among the new (i.e., not-yet-visited) configurations in the neighborhood. The trajectory tries to visit each point in a basin around a local maximum, although the multi-scale moves in the $x - y$ plane permit "jumps" if the large-size steps lead to a basin with higher function values.

This dynamic is acceptable from the efficacy point of view, but the efficiency is very low because every basin must be almost completely filled before a new region of the search space is entered. Actually, the discovery of the optimal point is not guaranteed: in some cases the search may be stuck if all moves would lead to previously visited points; in other cases the optimal point can be separated from the current one by "walls" of previously visited configurations and may never be reached. The probability of the above effects is low when the dimensionality of the problem is high, but it increases in the presence of constraints because they can limit the number of admissible moves. We found evidence of these results in problems with many constraints and using a simple set of elementary moves (like the 0-1 knapsack problem with single-add and single-drop moves). It is difficult to predict the impact of the above complex dynamics on a specific problem, although the "basin-filling" effect can become worse when the dimension D of the problem grows. In fact, an attraction basin of radius ρ contains a number of points proportional to ρ^D, so that the filling can become very time-consuming. The walls can be superated by allowing "tunneling", i.e., a passage over old configurations if this leads to better regions. Tunneling may be favored by complex basic moves. The "pedantic" way to explore is clearly shown in Figure 11, where the dynamics of the algorithm is depicted in the case of a typical successful run. The high mean value of the number of iterations necessary to find the solution in the case of a successful run (see Table I) is another indicator of this "almost exhaustive" type of search.

At this point, let us note that we do not discourage the use of S-TABU in a general way, in fact S-TABU required the minimum number of iterations for some QAP problems illustrated in Section 4, although the actual CPU time is larger than the time of R-TABU.

In Fixed-Tabu (i.e., tabu based on a fixed list_size), the next point of the trajectory is chosen among the configurations obtained by applying the moves which have not been used for a number of iterations given by list_size, unless the aspiration criterion is satisfied. The trajectories are more "jagged": the exploring point describes orbits surrounding local maxima without clear regular patterns. The basins are sampled without visiting every point. Obviously, the range of the explored region surrounding a local maximum grows with the list_size parameter and, if the attraction basin associated to a maximum is larger than the maximum "exploration range," the trajectory remains indefinitely trapped.

Fixed-Tabu on the F6 function does not converge frequently (it remains trapped), although it converges rapidly if the initial point is suitable. The success frequency, and also the number of iterations, increases with large sizes.

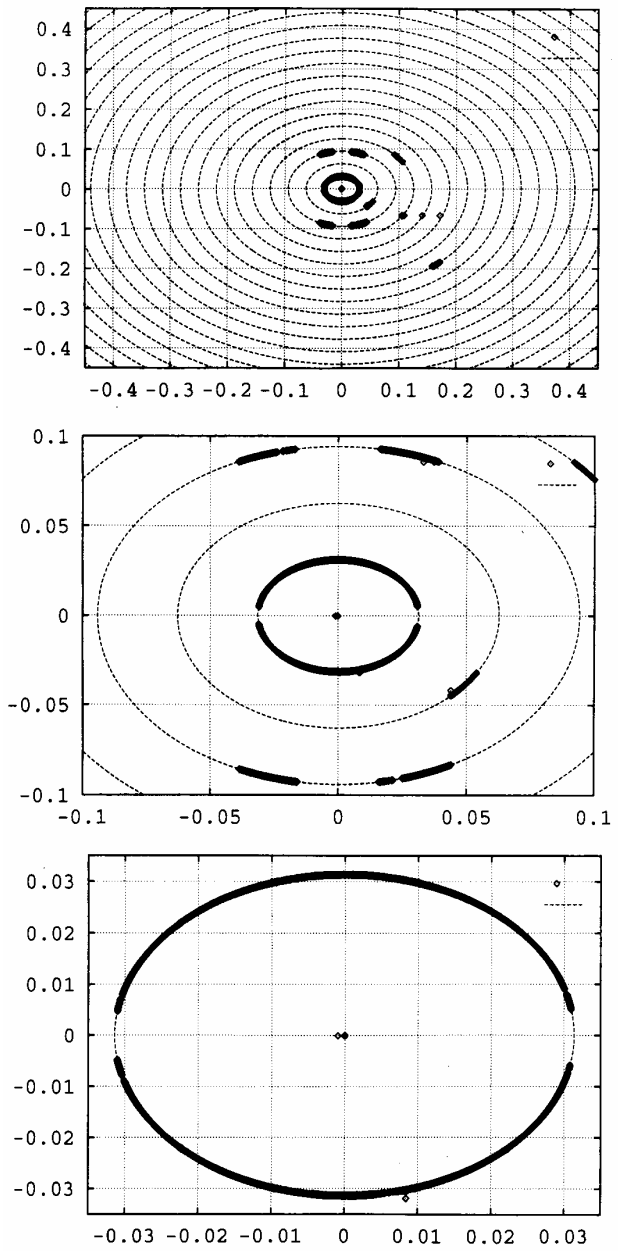

Figure 11. Visited point of function F6 for the strict-tabu method. Local maxima are located on the concentric lines (7227 iterations). The figures show the same region around the global optimum at different resolutions.

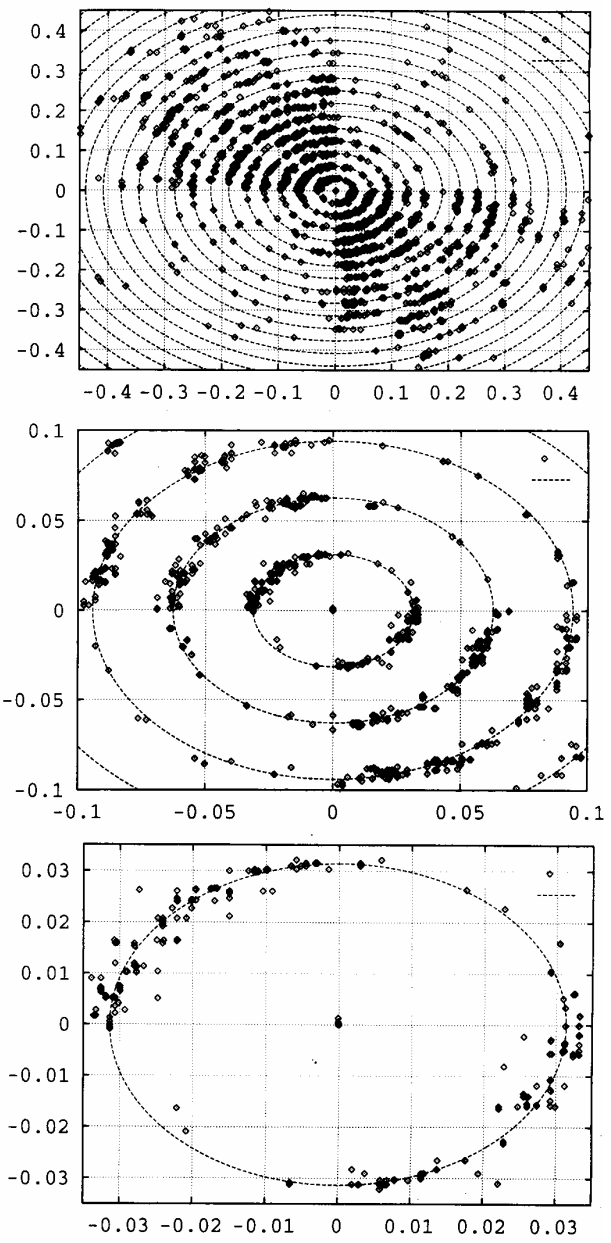

Figure 12. Visited points of function F6 for the fixed-tabu method. Local maxima are located on the concentric lines (3577 iterations). The figures show the same region around the global optimum at different resolutions.

This conclusion is derived from Table I, where the results of runs using different list_size values are given. The efficiency is clearly higher than in the case of the Strict-Tabu search.

In Figure 12 we show a successful run with list_size = 21. Let us note that the points are more scattered than the points of Figure 11, corresponding to the S-TABU case. The above considerations remain true also in the case of variable-length list algorithms,[15] if the range of variation is limited.

The dynamics of the Reactive-Tabu method show characteristics of the fixed-tabu search, but the unsuccessful cases

are completely absent with Gray encoding and rare with the standard binary encoding. Actually the three unsuccessful cases converged to a point next to the optimal configuration in the $x - y$ plane (with $F6 = 0.999963$ instead of 1.0). The maximum could not be reached in the allotted time because this required changing the binary string of one coordinate (x or y) from 01111111111111 to 10000000000000, i.e., *all* bits (but only a single bit with Gray encoding). A typical trajectory is shown in Figure 13.

Let us note a similarity of the trajectory with the trajectory of the S-TABU on a large scale, and with that of F-TABU in the small scale.

3. Space-Time Costs of REM, Hashing, and Digital Tree

To terminate the comparison among the different versions of tabu search, let us concentrate on their time and memory requirements. It is clear that the need to store the whole set of visited configurations and the need to check if a candidate point was already visited can affect the memory requirements and the CPU time.

Table II collects the asymptotic expressions for the space (memory) and time (CPU secs) complexity of the different schemes. In the two first columns we isolate the dependency on the number of iterations n, and in the last one we highlight the application dependencies. Let us note that the REM time complexity is high, being proportional to the square of the iteration numbers. Therefore, the REM scheme for S-TABU is convenient only when the memory cost of a single configuration is very high with respect to the cost of storing a single move.

In Table II, n denotes the number of iterations, N the problem size, f the function to be optimized, and $C(f, N)$ the computational cost for evaluating the neighborhood containing $|S|$ points, a number depending on the problem size. The constant k_0 is the cost of the single tracing step of REM, $\overline{k_1}$ is the average fraction of number of configurations evaluated in the neighborhood, D_0 is the cost of a single fetch-and-test operation on the node of the digital tree, and H_0 and H_1 depend on the specific hashing scheme (H_0 in the case of storing the whole configuration, H_1 in the case of storing a single compressed item). Let us note how the dependence on the factor $|S|$ is canceled in the expression for the time complexity of R-TABU with respect to S-TABU. This fact can reduce the computational cost, especially for large neighborhoods.

The space-time complexity of the hashing variant is a little higher than in the digital tree case, but it can be reduced if the hashing mechanism implements a compression mechanism, as described in [17], where the vector describing the configuration is "shrunk" into a 16-bit datum.

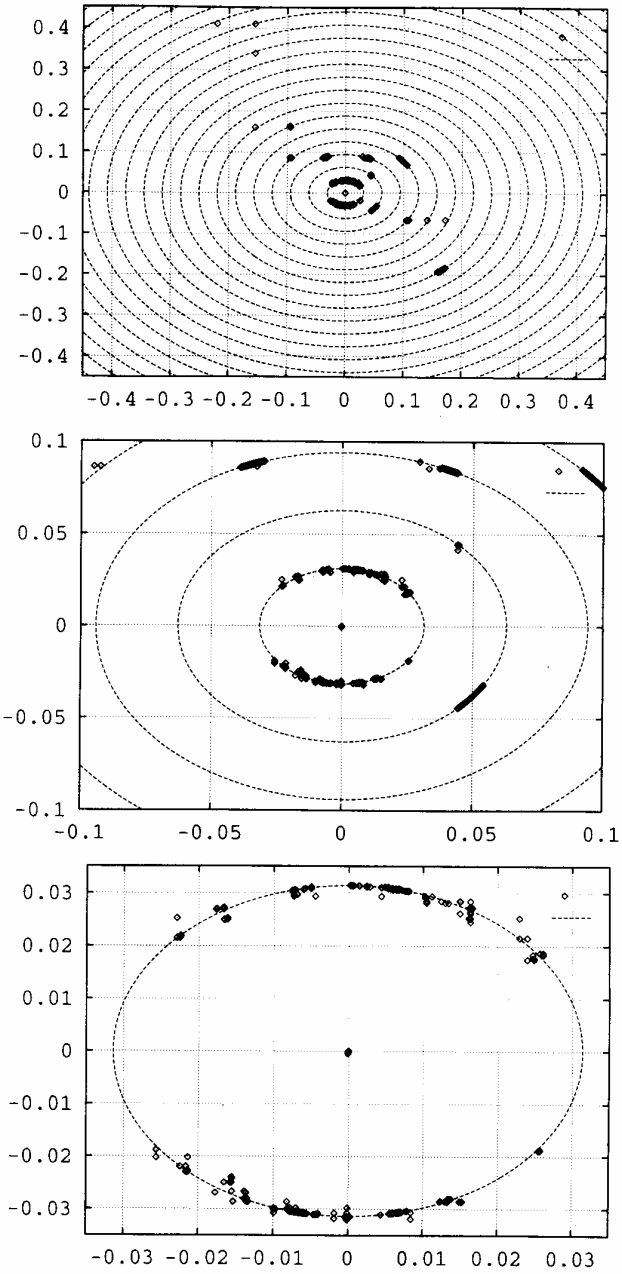

Figure 13. Visited points of function F6 for the reactive-tabu method. Local maxima are located on the concentric lines (1966 iterations). The figures show the same region around the global optimum at different resolutions.

4. Results on the Quadratic Assignment Problem

In the Quadratic Assignment Problem of size N, the function to be minimized is:

$$f(\phi) = \sum_{i=1}^{N} \sum_{j=1}^{N} a_{ij} b_{\phi(i)\phi(j)},$$

where the search space consists of the set of all possible permutations ϕ of N integers. The practical relevance of the problem is clear when ϕ is interpreted as the assignment of N units to N locations ($\phi(loc)$ is the unit assigned to location loc), the matrix element a_{ij} represents the distance between the locations i and j, and the element b_{ij} is

Table II. Asymptotic Requirements of CPU Time and Memory Space for Different Tabu Schemes

Tabu Variant	Time Complexity	Space Complexity	Problem Dependencies
Fixed	$\alpha_0 n$	Constant	$\alpha_0 = C(f, N)$
Strict (REM)	$\alpha_1 n^2 + \alpha_2 n$	n size of (MOVE)	$\alpha_1 = \frac{k_0}{2}$ $\alpha_2 = \overline{k_1} C(f, n) - \frac{k_0}{2}$
Strict (DTREE)	$\alpha_3 n$	n size of (NODE)	$\alpha_3 = D_0 N\|S\| + C(f, N)$
Strict (HASH)	$\alpha_4 n$	n size of (ENTRY)	$\alpha_4 = H_0 N\|S\| + C(f,N)$ or $\alpha_4 = H_1\|S\| + C(f, N)$
Reactive (DTREE)	$\alpha_5 n$	n size of (NODE)	$\alpha_5 = D_0 N + C(f, N)$
Reactive (HASH)	$\alpha_6 n$	n size of (ENTRY)	$\alpha_6 = H_0 N + C(f, N)$ or $\alpha_6 H_1 + C(f, N)$

the "flow" from location i to location j. Solving the QAP problem means searching for an assignment that minimizes the sum of the products "distance" times "flow" (the "transportation cost").

The problems used for the tests were created by using the pseudo-random procedure described in [15]. The symmetric and zero-diagonal matrices a_{ij} and b_{ij} are filled starting from the values obtained from a random number generator defined by the following recursive formula:

$$X_k = (aX_{k-1}) \mathbf{MOD} m,$$

where $a = 16807$, $m = 2^{31} - 1$, and $X_0 = 123456789$ (integers coded on 64 bits). The pseudo-random numbers are scaled and converted into integers in the range (0, 99). In detail, the elements a_{ij} above the diagonal are filled in a row-wise manner by using successive X_k values (X_1, X_2, X_3, ...) as: $a_{ij} \leftarrow \lfloor (100\ X_k)/m \rfloor$, and the lower part of the matrix is obtained from the symmetry requirement. The elements b_{ij} are then defined by "consuming" additional X_k values in the same way.

The elementary moves for the problem consist of all possible exchanges of the locations occupied by two units. Following the notation of [15], a new placement (permutation) π is obtained from the current placement ϕ by exchanging two units r and s:

$$\pi(k) = \phi(k) \quad \forall k \neq r, s;$$
$$\pi(r) = \phi(s); \qquad \pi(s) = \phi(r).$$

The complete evaluation of the neighborhood requires $O(N^2)$ operations. In the case of symmetric and null-diagonal matrices, the *value* of a move that brings from state ϕ to state π (i.e., the reduction $\Delta(\phi, r, s) \equiv f(\phi) - f(\pi)$) is:

$$\Delta(\phi, r, s) = 2 \sum_{k \neq r, s} (a_{sk} - a_{rk})(b_{\phi(s)\phi(k)} - b_{\phi(r)\phi(k)}). \quad (2)$$

If the move values starting from a configuration ϕ are stored, the move values for the new configuration π (obtained from ϕ by exchanging units r and s) can be calculated in constant time for u, v different from r or s by using:

$$\Delta(\pi, u, v) = \Delta(\phi, u, v) + 2(a_{ru} - a_{rv} + a_{sv} - a_{su}) \times (b_{\phi(s)\phi(u)} - b_{\phi(s)\phi(v)} + b_{\phi(s)\phi(v)} - b_{\phi(r)\phi(u)}). \quad (3)$$

We performed a series of computation tests by running different versions of the tabu algorithm (S-TABU, R-TABU with or without the escape mechanism) starting with different random initial points (the same for the different algorithms). For comparison we report the mean values obtained by the *robust tabu* scheme of [15].

In Table III we list the expected number of iterations for convergence to the best known solution listed in [15] and the standard deviation of the estimates. Each iteration consists of the complete neighborhood evaluation and the selection of the best move among those satisfying the tabu or aspiration requirements. We ran a total of 30 tests for problem sizes ranging from 5 to 35. All tests reached the desired target solution.

It can be noted that the *reactive tabu* is competitive with the *robust tabu*, especially for large problem sizes. The larger number of iterations for small problem sizes ($N \leq 12$) is expected because the R-TABU scheme needs a small

umber of iterations in the start-up phase, when an appro-riate list-size is "learned" from the evolution of the search et us remember that the initial size is one). Nonetheless, ıe R-TABU scheme pays off for large problem sizes, where ıe convergence to the optimal configuration is obtained in robust way without having to define at the beginning a ıitable list-size (or range of sizes).

The performance of the *strict tabu* scheme *in terms of erations* is good for this problem: for large problem sizes ıe average number of iterations for convergence is re-uced with respect to both R-TABU and S-TABU, but the lvantage is lost because of the larger CPU time per eration (for the $N = 35$ case, S-TABU is about 3.5 times ower than R-TABU per iteration).

The number of iterations tends to be proportional to the :tual CPU time in the same manner for the *reactive* and)bust versions. In fact, (see Section 3) the time for updat-ıg the hashing or the digital tree memory structure is pproximately $O(N)$, and because the neighborhood evalu-tion requires $O(N^2)$ iterations, the memory-updating ɔmponent tends to be negligible for large problem sizes. The case of S-TABU is different because for each iteration all the $O(N^2)$ points in the neighborhood have to be compared with stored configurations, with a total cost of $O(N^3)$, so that this is the dominant term for large N. If only the function values are stored (see below), one obtains a not negligible cost of $O(N^2)$, of the same order as that for the neighborhood evaluation.

The CPU time per iteration on a state-of-the-art workstation (Iris from Silicon Graphics) is approximately 6.7 N^2 μs per iteration. This value, like the relative speed of R-TABU vs. S − TABU, was obtained by using a C-language program and the standard *cc* compiler.

4.1. Discussion of R-TABU Choices

In the following series of tests we probe the functionality of R-TABU for changes in the design of the algorithm. First, we eliminated the escape mechanism and changed the speed with which the list-size is increased or decreased (see the INCREASE and DECREASE parameters in Section 1.4). In Table IV we present the results obtained from a series of 30 tests for each problem size (ranging from 5 to 20). The

Table III. Comparison of Different Schemes of Tabu Search

Size (N)	Tests (max. iter.)	R-TABU [1.1, 0.9, esc]	S-TABU	Robust TABU
5	30 (max.100K)	9.9 (1.9)	6.1 (0.5)	7.6
6	30 (max.100K)	12.2 (2.4)	7.4 (0.9)	6.6
7	30 (max.100K)	78.1 (12.2)	35.6 (3.1)	25.7
8	30 (max.100K)	40.9 (5.7)	32.5 (3.8)	29.4
9	30 (max.100K)	67.5 (12.0)	56.2 (8.1)	31.7
10	30 (max.100K)	256.7 (34.0)	161.3 (20.7)	137.1
12	30 (max.100K)	282.3 (51.4)	477.0 (95.7)	210.7
15	30 (max.100K)	1780.3 (319.0)	3642.2 (308.2)	2168.0
17	30 (max.100K)	4133.9 (646.8)	7364.2 (817.4)	5020.4
20	30 (max.500K)	37593.2 (6012.5)	25092.9 (6572.2)	34279.0
25	30 (max.1M)	38989.7 (6236.1)	20483.9 (3575.0)	80280.4
30	30 (max.2M)	68178.2 (11370.3)	48919.2 (9055.6)	146315.7
35	30 (max.4M)	281334.0 (48543.5)	146276.2 (47419.7)	448514.5(*)

R-TABU version with INCREASE = 1.1, DECREASE = 0.9, and escape mechanism. The standard deviation of the measured average is given in parenthesis. (*) needed the introduction of a long term memory mechanism.

Table IV. Robustness for Parameter Changes: R-TABU Without Escape, Different Values of INCREASE and DECREASE

N	INC = 1.1, DEC = 0.9	INC = 1.2, DEC = 0.9	INC = 1.1, DEC = 0.8	INC = 1.2, DEC = 0.8
5	9.9 (1.9)	8.0 (1.3)	9.9(1.9)	8.0 (1.3)
6	13.8 (2.6)	10.4 (1.8)	13.2 (2.4)	10.6 (1.9)
7	77.2 (10.1)	46.4 (5.8)	81.6 (10.3)	61.2 (7.8)
8	33.4 (4.2)	36.4 (5.8)	36.8 (4.2)	33.4 (3.9)
9	56.6 (11.7)	47.0 (7.1)	59.8 (8.8)	50.4 (7.0)
10	277.4 (45.0)	199.1 (27.3)	221.7 (35.0)	240.7 (36.1)
12	181.0 (37.4)	187.6 (35.2)	195.6 (33.5)	157.6 (19.9)
15	1962.4 (379.5)29/30	1827.7 (335.4)29/30	2153.7 (387.7)	2241.5 (393.0)29/30
17	3890.3 (556.1) 26/30	5767.3 (1258.5)25/30	4136.2 (794.5)24/30	4784.8 (796.7)25/30
20	13452.9 (4489.4)7/30	15710.5 (3988.3)21/30	19856.1 (4599.2)14/30	17997.3 (3156.1)19/30

INCREASE and DECREASE parameters are written at the top of each column. When the algorithm does not reach the optimum in the allowed maximum number of iterations listed in Table III, we report the proportion of optimal results in 30 runs.

While the results are acceptable for the smaller problems (up to $N = 12$), starting from $N = 15$ we observed that the algorithm fails for a growing fraction of runs. The size dynamics is not sufficient to avoid traps. The search either reaches the optimum in a relatively small number of iterations or it does not reach it at all. A possible explanation is that the algorithm is visiting only a limited portion of the search space, a portion that contains the optimal point in the lucky cases and only sub-optimal values in the remaining ones. The cancellation of limit-cycles with the list-size dynamics does not guarantee the success, and the additional escape mechanism is therefore needed in the algorithm (see also the discussion of chaotic attractors in Section 1).

In a second series of tests, we included the escape mechanism and tested different speeds for list-size variation (see Table V). Success is obtained in *all* cases, and the number of iterations is not affected in a critical way, justifying the choice of fixed values 1.1 and 0.9 for all tests.

In the last column of Table V, we modified the memory mechanism so that the function value is recorded instead of the configuration, the same method used in [2]. Because the same function value can be associated with *different* configurations, there is a small probability of "false alarms," i.e., reactions of the algorithm when there is no actual repetition of configurations. The advantage of the method is that the memory requirement is reduced: only a single 32-bit integer is stored instead of the entire configuration. The tests show no statistically significant difference with respect to the case when the precise configuration is saved.

More sophisticated "compression" techniques are described by Woodruff and Zemel,[17] where a hashing function is used to compress the vector describing the configuration. To adapt our hashing algorithm described in Section 1.5 to their proposal, it is sufficient to use the entries of the *bucket array* (see Figure 7) as flags for the existence of a configuration with the given index. In this case a single bit is sufficient for each slot.

4.2. New Sub-Optimal Solutions

Encouraged by the results obtained in the previous sections, we ran a series of tests for larger problem sizes (from $N = 40$ to $N = 100$). While for the smaller sizes we duplicated the optimal values listed in [15] and could not reach lower values (therefore confirming their status of "provably or probably optimal solutions"), for the larger sizes we could surpass all best known solutions listed in the cited paper, often by large relative amounts. The new obtained solution values and the percent below Taillard's values are listed in Table VI.

For the $N = 40$ case we ran a total of 10 tests, stopping when Taillard's value was reached or overcome. In all cases

Table V. Robustness for Parameter Changes R-TABU With Escape. Different Values of INCREASE and DECREASE

N	INC = 1.2, DEC = 0.9	INC = 1.1, DEC = 0.8	INC = 1.2, DEC = 0.8	INC = 1.1, DEC = 0.9, f
5	8.0 (1.3)	9.9 (1.9)	8.0 (1.3)	13.0 (2.3)
6	10.4 (1.8)	11.2 (2.1)	10.6 (1.9)	10.6 (1.7)
7	59.0 (12.7)	77.3 (9.2)	63.8 (11.3)	98.5 (13.1)
8	44.1 (7.6)	40.4 (5.5)	45.1 (6.6)	39.9 (5.3)
9	62.3 (6.9)	80.2 (8.5)	53.4 (5.7)	58.5 (7.7)
10	210.3 (28.4)	181.6 (29.1)	254.7 (39.9)	236.1 (33.5)
12	203.9 (57.6)	218.6 (30.1)	195.3 (39.3)	234.7 (40.2)
15	1981.0 (357.3)	1677.3 (222.2)	1994.7 (370.9)	1828.3 (519.1)
17	4408.5 (699.4)	4487.9 (918.3)	4215.5 (606.7)	4347.3 (764.6)
20	51648.5 (11499.3)	23652.6 (4117.1)	29230.3 (6440.7)	46019.6 (7567.9)

Table VI. Best Solutions Obtained and Percent Reduction with Respect to Taillard's Values

N	New Best	Percent	Iteration
40	3141702	−0.1529	1048900.2 (295738.0) 10 tests
50	4948508	−0.0514	7628548, 1 test
60	7228214	−0.6024	3071920, 1 test
80	13558710	−0.1718	4767363, 1 test
100	21160946	−0.3993	542561, 1 test

R-TABU with INCREASE = 1.1, DECREASE = 0.9, escape and storage of f values.

this value was reached, in 6 out of 10 cases the new best value ($f = 3141702$) was obtained. Excessive computing times prohibited extensive tests for larger sizes, but the results obtained in a single test (for $N = 50$, 60, 80, and 100) are extremely encouraging. In particular the optimal solution for $N = 100$ was ameliorated by almost 0.4% in about 500K iterations.

5. Conclusions

The Tabu technique pioneered the use of flexible memory structures in the search process. In the present work we presented both an overview of efficient storing and retrieval techniques to speed-up the search, and a new *reactive* version of tabu, where the appropriate size of the tabu list is adapted to the history of the search process.

The *hashing* and *digital tree* storing and retrieval methods permit the rapid comparison of a candidate configuration with all points previously encountered, in $O(1)$ time. These fast mechanisms can be used as the building block of both the *reactive* and *strict* versions of tabu. In the strict case, when the only forbidden moves are those leading to previously visited points, the trajectory obtained is the same as that of the Reverse Elimination Method, the difference being in the search speed.

The *reactive tabu* with the escape diversification technique and the exploitation of fast memory structures does not need the *a priori* choice of the list size and shows a robust and efficient convergence on the chosen test problems. An additional use of hashing functions is that advocated in [17], where a configuration vector is mapped to a "compressed" datum given by its hashing index. A comparable compression can be obtained by storing the function values, the method that we used for experimenting on the large-size QAP problems. Apparently the occurrence of the same function values for different configurations does not impair the efficacy and efficiency of the search.

The utility of the reaction mechanism, as compared to the strict cycle-avoidance, confirms that avoiding cycles is not the ultimate goal of the search process,[8] the broader objective being that of stimulating a "bold" exploration of the search space.

A straightforward parallel implementation of a primitive version of R-TABU was presented in [2], where independent searches are executed in the different nodes. We are now experimenting with the use of the above mentioned memory structures in the fully parallel case, where the information contained in a set of suboptimal configurations is used to create a new set of candidate points (see also [9]).

Acknowledgments

The authors would like to thank Professors Fred Glover, Dave Woodruff, and Stefan Voss for useful comments and for sending both relevant preprints and the data for some benchmark problems used in the paper. The anonymous referees helped to improve the clarity of the paper. The hardware facilities for the computational tests were kindly made available by the I.N.F.N. group of the University of Trento.

References

1. A.V. Aho, J.E. Hopcroft and J.D. Ullman, 1985. *Data Structures and Algorithms*, Addison-Wesley, Reading, MA.
2. R. Battiti and G. Tecchiolli, 1992. Parallel Biased Search for Combinatorial Optimization, *Microprocessors and Microsystems 16:7*, 351–367.
3. F. Dammeyer, P. Forst and S. Voss, 1991. On the Cancellation Sequence Method of Tabu Search, *ORSA Journal on Computing 3*, 262–265.
4. L. Davis, 1991. *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, New York.
5. F. Glover, C. McMillan and B. Novick, 1985. Interactive Decision Software and Computer Graphics for Architectural and Space Planning, *Annals of Operations Research 5*, 557–573.
6. F. Glover, 1989. Tabu Search—Part I, *ORSA Journal on Computing 1:3*, 190–206.
7. F. Glover, 1990. Tabu Search—Part II, *ORSA Journal on Computing 2:1*, 4–32.
8. F. Glover and M. Laguna, 1993. Tabu Search, in *Modern Heuristic Techniques for Combinatorial Problems*, C.R. Reeves (ed.), Blackwell Publishing, Oxford, 70–150.
9. F. Glover, J. Kelly and M. Laguna, 1992. Genetic Algorithms and Tabu Search: Hybrids for Optimization, Manuscript, University of Colorado, Boulder.
10. D.F. Elliot and K.R. Rao, 1982. *Fast Transforms, Algorithms, Analyses Applications*, Academic Press, Orlando, FL.
11. D.E. Knuth, 1973. *The Art of Computer Programming Vol. III: Sorting and Searching*, Addison-Wesley, Reading, MA.
12. R. Serra and G. Zanarini, 1990. *Complex Systems and Cognitive Processes*, Springer-Verlag, Berlin.
13. H.G. Shuster, 1984. *Deterministic Chaos*, Physik-Verlag, Weinheim.
14. J. Skorin-Kapov, 1990. Tabu Search Applied to the Quadratic Assignment Problem, *ORSA Journal on Computing 2:1*, 33–45.
15. E. Taillard, 1991. Robust tabu search for the quadratic assignment problem, *Parallel Computing 17*, 443–455.
16. F. Dammeyer and S. Voss, 1993. Dynamic Tabu List Management Using the Reverse Elimination Method, *Annals of Operations Research 41*, 31–46.
17. D.L. Woodruff and E. Zemel, 1993. Hashing Vectors for Tabu Search, *Annals of Operations Research 41*, 123–137.
18. H.M. Weingartner and D.N. Ness, 1967. Methods for the Solution of the Multi-Dimensional 0/1 Knapsack Problem, *Operations Research 15*, 83–103.