ELSEVIER

Discrete Applied Mathematics 119 (2002) 3–36

DISCRETE APPLIED MATHEMATICS

Tabu search and finite convergence ☆

Fred Glover[a,*], Saïd Hanafi[b]

[a] *Hearin Center for Enterprise Science, School of Business Administration, University of Mississippi, MS 38677 USA*

[b] *LAMIH - UMR CNRS no. 8530, Unité de Recherche Opérationnelle et d'Aide à la Décision, Université de Valenciennes et du Hainaut-Cambrésis, Le Mont Houy, B.P. 311 - 59304 Valenciennes Cedex, France*

Received 1 October 1999; received in revised form 1 December 2000; accepted 26 January 2001

Abstract

We establish finite convergence for some tabu search algorithms based on recency memory or frequency memory, distinguishing between symmetric and asymmetric neighborhood structures. These are the first demonstrations of explicit bounds provided by such forms of memory, and their finiteness suggests an important distinction between these ideas and those underlying certain "probabilistic" procedures such as annealing. We also show how an associated reverse elimination memory can be used to create a new type of tree search. Finally, we give designs for more efficient forms of convergent tabu search. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: Tabu search; Annealing; Recency memory; Frequency memory; Tree search

1. Introduction

We consider a combinatorial optimization problem stated in the form:

(P) Minimize $c(x)$ subject to $x \in X \subseteq E$,

where E is the space of potential solutions that satisfy certain fundamental constraints and X is the set of feasible solutions that must satisfy additional (usually more complex) constraints defined by the problem application. The objective function c is a linear or nonlinear mapping that assigns a real cost value $c(x)$ to each solution x. The problem is to find a globally optimal solution $x^* \in X$ such that $c(x^*) \leqslant c(x)$ for all $x \in X$.

☆ Results of this paper were originally presented at the INFORMS meeting, Seattle, 25–28 October, 1998.

* Corresponding author. Leeds School of Business, University of Colorado, Campus Box 419 Boulder, CO 80309-0419, USA. Tel.: +1-303-492-8589; fax: +1-303-492-5962.
E-mail address: fred.glover@colorado.edu (Fred Glover).

PII: S0166-218X(01)00263-3

Many optimization techniques (both heuristic and exact) for solving problem (P) are iterative procedures that start with an initial solution (feasible or infeasible) and repeatedly construct new solutions from current solutions by searching neighborhoods. The process continues to generate neighboring solutions until a certain stopping criterion is satisfied. Each solution $x \in E$ has an associated neighborhood $N(x)$, a subset of E, and the step by which the solution $x' \in N(x)$ is reached from the solution x is called a *move*.

From a graph perspective an iterative solution search method can be viewed as a walk in a digraph $G_N = (V, A)$ induced by the structure of the neighborhood N, where the node set V is the set of solutions E and where an arc $(x, x') \in A$ exists if and only if $x \in N(x')$. Generally, the imprint of the trajectory in graph G_N is an elementary path in the case of local methods (forms of a descent method), while for certain meta-heuristics the itinerary constitutes a more complex path that may be neither node-simple nor arc-simple.

The adaptive memory of tabu search (TS) includes a mechanism that forbids the search to revisit solutions already encountered unless the intervening trajectory is modified [1]. The main goal of memory structures in TS is not simply to forbid cycling, and in fact, the choice of a given neighborhood and a decision criterion for selecting moves with TS can force some solutions to be revisited before exploring other new ones. An example occurs in a proposal of Glover [2], which identifies a simple rule for revisiting solutions accompanied by a conjecture that such a rule has implications for finiteness in the zero-one integer program and optimal set membership problems. Hanafi [3] proves Glover's conjecture under the assumption that the graph of the neighborhood space is connected and symmetric. In this paper, we provide new proofs that yield specific bounds establishing the finite convergence of this TS proposal. Our results provide insights into the sequences of solutions generated by the search which disclose interesting contrasts with the more rigid rules underlying tree search methods. Based on these outcomes, we also give designs for more efficient forms of convergent tabu search, and provide special rules that create a new type of tree search.

The outline of this paper is as follows. Section 2 describes two convergent tabu search algorithms (CTS) based on recency-memory and frequency-memory, respectively. We show that the complexity of the search differs according to whether the neighbor graph G_N is symmetric or asymmetric, and for the asymmetric case demonstrate that the number of steps required by the CTS algorithm to visit all solutions in X is an exponential function of the cardinality of X. In Section 3, we propose an approach for accelerating the classical tabu search *Aspiration by Default* rule in this setting, which may transform an exponential search into a much faster polynomial search. Section 4 presents a tabu tree search (TTS) for the symmetric case, with enhancements of TTS for reducing the number of operations that are devoted to scanning neighbors of solutions visited. Section 5 gives some comparisons with other approaches in the literature. Finally, some practical considerations are described in Section 6.

2. A convergent tabu search (CTS) algorithm

2.1. A convergent algorithm based on recency-memory

Let Time(x) = the most recent time (iteration) that solution x was visited by a search process, whose form is determined as follows.

Initialization assumption (IA). The values Time(x), $x \in X$, begin as arbitrary non-negative integers, and the starting solution x^* for the search is assigned a value so that Time(x^*) > Time(x) for all x other than x^*. (The "step counter" that is incremented by 1 at each successive move to determine the new value of Time(x), each time a solution x is visited, begins at the initial value of Time(x^*).)

This assumption of course includes the case where the method begins with Time(x) = 0 for all $x \in X$ except x^*.

Method assumption (MA). From any current solution x', the search will choose next to visit a previously unvisited solution, $x'' \in N(x')$ if one exists, and otherwise will choose to visit a solution $x'' = \text{argmin}\{\text{Time}(x): x \in N(x')\}$.

Remark 1. By convention, we may define Time(x) = 0 if x has never been visited. Then MA simplifies to say that we always move to a solution x'' that satisfies $x'' = \text{argmin}\{\text{Time}(x): x \in N(x')\}$. (Note x'' may not be uniquely determined in the set given by Time(x) = 0.) Moreover, the term Time(x) can be replaced with Time(x', x), identifying the most recent iteration x was visited from x', and all the observations following continue to hold.

The "min{Time(x)} rule" is the one called the *Aspiration by Default* rule in the TS literature. This rule might also be called the *earliest time stamped neighbor* rule, since the "last label" is a time stamp that tells when a node was visited. This time stamp is a dynamic one, because the *Time* stamp label can write over itself, and thus erase an earlier time stamp. This is important, because if the method under consideration only used a simple version of an *earliest time stamped neighbor* rule, without allowing the time stamp to write over itself, then it might avoid some duplications but it could also fail to search the entire space.

Neighborhood assumption (NA). X is finite and there exists a neighborhood path from every solution in X to every other solution in X.

The three preceding assumptions IA, MA and NA define the framework for a particular method we will call CTS-Simple. We identify properties of this method as follows.

Denote the cardinality of X by $n = |X|$, and consider a value V_n for $n \geqslant 2$ which is given recursively by $V_2 = 1$ and $V_{n+1} = n(V_n + 1)$. The value V_n is a very loose upper

bound for establishing finiteness of a search that operates according to the assumption MA, given a neighborhood space that satisfies assumption NA.

Theorem 1. *Starting from any solution in X, the CTS-Simple method will visit every other solution in X in a number of steps bounded above by* V_n.

Proof. The value V_2 is evident. By induction, suppose the theorem is true for a given value n and consider the case for $n+1$ (i.e., where $|X| = n+1$). Let X' denote a subset of X consisting of solutions visited in V_n steps. If X' is not X, then by assumption we are assured that X' contains all of X except a single solution x. Assumption NA implies there exists some $x' \in X'$ that includes x in its neighborhood.

Let v be the number of steps required to visit x' the first time, where $v \leqslant V_n$. Possibly x is visited on step $v+1$, but if not, step $v+1$ visits another solution $x'' \in X'$, and Time(x'') becomes greater than Time(x). Then, either x will be visited in the next V_n steps or else the search continues to be confined to X', in which case x' will be visited. Continuing in this way, each time x' is visited but x is not, some $x'' \in N(x')$ is visited and assigned a value Time(x'') > Time(x). Each new x'' visited from x' must be different from all others previously visited from x', or else Time(x) would have a value smaller than all other elements of $N(x')$. The number of times this process can continue is bounded by $|N(x')|$; i.e., after visiting x' for the first time on step $v \leqslant V_n$, once x' is visited an additional number of times $vAdd \leqslant |N(x')| - 1$, the search process is compelled to move to x on the next step. A count of the number of steps required to reach x is therefore bounded above by $v + vAdd(V_n + 1) + 1$ (where $vAdd$ is multiplied by 1 more than V_n because of the extra step that moves from x' back into X' to restart each round). Given $v \leqslant V_n$, $vAdd \leqslant |N(x')| - 1 \leqslant n - 1$, the number of steps is bounded by $V_n + (n-1)(V_n + 1) + 1$, which equals $n(V_n + 1)$. This completes the proof. □

Remark 2. The considerable looseness of the bound V_n is evident by the fact that it already gives an overestimate of the number of iterations required by CTS-Simple to perform an exhaustive search, even for small values of n. For example, $V_3 = 2(1+1) = 4$, whereas an upper bound of 3 is accurate. Another indication of the looseness of the bound is that the foregoing proof applies to the case where Time(x) is replaced by Time(x', x), though the latter can sometimes involve lengthier search processes. Note also that the form of assumption MA is not arbitrary. That is, it is easy to demonstrate that a search may fail to visit all of X if the rule is changed to select $x'' = \text{argmax}\{\text{Time}(x)\colon x \in N(x')\}$.

The bound implied by $V_{n+1} = n(V_n + 1)$ is more than $n!$. We now provide a more compact proof of the theorem that gives a better bound. Define $U_1 = 0$ and define $U_{n+1} = 2U_n + 1$, for $n \geqslant 1$. The bound implied by $U_{n+1} = 2U_n + 1$ may equivalently be expressed as $U_n = 2^n - 1$. The theorem holds for this definition of the upper bound U_n.

Theorem 2. *Beginning with any solution $x^* \in X$, the CTS-Simple method will visit every solution in X in at most U_n steps.*

Proof. The theorem evidently holds for $n = 1$ and 2 (and 3). By induction, assume the theorem is true for $|X| \leqslant n$, and consider $|X| = n + 1$. After U_n steps, starting from some solution $x^* \in X$, a set X^* containing x^* has been visited. Define $N^*(x) = N(x) \cap X^*$. During the U_n steps executed to generate X^*, each solution x' that is visited yields $\min\{\text{Time}(x): x \in X^*\} = \min\{\text{Time}(x): x \in X\}$. (Otherwise, since X contains X^*, a smaller min value would occur in $X - X^*$, and the method would visit a solution not in X^*, contrary to assumption.)

If $|X^*| < n$, then the inductive hypothesis says all of X^* was visited in at most U_{n-1} steps. Clearly, once X^* is visited,

$$\text{Min}\{\text{Time}(x): x \in X^*\} > \text{Max}\{\text{Time}(x): x \in X - X^*\}.$$

Continuing for another U_{n-1} steps, we already know the solutions visited remain entirely in X^* and that the choices are exactly as if restricting the neighborhood to N^*. Hence, the inductive hypothesis says we will revisit all of X^* again. By NA, at least one of these visited solutions has a solution $x'' \in X - X^*$ as a neighbor. Since $\text{Time}(x'') < \text{Time}(x)$ for all $x \in X^*$, the solution x'' will be visited at least by $2U_{n-1} + 1 = U_n$ steps, contrary to assumption. Thus, we conclude $|X^*| \geqslant n$. If $|X^*| = n + 1$, we are done, so suppose otherwise. Then $X - X^*$ contains exactly one element, which again we denote by x''. We continue the process for another U_n steps, and if x'' is not visited, by the same arguments as above we are assured to have visited all of X^* again. Likewise, as before, x'' must be a neighbor of some $x \in X^*$, and $\text{Time}(x'') < \text{Time}(x)$ for all $x \in X^*$. This insures that x, and hence all of X, will be visited by $2U_n + 1 = U_{n+1}$ steps, thereby completing the proof. □

To provide intuitive insight into the nature of "worst case" solution sequences that can be generated by CTS-Simple and to see how close the method can come to reaching the bound of Theorem 2, a class of examples with symmetric and asymmetric neighborhood structures is given in Appendix A.

2.2. CTS algorithm based on frequency-memory

Since frequency-based memory is also useful in TS, it is natural to speculate that a "frequency version" of Theorem 2 is valid. In fact, the preceding proof serves to establish the result. We apply the natural definition, Frequency(x) = the number of times x has been visited, and replace the method assumption MA by MA′ and the initialization assumption IA by IA′, which are defined as follows.

Initialization assumption (IA′). Given the starting solution x^* for the search, the values Frequency(x), $x \in X$, begin with Frequency(x) = 0 for all $x \in X$ except x^*, and Frequency(x^*) = 1.

Method assumption (MA′). From any current solution x', the search will visit a previously unvisited solution, $x'' \in N(x')$ if one exists, and otherwise will visit a solution $x'' = \text{argmin}\{\text{Frequency}(x)\colon x \in N(x')\}$.

As before, the bound is loose (though tighter than the previous one) and applies by replacing Time(x) or Frequency(x) by Time(x', x) or Frequency(x', x).

Corollary to Theorem 2. *The conclusion of Theorem 2 holds when CTS-Simple is based on frequency memory and the assumptions IS and MA are replaced by IA′ and MA′ as indicated.*

The proof of the Corollary is omitted. However, the following remark may be useful.

Remark 3. In the frequency-based version of CTS-Simple, if the solution x has been visited $\alpha|N(x)|+\beta$ times, with $0 \leqslant \beta < |N(x)|$ then all neighboring solutions of x have been visited at least α times and there exist β elements in $N(x)$ which have been visited at least $\alpha + 1$ times.

Illustrative examples of the frequency-based version of CTS-Simple also appear in Appendix A.

3. Acceleration of the Aspiration by Default Rule

In this section, we propose an approach for accelerating the Aspiration by Default rule, which may transform an exponential search into a much faster search that is polynomial (or perhaps even linear) in $|X|$. The modified version incorporates additional information to gain its benefits.

3.1. Generalized assumptions

Since the method assumption (MA) uses only "local" information, it is natural to generalize MA as follows. Define the distance $d(x, y)$ (or more precisely $d_N(x, y)$) associated with the neighborhood structure N, as the length of a shortest path connecting x and y, where length is measured as the number of arcs or edges in the path, according to whether the neighborhood is asymmetric or symmetric. Thus, the neighborhood $N(x)$ is extended by the disk $N^k(x)$ centered at node x with radius k which is the set of all nodes having distance at most k to node x, i.e.

$$N^k(x) = \{y \in X\colon d(x, y) \leqslant k\}.$$

A simple way to generalize the original version of MA using the Aspiration by Default rule is to consider all solutions in $N^k(x)$. This version is noted MA-k (where MA-1 is equivalent to the original version of MA).

Method assumption (MA-*k*). From any current solution x', the search will visit a neighboring solution, $x'' \in N(x')$, lying on a shortest path of length less than k that leads to an unvisited solution, if one exists. Otherwise, if all solutions in $N^k(x)$ are visited, the search will visit a neighboring solution, $x'' \in N(x')$, lying on a shortest path of length less than k that leads to a solution y' such that $y' = \text{argmin}\{\text{Time}(y): y \in N^k(x)\}$.

A specification of the procedure for the case $k = 2$ is described below.

Method assumption (MA-2). From any current solution x', the search will visit a solution, $x'' \in N(x')$, by using the following rule:

1. Let $x^1 = \text{argmin}\{\text{Time}(x): x \in N(x')\}$. Move to x^1 if it is an unvisited solution ($x'' = x^1$). Otherwise,
2. Let $x^2 = \text{argmin}\{\text{Time}(x): x \in (N^2(x') - N(x'))\}$ and let x be a neighbor solution of both x' and x^2, (i.e., x is one solution on the path between x' and x^2). If x^2 is an unvisited solution, move to the solution x ($x'' = x$). Otherwise,
3. If ($\text{Time}(x^1) < \text{Time}(x^2)$) then set $x'' = x^1$, else $x'' = x$.

In MA-2 the instruction $x^2 = \text{argmin}\{\text{Time}(x): x \in (N^2(x') - N(x'))\}$ can be replaced by $x^2 = \text{argmin}\{\min\{\text{Time}(y): y \in N(x) - \{x'\}\}: x \in N(x')\}$. An application of MA-2 is illustrated in Appendix B. The significant reduction in duplicate labeling is conspicuous, and becomes increasingly evident as the size of the problem grows.

3.2. A streamlined acceleration procedure

A potential limitation of the preceding acceleration approach is the amount of effort required to scan the set of alternatives available at various distances from the current solution. Enumeration of the possibilities even for solutions that lie only two moves away can be taxing, by approximately squaring the number of possibilities that lie in the immediate neighborhood (one move away).

This limitation is partially offset by the fact that the Aspiration by Default rule tends to require multiple visits to solutions, and hence larger numbers of steps, only in situations where the graph is relatively sparse and has special structure. (The examples given in Appendix A are clearly of this nature.) Denser graphs, with many connections between solutions, afford many options for entering and leaving any given solution, and thus pose a reduced likelihood that any particular node of the graph will be visited multiple times. As a result, the recourse to the neighborhood $N^k(x)$, at least for $k = 2$, is relevant primarily in application to sparse graphs, and is not as time consuming as would otherwise be the case. Even so, the effort can be greater than might be preferred.

We identify an alternative that approximates the options available for $k = 2$ with the same order of effort required to operate simply with the original neighborhood $N(x)$, thereby eliminating the "squared effort" effect. This alternative is based on the assumption that the degree of each node, i.e., the number of elements in $N(x)$, is known in advance or is easily determined at the point when x is visited. For example, in the case of binary solution vectors, where $N(x)$ consists of all binary solutions that

can be reached by changing a single component of x, the value $degree(x)$ of each node x is just the dimension of x itself. We also assume we are able to record an "updated" (modified) value for degree(x), as the search progresses. We do not concern ourselves with auxiliary data structures or dynamic list management strategies, such as those provided by the reverse elimination method (REM) of tabu search [2,4,5] in order to implement the following rules in neighborhood spaces, but continue to describe the operations directly in terms of the graph structure.

3.2.1. Accelerated procedure based on knowledge of degree(x)

1. The first time any given node x is reached during the search, set degree(x'):= degree(x') $-$ 1 for each node x' such that $x \in N(x')$.
2. If the choice of an unvisited neighbor is not possible (i.e., all neighbors of x have been visited), choose a neighbor x' with degree(x') > 0. If degree(x') $= 0$ for all neighbors, then choose x' by the usual Aspiration by Default rule.

When the foregoing procedure is applied to symmetric graphs, the update of the recorded node degrees can be modified by setting degree(x'):=degree(x') $-$ 1 for each neighbor $x' \in N(x)$.

This procedure achieves the same reduction in numbers of solutions visited as the method based on MA-2 in Section 3.1, while requiring substantially less effort. Since such an accelerated approach is primarily useful in connection with sparse graphs, the scan of all immediate neighbors in step 1 above can be performed without excessive work. An illustrative comparison of alternative strategies using the preceding ideas is given in Appendix C.

4. Tabu tree search

The "recency-based" memory commonly employed in tabu search, which is the basis for the Aspiration by Default rule, can also be applied with a slight change to provide a form of tree search. As observed in Glover [2], the use of staged decision rules in tabu search generates a standard form of tree search as a special case. However, in the present instance, the tree search that results is substantially different. By the variation subsequently described, for example, we obtain a *tabu tree search* that departs significantly from the customary branch-and-bound tree searches such as those used in popular methods for integer programming.

We continue to focus on the symmetric case unless otherwise specified, and label each solution x with a value Time(x) which indicates the "time" (iteration) at which it was visited. In contrast to our previous use of this label, however, we add the stipulation that as soon as Time(x) is assigned a value (i.e., as soon as x is visited), we do not permit its value to be further changed. (For simplicity, we do not increase the "time counter" except as each node is visited for the first time, so that the number of labels generated is at most $|X|$.) Accompanying this, we now reverse the Aspiration

by Default rule, to require that, whenever all elements of $N(x)$ have previously been visited, the method moves from x to the node $x' \in N(x)$ that has the *largest* (rather than smallest) value of Time(x'), subject to the limitation that this value must be smaller than that of Time(x) itself.

The resulting method is as follows.

Tabu tree search (TTS)

1. From a given solution x, move to an unvisited neighbor $x' \in N(x)$ whenever possible (i.e., a neighbor for which Time(x') is not yet determined), and stop if the label thus assigned to x' is Time(x') $= |X|$. Otherwise,
2. Move to the visited neighbor x' with the largest value of Time(x') that is less than Time(x).

We establish the relevant properties of the method as follows, under the assumption that the graph of the neighborhood space is connected.

Theorem 3. *The TTS method generates a tree, rooted at the initial solution, that spans the nodes of the neighborhood graph. Each edge of the tree is crossed exactly once in the direction away from the root, and at most once in the direction toward the root.* (*No edges outside of the tree are crossed*). *In addition*:

(a) *The unique path from any solution to the root is generated by repeatedly executing the rule of step* 2 *of the TTS method.*

(b) *Each time any solution x is visited, each labeled neighbor x' of x is either an ancestor or descendant of x in the tree currently constructed* (*i.e., either x' lies on the path from the root to x, or else x lies on the path from the root to x'*).

(c) *Each time step* 2 *is executed to reach a visited node x', all nodes of the graph that are neighbors of visited nodes x'', where Time(x'') > Time(x'), are also visited nodes.*

(d) *Each time step* 1 *successfully identifies an unvisited neighbor of x, then node x satisfies the condition $x = Argmax\{Time(y)$: y is a node of the current tree and y has an unvisited neighbor$\}$.*

Proof. We establish the theorem inductively. Except for the claim that the tree spans the graph, each of the assertions of the theorem is clearly true on all steps until and including the first time that step 2 is executed. At this point the subgraph generated is a simply path from the root, and step 2 is executed because the node x at the end of this path has no unvisited neighbors. (We suppose not all nodes are yet reached, or else the proof is complete.) Furthermore, these assertions remain true if step 2 is immediately executed again, and remains true throughout all subsequent executions of step 2 until step 1 is finally executed. By connectivity, if any unvisited node of the graph exists, it must be a neighbor of at least one node previously visited, and the assertion (c) implies we will identify a node of the present tree with access to an unvisited node. Let x^* denote the node x' reached on this execution of step 2, where x^* also becomes the node x at the following execution of step 1. Then it

is clear that x^* qualifies as the particular node x of the current tree that satisfies assertion (d).

Given these relationships established to this point, the argument now follows inductively, since we may repeat the same observations relative to the path now generated from the root through x^*, until finally reaching a stage where step 2 must again be executed, proceeding through the identification of a new x^*. The fact that the assertions are maintained at each earlier step of the construction, and are augmented repetitively for the path through each new x^*, assures the assertions will continue to hold, and ultimately that the tree must become a spanning tree. □

The theorem immediately permits the following observation.

Remark 4. The values assigned to the labels Time(x) can alternately be changed so that, instead of increasing each time a new solution x' is visited in step 1, Time(x'):= Time(x) + 1. Then the rule for step 2 identifies x' to be the solution that yields Time(x') = Time(x) − 1. The stopping criterion is changed to stop the process as soon as all solutions in X are visited.

Note that in the previous TTS procedure the labels Time(x) can be interpreted as the order of visiting the solution x. With the alternative change proposed in Remark 4, the label Time(x) is equal to the length of the path from the root (initial solution) to the solution x plus one.

By the labeling of Remark 4, multiple solutions can receive the same label Time(x). Theorem 3 implies, however, that the solution identified by Time(x') = Time(x) − 1 in step 2 is nevertheless uniquely determined. It also implies that no neighbors of a given solution can have the same label.

An obvious extension of the approach, which adds at most one solution to the neighbors of any given solution, occurs as follows.

Remark 5. The TTS method can be applied to asymmetric graphs if step 1 is modified so that $N(x')$, for the solution x' selected to be reached by a move from x, is allowed to be augmented to include the solution x, if x is not already in $N(x')$.

4.1. TTS and flexibility of choice

In common with the Aspiration by Default rule, the TTS approach in some cases may visit all solutions by only visiting each solution a single time, hence effectively generating a Hamiltonian path through the neighborhood space, in contrast to the type of trajectory created by usual forms of tree search. However, more importantly, the TTS approach allows substantially greater flexibility of choice than customary types of tree search, as embodied in branch-and-bound approaches. We illustrate this as follows.

Example: *n*-dimensional binary vectors

Consider the set X of four-dimensional binary vectors. A standard *backtracking* (depth first) branch-and-bound approach, where the symbol "∗" denotes an unassigned

value, generates a sequence such as the following.

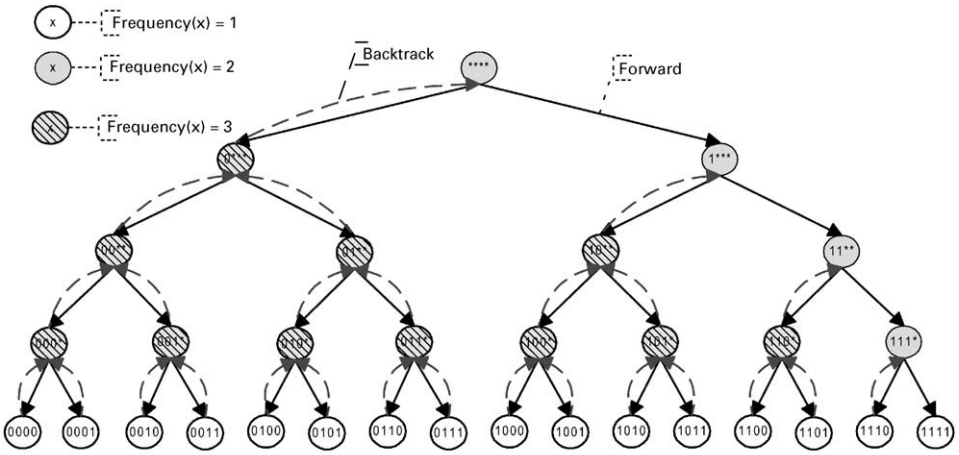

Figure 1(a) : A standard "backtracking" (depth first) branch-and-bound approach.

By contrast, the TTS approach can create a very different set of solutions. A set of choices for this example (purposely designed to backtrack as early and as often as possible) yields the following sequence:

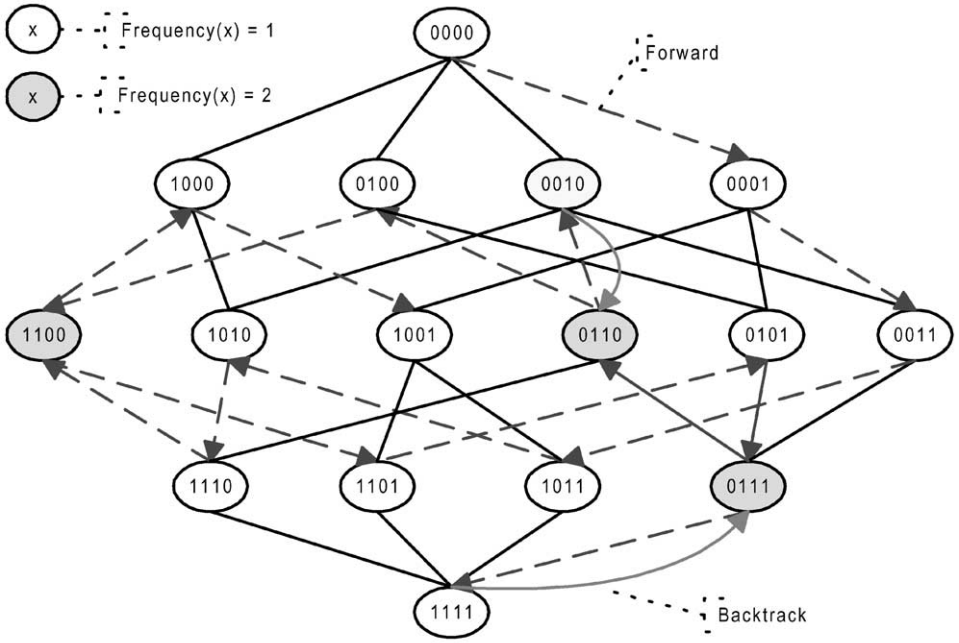

Figure 1(b) : TTS approach.

It is also appropriate to keep in mind that the TTS approach can produce different outcomes depending on the neighborhood structure selected. The preceding illustration

relies on a neighborhood that changes the value of a single variable at a time. Different forms of search, and different types of "tree structures", are created for different neighborhoods, such as those that allow the value of 2 or more variables to be changed simultaneously.

4.2. Contrasts between TTS and branch and bound

The foregoing example shows not only that the TTS approach generates different solutions, but that the number of backtracking steps is much smaller than in the branch-and-bound procedure. In essence, the method runs considerably "deeper" than branch-and-bound search before encountering a situation where it is necessary to reverse its trajectory. On the other hand, customary branch and bound never repeats a solution, as a result of structuring the tree according to the use of unassigned values.

Every depth first branch-and-bound search always follows exactly the pattern illustrated in Figs. 1a and 1b except that a variable may first branch to 1 rather than to 0, and the choice of the variable to branch on (i.e., implicitly, the indexing of the variables) may be changed on forward steps. Variants that generate the branch-and-bound tree by a different sequence than the depth first rule (such as a *best bound* rule), can change the order of steps in which branches are created, but still produce the same tree (disregarding fathoming possibilities that may exclude certain branches).

On the other hand, the TTS structure differs according to the choices made — that is, different choices may produce different numbers of revisited solutions (and, as previously remarked, some may produce no revisited solutions), thus producing trees of different topologies.

4.3. Enhanced TTS procedures for graph search

An enhancement of TTS is possible for graph search by maintaining and updating a record of degree(x), under the same assumptions previously described for maintaining such a record in applying the Aspiration by Default rule. However, the manner in which degree(x) is used differs from the earlier proposal. As an enhancement of TTS, the reliance on degree(x) does not have the purpose of reducing the number of times that particular solutions are visited, but rather of reducing the number of operations that are devoted to scanning neighbors of solutions visited. This second type of reduction can produce a significant savings in computational effort, particularly in graphs of moderately high density. The ability to enhance the TTS approach in this way results from the fact that the tree predecessor of a given solution remains invariant throughout the search, and thus the identity of this predecessor can be saved by recording a single additional item of information for each solution visited. (The complete solution need not be recorded, as long as sufficient information is retained to recover the solution directly from its neighbor.) The process is as follows.

4.3.1. Enhanced TTS method

1. The first time node x is reached by applying the TTS method, set degree(x'):= degree(x') $-$ 1 for each neighbor $x' \in N(x)$. In addition, record predecessor(x):=x', for the particular neighbor x' such that x has been reached (first) by the move from x' to x.
2. Whenever a node x is visited after the first time (i.e., x already has been assigned a predecessor), check whether degree(x) = 0. If so, immediately execute step 2 of the TTS method, identifying the solution x' selected at this step to be x':=predecessor(x).

Remark 6. Instruction 2 of the enhanced TTS method always occurs upon executing step 2 of the original TTS method, since this step is the one that leads to a previously visited node. Hence, the condition degree(x) = 0 causes step 2 to be executed again (and setting x':=predecessor(x) avoids examining the neighbors of x).

The search process is accelerated by avoiding the examination neighbors of x, as indicated in Remark 6. Clearly, the larger the number of solutions that are visited before backtracking, the greater the opportunity to save effort by this approach. If the search traces a Hamiltonian path, for example, then the approach would eliminate the examination of neighbors for every node, for a saving of effort roughly equal to twice the total number of edges in the graph. In general, although it may be rare for the search to follow a Hamiltonian path, the fact that a TTS approach typically goes very deep relative to the starting (root) node, implies that the method is likely to yield degree(x) = 0 for a considerable number of nodes encountered at earlier depths of the tree, as a result of visiting their neighbors as descendants later in the search. Graphs with hub-and-spoke structures, where collections of nodes can reach each other only by paths that cross one or a small number of edges contained in a "hub" about the root, will tend to result in setting degree(x) = 0 for a substantial number of nodes in each collection.

A further enhancement is possible by the device of recording the predecessor as a complete solution, which is then "passed along" to provide a new neighbor for other solutions. (In the graph setting, a particular node thus becomes accessed as a neighbor of other nodes by such a passing operation.) This gives rise to an opportunity to create a *reverse jump*, which bypasses a number of backtracking steps, in cases where the search generates degree(x) = 1 for a string of solutions successively encountered.

4.3.2. Reverse jump TTS

1. Whenever a node x is visited by the enhanced TTS approach, and degree(x) = 1, identify the unique unvisited neighbor x' of x, and pass forward the node predecessor(x) by assigning predecessor(x'):=predecessor(x) (instead of predecessor(x') = x) when x' is visited.

2. At each execution of step 2 of the enhanced TTS approach, if degree(x) = 0, then the assignment x':=predecessor(x) creates a "reverse jump" to the earliest predecessor in a string generated by step 1.

By the preceding reverse jump procedure, the backtracking process can avoid intermediate steps that otherwise would require lengthy calculation. Such a variation of the enhanced TTS approach is likely to be useful for graphs that have "long and skinny" appendages. It can also be useful in situations where the search progresses from a set of nodes N' to a set N'', where for each $x \in N''$, all but one (or a small number) of neighbors of x lie in N'.

Remark 7. The reverse jump tabu tree search can be deduced from the enhanced tabu tree search method by only changing the step 2 as follows: while degree(x) = 0 do x = predecessor(x).

Appendix C gives an illustrative comparison of alternative enhanced strategies for TTS.

4.4. Novelty of the TTS method

In spite of the illustrated differences between branch and bound and TTS for moving through the search space, the TTS approach involves no fundamentally new ideas for achieving a finite search — in contrast to a TS approach based on using the Aspiration by Default rule. In terms of a graph search, the TTS approach is an entirely straightforward form of tree search, which follows a depth first design.

There is a misconception in portions of the search literature (often fostered by textbooks in artificial intelligence), that all depth first methods are essentially the same. We have already noted the marked contrast between TTS and customary branch-and-bound procedures (both depth first and otherwise), and the implications of this contrast for the mechanisms that are available for generating an effective search. One of the most important differences, is the freedom of choice offered by the TTS approach. The greater flexibility to choose values assigned to variables, without having to interrupt the search by backtracking to earlier (incomplete) solutions, supports the goal of exploiting tailored heuristics to guide the search.

The relevance of this design difference can be illustrated by comparing TTS to another type of depth first tree search, called *reverse search* [6–8]. Reverse search is a significant form of depth first search in applications such as enumerating vertices of polyhedra. Nevertheless, it restricts the available decisions even more rigidly than branch and bound, and shares with branch and bound the characteristic of penetrating only to very limited depths before encountering the necessity of backtracking. The enhancements we have identified for applying TTS to graph searches in Section 4.4 have no counterparts (and in fact no meaningful interpretation) in the contexts of both reverse search and branch and bound.

Apart from such distinctions among different forms of tree search, a primary novelty of the TTS approach stems from its ability to be coupled with the REM memory

procedure developed for tabu search. The illustration of enumerating 0–1 vectors in Section 4.1, which compares TTS with branch and bound, makes the importance of this connection clear. Evidently, whenever flexible choice rules are used (as implicitly occurs for TTS in this illustration), it is not a trivial matter to identify which binary solutions are currently available to be visited at each step, nor to identify when backtracking becomes necessary. The REM procedure handles these challenges automatically, thus making it possible to apply the TTS method in the context of neighborhood search without ambiguity. An analysis of relevant considerations, based on a special *channeling* concept, is given in Appendix D.

5. Comparisons with other approaches

As a basis of comparison, it is interesting to briefly consider other proposals for graph searches. One of the earliest, which has an elegant statement and justification, is the Tarry Traverse [9]. (An illuminating exposition of this method can be found in Thompson [10].) In contrast to the approaches described here, the Tarry Traverse utilizes a memory structure that attaches labels to edges rather than nodes, and crosses each edge twice, once in each direction. Since the total number of edges in the graph can be significantly larger than the number of edges in a tree, the amount of effort (and memory) in such a traverse is evidently somewhat greater than in the TTS approach. Charnes and Cooper [11] have remarked that the Tarry Traverse may be used as a basis for enumerating the extreme points of a linear program. Clearly, as our discussions show, it is possible to do better.

Another approach worth noting is the reverse search method, briefly alluded to earlier, which can be applied to exhaustively visit the nodes of a graph. Reverse search may be viewed as a class of methods, whose members vary by relying upon different evaluation functions that satisfy particular properties. Going beyond these methods, there exists a broad class of procedures that combine various characteristics of reverse search with complementary characteristics of branch and bound, to produce searches with useful properties of memory economy and flexibility [12]. However, the degree of flexibility represented by these approaches is still markedly less than that afforded by the TTS design. In this connection, we conjecture that forms of TS based on the Aspiration by Default rule allow access to a greater variety of search paths than TTS, with the potential disadvantage that they also admit a larger number of solutions to be revisited. An analysis of relevant considerations, based on a special *channeling* concept, is given in Appendix D.

6. Practical considerations

From a theoretical point of view, a finite convergence result is “infinitely better” than an infinite convergence result. For example, the popular *convergence in*

probability result of simulated annealing does not assure that an optimal solution will be found the first time in any finite number of steps: for the purpose of finding such a solution the first time, the method offers no advantages over relying on blind randomization. On the other hand, the magnitude of "finite" in a finite method can still be large, and the primary relevance of a finiteness result depends on providing a structure that can embrace useful heuristic features. We emphasize the ability to apply the finiteness results for tabu search processes in a way that allows significant latitude for implementing associated strategic processes.

By contrast, most of the search literature places great significance on theoretical foundations involving forms of convergence that are conspicuously not finite. Reversion to an infinite guarantee—i.e., one that provides no assurances about convergence in finitely bounded time—would be justified if it allowed a wider range of strategic considerations to be embraced. Yet, ironically, the rationale for these alternative theoretical developments has nothing to do with enlarging the range of strategic choice. Rather, by basing the control mechanisms on randomization, the rationale for the search becomes farther removed from considerations of strategy. There may be fascination in the pin of a roulette wheel, but resorting to such a mechanism in combinatorial search carries the price of abandoning a quest for finiteness.

In summary, the key observations of this paper are: (1) strategic flexibility is compatible with assured finite convergence, by special forms of memory introduced in certain forms of tabu search; (2) the resulting search traverses the nodes of a graph in a significantly different way than provided by tree search; (3) a simple tree search variant of the approach produces a type of tree search that offers novel contrasts with branch and bound, and also differs notably from other tree searches such as reverse search and the Tarry Traverse.

Appendix A. Exponential and quadratic paths

In this appendix, we illustrate the behavior of the two versions of CTS-Simple (recency-based and frequency, respectively) on three classes of examples with symmetric and asymmetric neighborhood structures. The two versions of CTS-Simple applied to the first asymmetric example generate an exponential path, which shows the tightness of the bound provided in Theorem 2.

For each example in the following, we start at node 1, and use the *least node index* rule for breaking ties when $\text{Time}(x) = 0$ or $\text{Frequency}(x) = 0$, finally stopping when reaching node n. The sequence of labels for each node and the path generated are given.

Example 1: Exponential path in an asymmetric graph

We construct a digraph $G_n = \langle X, A \rangle$, where n is an even number ($n = 2p$), as follows:

- $X = \{1, 2, \ldots, n\}$.
- $A = \{(1,2),(n,1)\} \cup \{(2k, 2(k+1)), (2k, 2k+1), (2k+1, 1)$: for $k = 1, 2, \ldots, (n-2)/2\}$.

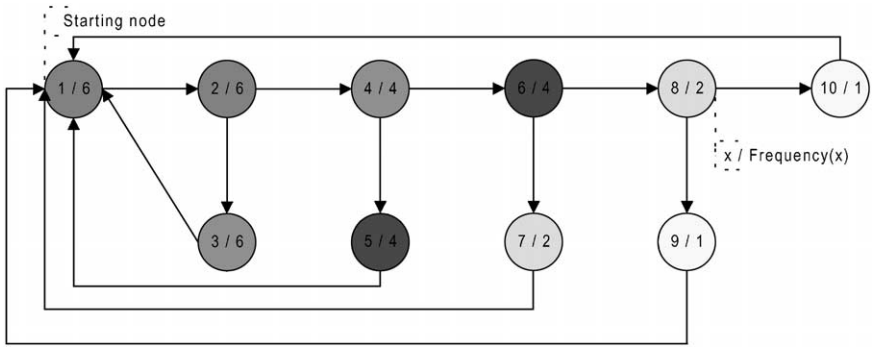

Fig. 1.

Table 1

Node	Labels (visiting time of the node)															
1	1	4	8	11	16	19	23	26	32	35	39	42	47	50	54	57
2	2	5	9	12	17	20	24	27	33	36	40	43	48	51	55	58
3	3	10	18	25	34	41	49	56								
4	6	13	21	28	37	44	52	59								
5	7	22	38	53												
6	14	29	45	60												
7	15	46														
8	30	61														
9	31															
10	62															

Thus the graph G_n has n nodes and $(3n-1)/2$ arcs. For example, the graph G_{10} $(n=10)$ is shown in Fig. 1. Node 1 forms the base of the graph. Besides, node 1 appear to parallel lines of nodes. The nodes in the line directly beside node 1 are numbered $2,4,6,8,10$ and the nodes in the adjacent line, just above the first line, are numbered $3,5,7,9$.

Path generating rule: Start at node 1, using the Aspiration by Default rule ($\min\{\text{Time}\}$). Whenever there is a tied choice (because the path is presented with two choices that both have not yet been visited ($\text{Time}(x)=0$), choose the node with the smaller index. Assume that the vector Time has been initialized as follows:

$$\text{Time}(k) = -n+k-1; \quad \text{for } k=1,\ldots,n.$$

Table 1 shows nodes and their associated labels which are generated by using an instance of the graph G_n with $n=10$. Thus the path first goes from 1 to 2, and upon reaching 2 (where both 3 and 4 are not yet visited), next visits 3. Thereafter, the path follows the smallest of the previous $\text{Time}(x)$ labels, until again reaching a point where a choice must be made. By the path trace, the method finally reaches node 10 (i.e., node n) at step $62(=2^{(n+2)/2}-2$, with $n=10)$.

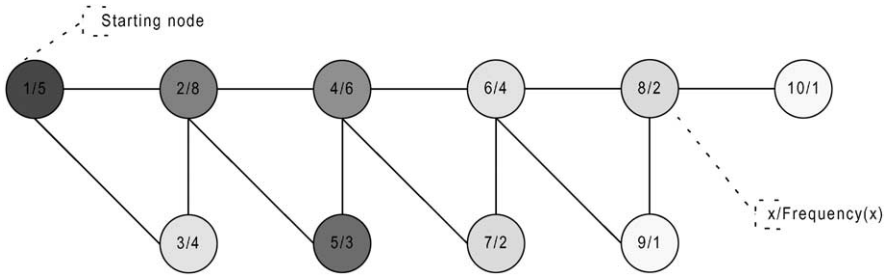

Fig. 2.

Table 2

Node	Labels (visiting time of the node)							
1	1	4	10	19	31			
2	2	5	8	11	17	20	29	32
3	3	9	18	30				
4	6	12	15	21	27	33		
5	7	16	28					
6	13	22	25	34				
7	14	26						
8	23	35						
9	24							
10	36							

In the general case, when n is an even number ($n=2p$), it is easy to observe that the frequency of the even nodes and odd nodes is the same. Precisely, we have

$$\text{Frequency}(2k-1)=\text{Frequency}(2k)=2^{(n-2k)/2} \quad \text{for } k=1,\ldots,n/2.$$

Hence, by summing all the frequencies, the value of V_n is equal to $2^{(n+2)/2}-2$.

Example 2: Quadratic path in a symmetric graph

Consider a graph of undirected edges, whose structure is similar to that of Example 1, except that the arcs $(2k+1,1)$ that connect back to node 1 are replaced by edges $(2k+1,2(k-1))$. The "right column" turns into a "ladder" (or a "saw tooth" structure). Specifically, the graph $G_n=\langle X,A\rangle$, has the following form:

- $X=\{1,2,\ldots,n\}$.
- $A=\{(1,2);(1,3)\}\cup\{(2k,2k+1);(2k,2k+2);(2k,2k+3)$: for $k=1,2,\ldots,(n-2)/2\}$

For example, the graph G_{10} is shown in Fig. 2.

Path generation rule: Exactly the same as in Example 1.

Hence the sequence for $n=10$ is:

1, 2, 3, 1, 2, 4, 5, 2, 3, 1, 2, 4, 6, 7, 4, 5, 2, 3, 1, 2, 4, 6, 8, 9, 6, 7, 4, 5, 2, 3, 1, 2, 4, 6, 8, 10

(Table 2).

In the general case, when n is an even number ($n=2p$), the frequency of even nodes is: $\text{Frequency}(2k)=n-2k$, for $k=1,\ldots,(n-2)/2$; and the frequency of odd nodes is: $\text{Frequency}(2k+1)=(n-2k+2)/2$, for $k=1,\ldots,(n-2)/2$, and the frequency

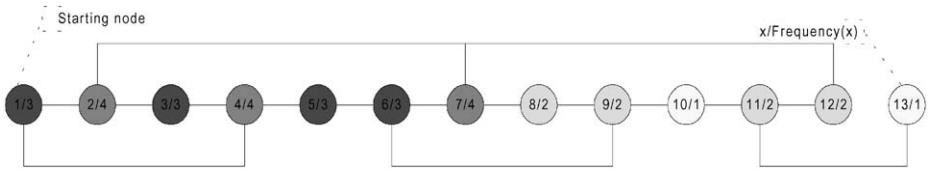

Fig. 3.

of the end node is equal to 1 (Frequency$(n)=1$). Thus by summing all frequencies, the number of steps as a function of n is $V_n=(3n^2-2n+8)/2$.

Example 3: Quadratic path in a symmetric graph

Construct a digraph $G_n=(X,A)$, where $n=5p+3$, as follows:

- $X=\{1,2,\ldots,n\}$.
- $A=\{(k,k+1)$: for $k=1,\ldots,n-2\}\cup\{(5k+1,5k+4);(5k+2,5k+7)$: for $k=0,1,\ldots,p-1\}\cup\{(n-2,n)\}$.

For example, the graph G_{13} ($p=2$) is shown in Fig. 3.

Path generation rule: Start with node 1, and visit the unvisited nodes in the sequence 1 to n. Then apply the Aspiration by Default (min(Time(x)) rule. (No tie breaking rule is needed, except as implicit in the beginning sequence from 1 to N.) We assume that the vector Time has been initialized as follows:

$$\text{Time}(k)=-n+k-1; \quad \text{for } k=1,\ldots,n.$$

The following results (nodes and their associated labels) are generated by using an instance of the graph G with $n=13$. The first column in Table 3 indicates the node number. The second column shows, for each node, the Time(x) values that the node receives each time it is visited. The third column indicates the number of times (frequency) each node has been visited at the end of the process. Hence the sequence for $n=13$ is:

$$1,2,3,4,5,6,7,8,9,10,11,12,7,2,1,4,3,2,7,6,5,4,1,2,3,4,5,6,9,8,7,12,11,13.$$

By the path trace, the method finally reaches node 11 (i.e., node $n-2$) a second time at step 33. Since we suppose this node connects to the final unvisited node $n=13$, the process ends at step 34. Thus, the method visits all nodes after 34 steps ($V_{13}=34$).

In the general case, when $n=5p+3$, the frequency of nodes is:

Frequency$(1)=p+1$; Frequency$(5k+1)=2(p-k+1)-1$ for $k=1,\ldots,p-1$; Frequency$(5p+1)=2$,

Frequency$(2)=p+2$; Frequency$(5k+2)=p-k+3$ for $k=1,\ldots,p-1$; Frequency $(5p+2)=2$,

Frequency$(5k+3)=p-k+1$ for $k=0,\ldots,p$,

Frequency$(5k+4)=2(p-k)$ and Frequency$(5k+5)=2(p-k)-1$ for $k=0,\ldots,p-1$.

Hence, the number of steps as function of the parameter p is given by $V_n=4p^2+7p+4$, where $n=5p+3$. In terms of the number of nodes n, this translates into $V_n=(4n^2+11n+31)/25$.

Table 3

Node(x)	Time (x)		Frequency(x)	
Example 2				
1	1	4	2	
2	2	5	2	
3	3		1	
4	6	8	2	
5	7		1	
6	9	11	2	
7	10		1	
8	12	14	2	
9	13		1	
10	15		1	
Example 3				
1	1	15	19	3
2	2	14	18	3
3	3	17		2
4	4	16	20	3
5	5	21		2
6	6	22		2
7	7	13	25	3
8	8	24		2
9	9	23		2
10	10			1
11	11	27		2
12	12	26		2
13	28			1

Below we give the results obtained by applying the CTS-Simple algorithm based on frequency-memory to the three preceding examples. For Example 1, this algorithm generates the same sequence described in Table 1, as the one based on recency-memory. The results obtained for Examples 2 and 3 are described in the following tables.

As shown numerically in those examples, the number of visited solutions with CTS based on frequency-memory is smaller than the one obtained by CTS-Simple based on recency-memory, specially for symmetric graphs (Examples 2 and 3).

We give below another example for the asymmetric case, where the bound is polynomial. In this case, the neighborhood graph $G = \langle X, A \rangle$ is defined by $X = \{1, 2, \ldots, n\}$ and $A = \{(1, k); (k, k-1)$: for $k = 2, \ldots, n\}$. The initialization step: Let $\text{Time}(x) = -x$, for $x \in X$ and start the search with the initial solution $x^* = 1$. It is easy to see that $V_n = n(n-1)/2 + 1$, for $n \geqslant 2$.

Appendix B. Illustration of improvement using the accelerated Aspiration by Default rule

The effect of applying the accelerated Aspiration by Default rule is demonstrated by the following tables, which show the label values for the nodes in the three examples of Appendix A, that result by using MA-2 (Table 4).

Table 4

	Example 1									Example 2			Example 3		
x	Labels (visiting time of the node)								Freq	Labels		Freq	Labels		Freq
1	1	4	8	13	16	22	26	29	8	1		1	1		1
2	2	5	9	14	17	23	27	30	8	2	4	2	2		1
3	3	15	28						3	3		1	3		1
4	6	10	18	24	31				5	5	7	2	4		1
5	7	25							2	6		1	5		1
6	11	19	32						3	8	10	2	6		1
7	12								1	9		1	7		1
8	20	33							2	11	13	2	8		1
9	21								1	12		1	9		1
10	34								1	14		1	10		1
11													11	13	2
12													12		1
13													14		1

Note that the gain is appreciable, especially for large problems. In Example 3 with $n = 53$, the bound obtained by using MA-1 is equal to 474 using MA-2 it is equal to 54. A suitable value of a parameter k depends on the neighborhood structure N and the size of the problem. (For example, the value $\max\{|N(x)|: x \in X\}$ is a parameter that may be used to control k.)

Evidently, the number of steps needed to explore all the solutions by using MA-k, depends on the initial solution chosen. The following table shows an instance of Example 3 with $n = 53$. The initial solution x^0 has been varied from 1 to n, and we have compared the bounds generated by the two methods MA-1 and MA-2 (Fig. 4).

Appendix C. Numerical experiments comparing alternative strategies

We have implemented the different strategies for exploring all nodes of a given connected graph discussed in Sections 3 and 4, using the C programming language, and with testing performed on a PC Pentium 300. The following Table 5 gives the names of the codes compared in our experiments.

We represent the input graph used in our implementation as an adjacency list. A symmetric graph of n nodes is represented by n adjacency lists, one for each node. The adjacency list for a node x is a list of all nodes y successors of x. For some algorithms (particularly these described in Section 3), an asymmetric graph is represented by the set of its nodes and two lists are associated with each node x, one containing the predecessors, the other the successors of x.

To study the performance and analyze the algorithms, six families of graphs have been chosen. Table 6 summarizes these families of graphs whose size depends on a parameter we have denoted by p.

In our implementation, for each graph we start at node 1, and use the "least node index" rule for breaking ties. We stop when all nodes are visited.

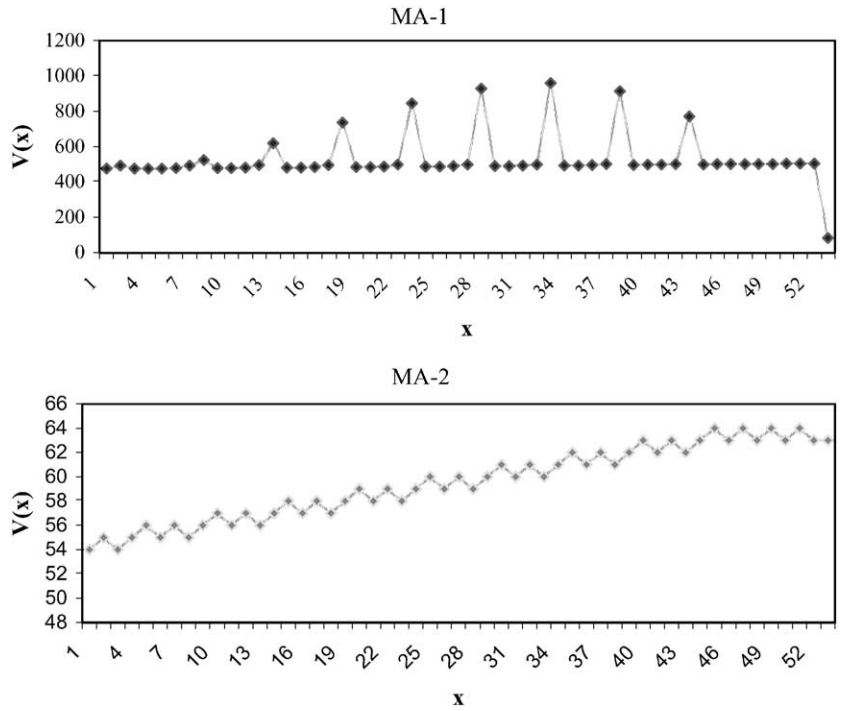

Fig. 4. Influence of initial solution (Example 3 with $n = 53$).

Table 5

Code name	Algorithm
CTS-R	Convergent tabu search algorithm based on recency-memory
CTS-F	Convergent tabu search algorithm based on frequency-memory
CTS-R-2	Acceleration of the convergent tabu search algorithm based on recency-memory
CTS-F-2	Acceleration of the convergent tabu search algorithm based on recency-memory
ACTS-R-2	Approximate CTS-2 procedure based on knowledge of degree and frequency-memory
ACTS-F-2	Approximate CTS-2 procedure based on knowledge of degree and frequency-memory
TTS	Tabu tree search
E-TTS	Enhanced tabu tree search method
RJ-TTS	Reverse jump tabu tree search method

In order to compare the results obtained by the accelerated procedure based on knowledge of degree(x) (ACTS-R-2 and ACTS-F-2), the strategy in the process of selecting the next solution has been tested as follows. Step 1 chooses the first unvisited solution encountered, and step 2 chooses the first neighbor solution x' with degree(x') > 0.

Table 6

Class name	Brief description	Size $n=$
G_1	Exponential path in an asymmetric graph	$2p$
G_2	Exponential path in an asymmetric graph	$3p-1$
G_3	Quadratic path in a symmetric graph	$2p$
G_4	Quadratic path in a symmetric graph	$5p+3$
G_5	n-dimensional binary vectors	2^p
G_6	Tree binary graph	2^p-1

Table 7
Comparison of different strategies

	n	CTS-R	CTS-R-2	ACTS-R-2
G_1	$2p$	$2^{(n+2)/2}-2$	$(n^3+6n^2-16n+192)/48$	$2^{n/2}+n/2$
G_2	$3p-1$	$5(2^{(n-2)/3})-3$	$(n^3+21n^2-78n-62)/162$	$5(2^{(n-5)/3})+(n+1)/3$
G_3	$2p$	$(3n^2-2n+8)/2$	$(3n-2)/2$	$(3n-2)/2$
G_4	$5p+3$	$(4n^2+11n+31)/25$	$n+1$	$n+1$
G_5	2^p			n
G_6	$2^{p+1}-1$	$2n-\log_2(n+1)$	$2n-\log_2(n+1)$	$2n-\log_2(n+1)$

	n	CTS-F	CTS-F-2	ACTS-F-2
G_1	$2p$	$2^{(n+2)/2}-2$	$3(2^{(n-2)/2})-1$	$3(2^{(n-2)/2})-1$
G_2	$3p-1$	$5(2^{(n-2)/3})-3$	$15(2^{(n-8)/3})-2$	$15(2^{(n-8)/3})-2$
G_3	$2p$	$3n/2$	$(3n-2)/2$	$(3n-2)/2$
G_4	$5p+3$		$n+1$	$n+1$
G_5	2^p	n	n	n
G_6	$2^{p+1}-1$	$(5n-2\log_2(n+1)-7)/2$	$(9n-4\log_2(n+1)-15)/4$	$(5n-2\log_2(n+1)-7)/2$

	n	TTS	E-TTS	RJ-TTS
G_3	$2p$	$(3n-2)/2$	$(3n-2)/2$	$(3n-2)/2$
G_4	$5p+3$	$n+1$	$n+1$	$n+1$
G_5	2^p	n	n	n
G_6	$2^{p+1}-1$	$2n-\log_2(n+1)$	$[(10n-3\log_2(n+1)+1)/6]$	$(3n-1)/2$

For each run of a given code, noted M, two measures are reported:

- V(M): the number of steps needed for visiting all nodes of a given graph; and
- CPU(M): the running time measured in CPU seconds required for visiting all nodes of the graph, excluding the input times (CPU time for generating the graph) and output times (the output of the results).

Under the platform configuration used, a size $n \geqslant 200,000$ of the input graph creates problems of memory allocation or out-of-range floating-point values.

The following Table 7 gives the number of steps required for visiting all solutions in different graphs.

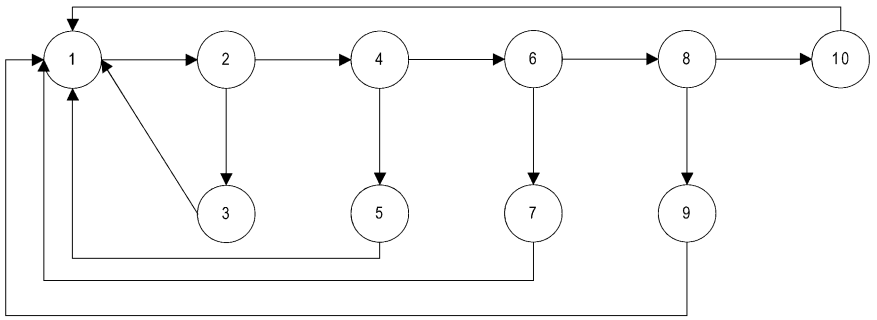

Fig. 5. Asymmetric graph G_1 with $n = 10$.

Example 1: Exponential path in an asymmetric graph (G_1)

We construct a digraph $G_n = \langle X, A \rangle$, where $n = |X| = 2p$, as follows:

- $X = \{1, 2, \ldots, n\}$.
- $A = \{(1,2), (2p, 1)\} \cup \{(2k, 2(k+1)), (2k, 2k+1), (2k+1, 1)$: for $k = 1, 2, \ldots, (n-2)/2\}$.

Thus the graph G_n has n nodes and $(3n - 4)/2$ arcs. For example, the graph G_{10} ($p = 5$) is shown in Fig. 5.

Tables 8(a) and (b) present results of experiments on this family of graphs G_1. These results show that CTS-R-2 outperforms all other algorithms. The number of steps done by CTS-R and CTS-F are exactly the same. This is also the case for the CTS-F-2 and ACTS-F-2 methods. However, the CPU time of CTS-F is slightly greater than the one with the CTS-R algorithm, but the running times of the ACTS-F-2 method decrease by roughly a factor of two compared with those of CTS-F-2. Although the performance of ACTS-R-2 is not good compared with that of CTS-R-2, this method remains interesting because the number of steps required to visit all solutions and the CPU time is less than with CTS-R. Comparing CTS-F and CTS-F-2, we observe that the number of steps required for CTS-F-2 is divided by a factor of two. The disadvantage is that the CPU times are multiplied by more than two.

The fastest code for this problem family is CTS-R-2. Indeed, for the family of graphs G_1, all algorithms required an exponential number of steps for visiting all nodes except CTS-R-2 which has required a polynomial number of steps. These experiments confirm the accuracy of our estimates for the number of steps needed to scan all nodes of a graph of type G_1 ($n = 2p$) for CTS-R and CTS-R-2 ($2^{(n+2)/2} - 2$ and $(n^3 + 6n^2 - 16n + 192)/48$, respectively).

Example 2: Exponential path in an asymmetric graph (G_2)

We construct a digraph $G_{3p-1} = \langle X, A \rangle$, where $n = 3p - 1$, as follows:

- $X = \{1, 2, \ldots, n\}$.
- $A = \{(1,2); (n,1)\} \cup \{(3k - 1, 3k + 2); (3k - 1, 3k); (3k, 3k + 1); (3k + 1, 1)$: for $k = 1, 2, \ldots, (n + 4)/3\}$.

An instance of this digraph G_{14} ($p = 5$) is given in Fig. 6.

Table 8

n	CTS-R	CTS-F	CTS-R-2	CTS-F-2	ACTS-R-2	ACTS-F-2
(a) *Number of steps with asymmetric graph* G_1						
10	62	62	34	47	37	47
20	2046	2046	214	1535	1034	1535
30	65,534	65,534	669	49,151	32,783	49,151
40	2,097,150	2,097,150	1524	1,572,863	1,048,596	1,572,863
50	67,108,862	67,108,862	2904	50,331,647	33,554,457	50,331,647
60	2,147,483,646	2,147,483,646	4934	1,610,612,735	1,073,741,854	1,610,612,735

n	p	CTS-R	CTS-F	CTS-R-2	CTS-F-2	ACTS-R-2	ACTS-F-2
(b) *Computing time with asymmetric graph* G_1							
10	5	0.00	0.00	0.00	0.00	0.00	0.00
20	10	0.00	0.00	0.00	0.00	0.00	0.00
30	15	0.02	0.01	0.00	0.03	0.01	0.02
32	16	0.03	0.04	0.00	0.08	0.02	0.04
34	17	0.06	0.06	0.00	0.14	0.05	0.06
36	18	0.12	0.13	0.00	0.30	0.09	0.14
38	19	0.24	0.25	0.00	0.58	0.17	0.28
40	20	0.51	0.51	0.00	1.15	0.36	0.55
42	21	1.02	1.02	0.00	2.30	0.71	1.09
44	22	2.05	2.08	0.00	4.72	1.48	2.21
46	23	4.07	4.10	0.00	9.20	2.92	4.40
48	24	8.18	8.17	0.00	18.39	5.84	8.76
50	25	16.23	16.33	0.00	36.85	11.63	17.48
52	26	32.63	32.76	0.00	73.64	23.31	34.97
54	27	65.28	65.45	0.00	147.36	46.68	69.95
56	28	130.29	131.30	0.01	294.48	93.11	140.01
58	29	259.96	261.51	0.01	588.60	186.31	279.53
60	30	521.40	523.44	0.00	1187.04	376.08	560.69

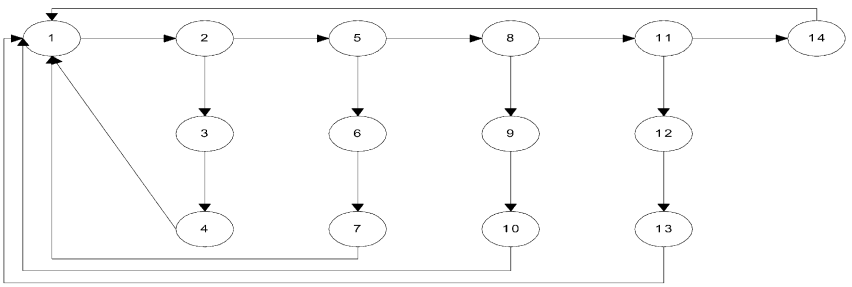

Fig. 6. Asymmetric graph G_2 with $n = 14$ ($p = 5$).

The outcomes are almost the same as for Example 1. The small difference is that the CPU time of CTS-F is slightly smaller than that of the CTS-R algorithm.

For this family of graphs, the number of steps required for visiting all nodes, with the CTS-R algorithm is exponential. More precisely, for a given graph with size $n = 3p-1$, the number of steps required by CTS-R is equal to $5(2^{(n-2)/3}) - 3$. However, the

Table 9

n	p	CTS-R	CTS-F	CTS-R-2	CTS-F-2	ACTS-R-2	ACTS-F-2
(a) *Number of steps V_n with asymmetric graph G_2*							
14	5	77	77	41	58	45	58
29	10	2557	2557	251	1918	1290	1918
44	15	81,917	81,917	761	61,438	40,975	61,438
59	20	2,621,437	2,621,437	1696	1,966,078	1,310,740	1,966,078
74	25	83,886,077	83,886,077	3181	62,914,558	41,943,065	62,914,558
(b) *Computing times with asymmetric graph G_2*							
14	5	0.00	0.00	0.00	0.00	0.00	0.00
29	10	0.00	0.00	0.00	0.00	0.01	0.00
44	15	0.02	0.02	0.00	0.04	0.01	0.02
47	16	0.04	0.03	0.00	0.08	0.02	0.04
50	17	0.07	0.07	0.00	0.19	0.06	0.08
53	18	0.16	0.15	0.00	0.36	0.11	0.18
56	19	0.31	0.30	0.01	0.71	0.23	0.35
59	20	0.60	0.62	0.00	1.44	0.46	0.67
62	21	1.24	1.21	0.00	2.85	0.92	1.35
65	22	2.46	2.44	0.00	5.69	1.83	2.71
68	23	4.96	4.91	0.00	11.38	3.68	5.41
71	24	9.90	9.84	0.01	23.30	7.38	10.86
74	25	19.76	19.75	0.00	45.50	14.83	22.23
77	26	38.99	39.06	0.00	85.84	26.87	40.31
80	27	78.00	78.18	0.01	171.69	53.73	80.67
83	28	155.98	157.37	0.00	343.25	107.42	161.15
86	29	311.79	312.53	0.00	685.80	214.54	323.00
89	30	499.43	499.87	0.00	1382.78	435.83	653.13

CTS-R-2 algorithm requires a polynomial number of steps, which is equal to $(n^3 + 21n^2 - 78n - 62)/162$. The numerical experiments shown in Tables 9(a) and (b) confirm the accuracy of the bounds given.

Example 3 Quadratic path in a symmetric graph (G_3)

In this example, we consider a graph of undirected edges, whose structure is similar to that of Example 1, except that the arcs $(2k+1, 1)$ that connect back to node 1 are replaced by edges $(2k+1, 2(k-1))$. The "right column" turns into a "ladder" (or a "saw tooth" structure). Specifically, the graph $G_n = \langle X, A \rangle$, where $n = 2p$, has the following form:

- $X = \{1, 2, \ldots, n\}$.
- $A = \{(1,2); (1,3)\} \cup \{(2k, 2k+1); (2k, 2k+2); (2k, 2k+3)$: for $k = 1, 2, \ldots, (n-2)/2\}$.

For example, the graph G_{10} ($p = 5$) is shown in Fig. 7.

The number of steps required for visiting all n nodes of the symmetric graph G_3 is quadratic and equal to $(3n^2 - 2n + 8)/2$ using CTS-R algorithm. This number of steps is linear for the other algorithms, that is, it is equal to $3n/2$ using the CTS-F algorithm and equal to $(3n-2)/2$ for the rest of the algorithms presented (CTS-R-2, CTS-F-2, ACTS-R-2, ACTS-F-2, TTS, E-TTS and RJ-TTS). Our numerical experiments in Table 10 confirm this fact.

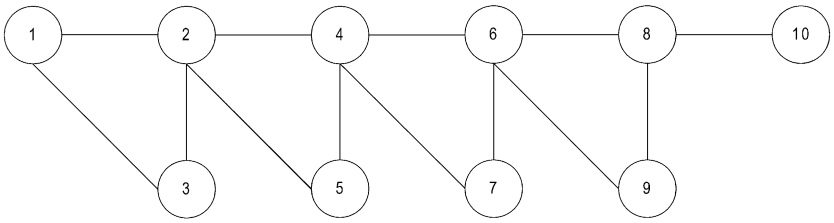

Fig. 7. Symmetric graph G_3 with $n = 10$ ($p = 5$).

Table 10
Computing times with symmetric graph G_3

n	CTS-R	CTS-F	CTS-R-2	CTS-F-2	ACTS-R-2	ACTS-F-2	TTS	E-TTS	RJ-TTS
20,000	52.71	0.01	0.03	0.03	0.02	0.02	0.02	0.02	0.02
40,000	264.79	0.03	0.05	0.06	0.05	0.04	0.04	0.04	0.05
60,000	668.37	0.05	0.08	0.09	0.06	0.06	0.05	0.06	0.07
80,000	1,130.51	0.07	0.11	0.12	0.09	0.09	0.06	0.09	0.09
100,000	1,086.86	0.08	0.13	0.13	0.11	0.10	0.09	0.12	0.10
120,000	1,090.22	0.11	0.15	0.16	0.12	0.13	0.10	0.12	0.12
140,000	1,120.31	0.12	0.19	0.19	0.14	0.15	0.12	0.15	0.16
160,000	1,112.22	0.14	0.21	0.22	0.18	0.17	0.13	0.17	0.17
180,000	1,089.16	0.16	0.23	0.24	0.23	0.19	0.16	0.19	0.18
200,000	1,098.09	0.17	0.25	0.27	0.23	0.22	0.16	0.21	0.21
Average	871.32	0.09	0.14	0.15	0.12	0.12	0.09	0.12	0.12

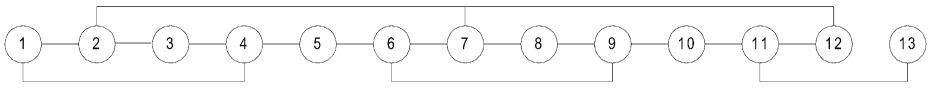

Fig. 8. Symmetric graph G_4 with $n = 13$ ($p = 2$).

Regarding the running time, we observe that for the family of graphs of type G_3 the 4 algorithms ACTS-R-2, ACTS-F-2, E-TTS and RJ-TTS require almost the same CPU times. The CTS-R-2 and CTS-F-2 algorithms are equivalent but are worse than the four methods cited previously. The CTS-R algorithm turns out to be the worst one for this type of graphs, while CTS-F and TTS algorithms are the best ones.

Example 4: Quadratic path in a symmetric graph (G_4)

We construct a graph $G_n = (X, A)$, where $n = 5p + 3$, as follows:

- $X = \{1, 2, \ldots, n\}$.
- $A = \{(k, k+1)$: for $k = 1, \ldots, n-2\} \cup \{(5k+1, 5k+4); (5k+2, 5k+7)$: for $k = 0, 1, \ldots, p-1\} \cup \{(n-2, n)\}$.

For example, the graph G_{13} ($p = 2$) is shown in Fig. 8.

Table 11
Number of steps with symmetric graph G_4

n	p	CTS-R	CTS-F	CTS-R-2
5003	1000	4,007,004	2,505,698	5004
10,003	2000	16,014,004	10,010,816	10,004
15,003	3000	36,021,004	22,517,042	15,004
20,003	4000	64,028,004	40,021,986	20,004
25,003	5000	100,035,004	62,526,658	25,004
30,003	6000	144,042,004	90,031,974	30,004
35,003	7000	196,049,004	122,539,310	35,004
40,003	8000	256,056,004	160,044,108	40,004
45,003	9000	324,063,004	202,549,054	45,004
50,003	10,000	400,070,004	250,055,332	50,004
100,003	20,000	1,600,140,004	1,000,111,262	100,004

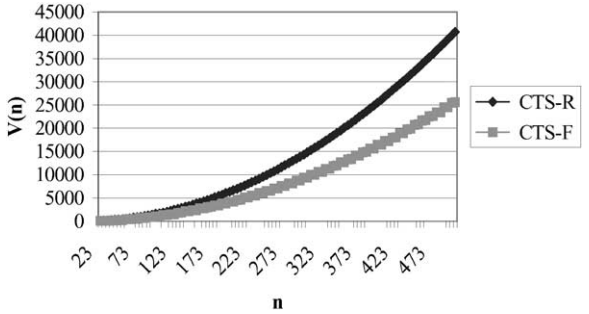

Fig. 9. Comparison between CTS-R and CTS-F algorithms with graph G_4.

For a given symmetric graph of type G_4 having $n = 5p + 3$ nodes the 7 algorithms (CTS-R-2, CTS-F-2, ACTS-R-2, ACTS-F-2, TTS, E-TTS and RJ-TTS) require the same linear number of steps for visiting all nodes, which is equal to $n+1$. The CTS-R algorithm requires a quadratic number of steps equal to $(4n^2 + 11n + 31)/25$. Table 11 and Fig. 9 show the comparison of CTS-R, CTS-F and CTS-R-2 algorithms.

Regarding the running times, we observe that for the family of graphs of type G_4 the 6 algorithms CTS-R-2, CTS-F-2, ACTS-R-2, ACTS-F-2, E-TTS and RJ-TTS require almost the same CPU times. The CTS-R and CTS-F algorithms turn out to be the worst ones for these graphs, while the TTS algorithm is the best one. This is shown in Table 12 below.

Example 5: *n*-dimensional binary vectors (G_5)

The 5th example is the one where the set of nodes $X = p$-dimensional binary vectors, so $n = |X| = 2^p$.

The number of steps required for visiting all $n = 2^p$ nodes of the symmetric graph G_5 is equal to n for the 7 algorithms CTS-F, CTS-F-2, ACTS-R-2, ACTS-F-2, TTS, E-TTS and RJ-TTS. The remaining 2 algorithms CTS-R and CTS-R-2 require a greater

Table 12
Computing times with symmetric graph G_4

n	CTS-R	CTS-F	CTS-R-2	CTS-F-2	ACTS-R-2	ACTS-F-2	TTS	E-TTS	RJ-TTS
5003	1.32	0.82	0.00	0.00	0.01	0.00	0.00	0.01	0.00
10,003	5.44	3.33	0.00	0.00	0.01	0.01	0.00	0.01	0.01
15,003	13.01	7.87	0.01	0.01	0.01	0.01	0.01	0.01	0.01
20,003	23.45	14.45	0.01	0.02	0.01	0.02	0.02	0.01	0.02
25,003	38.91	23.28	0.02	0.02	0.02	0.02	0.01	0.03	0.02
30,003	57.94	35.11	0.03	0.02	0.02	0.03	0.02	0.03	0.03
35,003	82.72	48.91	0.03	0.03	0.03	0.03	0.03	0.03	0.03
40,003	111.64	66.62	0.03	0.03	0.04	0.04	0.03	0.03	0.04
45,003	142.81	84.72	0.04	0.03	0.04	0.04	0.03	0.04	0.04
50,003	186.22	107.66	0.04	0.04	0.05	0.06	0.03	0.05	0.04
100,003	829.62	474.38	0.08	0.08	0.09	0.09	0.07	0.09	0.08
150,003	1,116.72	1,058.40	0.13	0.12	0.13	0.15	0.10	0.13	0.12
200,003	1,152.16	1,069.82	0.17	0.16	0.19	0.19	0.14	0.18	0.15
250,003	1,128.79	1,055.12	0.21	0.21	0.24	0.22	0.17	0.22	0.21
Average	349.34	289.32	0.06	0.06	0.06	0.07	0.05	0.06	0.06

Table 13
Number of steps with symmetric graph G_5

n	p	CTS-R	CTS-R-2	CTS-F
16	4	16	16	16
32	5	32	32	32
64	6	95	65	64
128	7	128	128	128
256	8	374	259	256
512	9	853	518	512
1024	10	1727	1031	1024
2048	11	3711	2469	2048
4096	12	8213	4948	4096
8192	13	15,729	8209	8192
16,384	14	33,539	23,027	16,384
32,768	15	73,454	45,756	32,768
65,536	16	156,696	93,150	65,536
131,072	17	303,645	159,170	131,072
262,144	18	662,755	383,121	262,144
524,288	19	1,286,126	908,743	524,288

number of steps. The following Table 13 and Figs. 10–12 compare these 2 algorithms with CTS-F algorithm.

Concerning the CPU times, we observe that the E-TTS and RJ-TTS algorithms which are enhancements of TTS give the expected results over this type of graphs, since they enhance the performance of TTS. CTS-F and CTS-F-2 algorithms require almost the same CPU times. Among the 9 algorithms tested ACTS-F-2 turns out to be the best one on this type of graphs, while the worst is the CTS-R algorithm (Table 14).

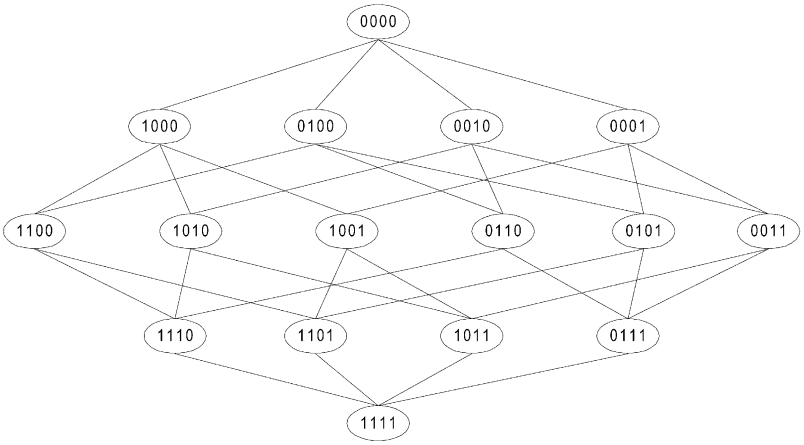

Fig. 10. Symmetric graph G_5 with $n = 16$ ($p = 4$).

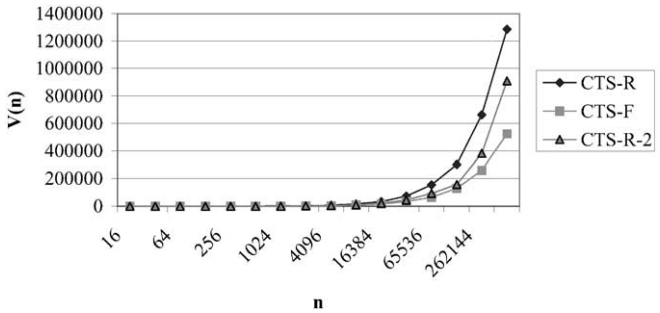

Fig. 11. Comparison of CTS-R, CTS-R-2 and CTS-F algorithms with graph G_5.

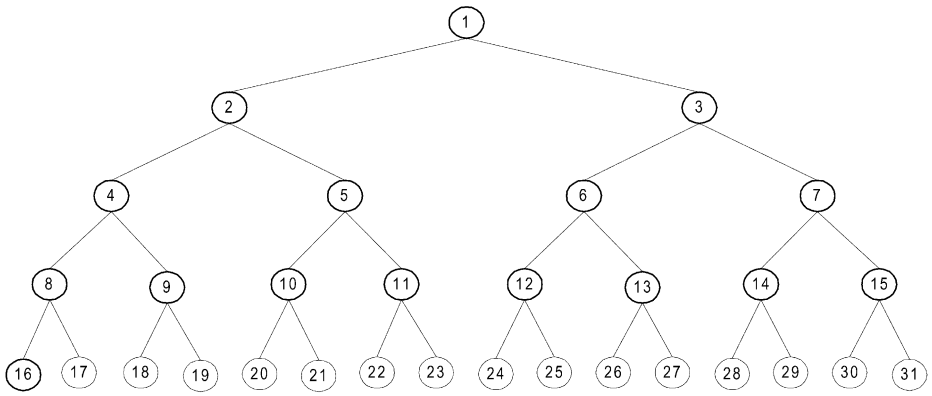

Fig. 12. Complete binary tree G_6 ($2^{p+1} - 1$) with $p = 4$.

Table 14
Computing times with symmetric graph G_5

n	p	CTS-R	CTS-F	CTS-R-2	CTS-F-2	ACTS-R-2	ACTS-F-2	TTS	E-TTS	RJ-TTS
8192	13	0.01	0.01	0.01	0.01	0.02	0.01	0.01	0.01	0.01
16,384	14	0.06	0.02	0.14	0.02	0.03	0.03	0.01	0.03	0.04
32,768	15	0.11	0.06	0.30	0.06	0.06	0.06	0.03	0.07	0.08
65,536	16	0.30	0.11	0.78	0.13	0.16	0.16	0.07	0.20	0.18
131,072	17	0.65	0.25	1.10	0.27	0.38	0.35	0.12	0.35	0.34
262,144	18	16.67	0.60	4.64	0.60	0.77	0.74	4.21	1.55	3.31
524,288	19	185.32	36.70	170.12	36.70	31.54	29.57	84.56	59.18	42.66
Average		29.02	5.39	25.30	5.40	4.71	4.42	12.72	8.77	6.66

Example 6: Complete binary tree (G_6)

We consider a complete binary tree of height p ($n=2^{p+1}-1$) represented as follows. The root corresponds to the node 1, and the left son of node i is the node numbered with $2i$ and the right son with $2i+1$. The father of a given node $i>1$, is the node $[i/2]$. In other terms, a complete binary tree of height p is represented by the graph $G_n=\langle X,A\rangle$, where:

- $X=\{1,2,\ldots,n\}$.
- $A=\{(1,2);(1,3)\}\cup\{(k,[k/2]);(k,2k);(k,2k+1)$: for $k=2,\ldots,2^p-1\}\cup\{(k,[k/2])$: for $k=2^p,\ldots,n\}$.

For the 4 algorithms CTS-R, CTS-R-2, ACTS-R-2 and TTS the number of steps required for visiting all n (where $n=2^{p+1}-1$) nodes of the symmetric graph G_6 is equal to $2n-\log_2(n+1)$. The number of steps of the two algorithms CTS-F and ACTS-F-2 as a function of nodes n is given by $(5n-2\log_2(n+1)-7)/2$. The number of steps of the algorithms CTS-F-2 as a function of nodes n is given by $(9n-4\log_2(n+1)-15)/4$. The number of steps of the algorithm E-TTS as a function of the parameter p can be described recursively by $V_4=49$ and $V_{p+1}=2(V_p+1)+[p/2]$. In terms of the number of nodes n, this translates explicitly into $[(10n-3\log_2(n+1)+1)/6]$. Among the 9 algorithms tested RJ-TTS turns out to be the fastest one on this type of graph, which requires $(3n-1)/2$ number of steps, while the worst ones are the CTS-F and ACTS-F-2 algorithms which require the same number of steps. The following Table 15(a) and (b) compares these algorithms.

Appendix D. Fathoming versus informed choice and channeling

One of the strongest advantages of branch and bound, not visible when simply itemizing all possible solutions, is the ability to avoid examining segments of the tree by fathoming — i.e., by determining that some branches offer no possibility of leading to an improved solution. Usually this is done by solving relaxed problems, easier to solve than the original, which give useful information about bounds or feasibility. In an integer programming context, this ability derives from the fact that the decision about values to be assigned to variables is deferred, and built up incrementally.

Table 15

n	CTS-R	CTS-F	CTS-R-2	CTS-F-2	ACTS-R-2	ACTS-F-2	TTS	E-TTS	RJ-TTS
(a) *Number of steps with complete binary tree* G_6									
31	57	69	57	61	57	69	57	49	46
63	120	148	120	132	120	148	120	102	94
127	247	307	247	275	247	307	247	208	190
255	502	626	502	562	502	626	502	421	382
511	1013	1265	1013	1137	1013	1265	1013	847	766
1023	2036	2544	2036	2288	2036	2544	2036	1700	1534
2047	4083	5103	4083	4591	4083	5103	4083	3406	3070
4095	8178	10,222	8178	9198	8178	10,222	8178	6819	6142
8191	16,369	20,461	16,369	18,413	16,369	20,461	16,369	13,645	12,286
16,383	32,752	40,940	32,752	36,844	32,752	40,940	32,752	27,298	24,574
32,767	65,519	81,899	65,519	73,707	65,519	81,899	65,519	54,604	49,150
65,535	131,054	163,818	131,054	147,434	131,054	163,818	131,054	109,217	98,302
131,071	262,125	327,657	262,125	294,889	262,125	327,657	262,125	218,443	196,606
262,143	524,268	655,336	524,268	589,800	524,268	655,336	524,268	436,896	393,214
524,287	1,048,555	1,310,695	1,048,555	1,179,623	1,048,555	1,310,695	1,048,555	873,802	786,430
1,048,575	2,097,130	2,621,414	2,097,130	2,359,270	2,097,130	2,621,414	2,097,130	1,747,615	1,572,862
(b) *Computing times with complete binary tree* G_6									
2047	0.00	0.00	0.01	0.00	0.00	0.01	0.00	0.00	0.00
4095	0.00	0.00	0.01	0.01	0.00	0.01	0.00	0.00	0.00
8191	0.01	0.01	0.01	0.02	0.00	0.01	0.01	0.01	0.00
16,383	0.01	0.01	0.03	0.03	0.02	0.02	0.01	0.02	0.01
32,767	0.04	0.04	0.06	0.07	0.04	0.05	0.03	0.04	0.03
65,535	0.06	0.07	0.12	0.13	0.09	0.11	0.07	0.08	0.06
131,071	0.14	0.14	0.25	0.27	0.19	0.21	0.15	0.16	0.12
262,143	0.28	0.33	0.49	0.54	0.39	0.43	0.28	0.33	0.24
524,287	0.54	0.61	0.96	1.11	0.77	0.87	0.60	0.66	0.50
1,048,575	30.25	3.16	1.99	2.36	1.62	1.85	13.47	13.19	7.40
Average	3.13	0.44	0.39	0.45	0.31	0.36	1.46	1.45	0.84

Yet this advantage comes with an associated disadvantage. The fewer the decisions that have been made (i.e., the fewer the variables that have been assigned values), the less complete is the information available about good values to assign to remaining variables. Consequently, in some settings this lack of information can lead to poor choices at early stages of the tree, and the inappropriateness of such choices can take a long time to discover. (In such cases, the influence of the poor choices is not only inherited by a large set of descendants, but the search may generate no information to suggest that the poor choices are indeed inferior, and that the branching alternatives in their part of the tree should be visited "out of sequence", as by a best bound rule.)

On the other hand, the type of approach that generates a full solution at each step, as illustrated in the earlier example of the TTS approach, affords fuller information about the contribution of each variable (given the values of the others). Thus, there is a certain advantage of "informed choice" available, even if this information is highly local in nature.

There is also another feature of the type of neighborhood-based search structures embodied in TTS and TS (with the Aspiration by Default rule), in contrast to the more usual branch-and-bound approaches. This derives from a conjecture that neighborhood structures often have a form that allows the search to be restricted to only a small part of the neighborhood space by following certain *channels* through it, which are collectively guaranteed to have access to optimal solutions. Under such circumstances, the finiteness guarantee applicable to the full space is likewise applicable to the reduced (*channeled*) space. The process of strategically selecting and following channels, which we call *channeling*, can significantly diminish the combinatorial complexity of the search and still offer the benefit of a finiteness guarantee.

The concept of channeling can be understood by considering as an example the special case of a 0-1 multidimensional knapsack problem (a maximization problem with less-than-or-equal-to constraints and all problem coefficients nonnegative). In this instance, it is clear that candidates for optimal solutions can be restricted to those that are as close as possible to the feasibility boundary, in the sense that no variable currently 0 can be changed to 1 (in an effort to move the solution closer to the boundary), except by violating feasibility. Thus a channel of solutions that hugs the boundary, moving just far enough away to allow access to other solutions that are appropriate candidates, is both strategically useful and offers a high likelihood of leading to an optimal solution. Channels that are allowed to penetrate to controlled depths on a given side of the feasibility boundary, or alternately on both sides, can be made subject to the finiteness rules we have identified. (These variable depth excursions, organized in relation to selected critical boundaries or regions, are the basis of the tabu search approach called *strategic oscillation*.) The allowance for channels that include moves through infeasible regions typically permits the channel width — the degree of departure from the feasibility boundary — to be reduced. In contexts more general than multidimensional knapsack problems, paths that traverse infeasible regions are often not merely useful but essential.

It is important to note that the channel tracing process cannot be done effectively by ordinary branch and bound. The reason stems from the following phenomenon: the region that demarcates a channel boundary characteristically is encountered by the "end branches" of a branch and bound tree. Accordingly, if variables whose branches appear earlier in the tree must change their values in order to progress along a promising channel, this can only be done by eliminating all decisions that are descendants of such branches, and then building up again a new set of decisions (by creating extensions of the alternatives to these branches). That is, the branch and bound type of tree search cannot stay within the channel region, because to progress between points that are contiguous within this region requires reverting to earlier parts of the tree (jumping out of the region). Modifying the values of earlier assigned variables is the only way to re-construct the access to the desired part of the channel. (An effort to shortcut the process either looses the tree structure or amounts to abandoning the branch and bound staging for exactly the kind of procedure we identify as an alternative.)

Channeling operates in different ways for different kinds of problems. The unifying feature of these applications is that channels leading to optimal solutions may be expected to involve an exploration of a much smaller portion of the space than would be generated by a full enumeration. This theme is similar to the type of expectation that exists in applying branch and bound, where the itemization of some limited number of alternatives is anticipated to succeed in reaching an optimal solution (in this case via fathoming). However, the mechanisms and the rationale leading to the expectation of a reduced search are entirely different for channeling than they are for the fathoming process of branch and bound. Whether one or the other of these expectations becomes more likely to be fulfilled will unquestionably depend on the setting. The relevance of channeling, as a strategy that offers a set of advantages contrasting with those of branch-and-bound fathoming, is worth heeding.

References

[1] F. Glover, M. Laguna, Tabu search, Kluwer Academic Publishers, Dordrecht, 1997.
[2] F. Glover, Tabu search, Part 2, ORSA J. Comput. 2 (1990) 4–32.
[3] S. Hanafi, On the convergence of tabu search, Working paper, University of Valenciennes, France, 1998, J. Heuristics, to appear.
[4] F. Dammeyer, S. Voss, Dynamic tabu list management using the reverse elimination method, Ann. Oper. Res. 41 (1993) 31–46.
[5] S. Hanafi, A. Fréville, Extension of reverse elimination method through a dynamic management of the tabu list, R.A.I.R.O., to appear.
[6] D. Avis, K. Fukuda, A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra, Proceedings of the Seventh ACM Symposium on Computational Geometry, North Conway, New Hampshire, 1991a, pp. 98–104.
[7] D. Avis, K. Fukuda, A basis enumeration algorithm for linear systems with geometric applications, Appl. Math. Lett. 5 (1991b) 39–42.
[8] D. Avis, K. Fukuda, Reverse search for enumeration, Discrete Appl. Math. 6 (1996) 21–46.
[9] G. Tarry, Le problème des labyrinthes, Nouv. Ann. Math. 14 (3) (1895) 187–190.
[10] G.L. Thompson, The Tarry Traverse, Class notes, Carnegie-Mellon University, Graduate School of Industrial Administration, 1998.
[11] A. Charnes, W.W. Cooper, Mathematical Models and Industrial Applications of Linear Programming, Vol. I, Wiley, New York and London, 1961, pp. 438–444.
[12] F. Glover, S. Hanafi, Composite tree searches for global optimization, Research Report, Graduate School of Business and Administration, University of Colorado, Boulder, 1998.