UNIVERSITÀ DEGLI STUDI
DI TRENTO

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

38050 Povo — Trento (Italy), Via Sommarive 14
http://dit.unitn.it/

REACTIVE SEARCH:
MACHINE LEARNING FOR MEMORY-BASED HEURISTICS

Roberto Battiti, Mauro Brunato
(battiti,brunato)@dit.unitn.it

October 27, 2005

Technical Report # DIT-05-058

Reactive Search: Machine Learning for Memory-Based Heuristics

Roberto Battiti, Mauro Brunato
Department of Computer Science and Telecommunications
Università di Trento
Via Sommarive, 14 — I-38050 Trento — Italy
E-mail: (battiti,brunato)@dit.unitn.it

October 27, 2005

1 Introduction: the role of the user in heuristics

Most state-of-the-art heuristics are characterized by a certain number of choices and free parameters, whose appropriate setting is a subject that raises issues of research methodology [5, 41, 51].

In some cases, these parameters are tuned through a feedback loop that *includes the user as a crucial learning component*: depending on preliminary algorithm tests some parameter values are changed by the user, and different options are tested until acceptable results are obtained. Therefore, the quality of results is not automatically transferred to different instances and the feedback loop can require a lengthy "trial and error" process every time the algorithm has to be tuned for a new application.

Parameter tuning is therefore a crucial issue both in the scientific development and in the practical use of heuristics. In some cases the role of the user as an intelligent (learning) part makes the reproducibility of heuristic results difficult and, as a consequence, the competitiveness of alternative techniques depends in a crucial way on the user's capabilities.

Reactive Search advocates the use of simple sub-symbolic machine learning to automate the parameter tuning process and make it an integral (and fully documented) part of the algorithm.

If learning is performed on line, task-dependent and local properties of the configuration space can be used by the algorithm to determine the appropriate balance between *diversification* (looking for better solutions in other zones of the configuration space) and *intensification* (exploring more intensively a small but promising part of the configuration space). In this way a single algorithm maintains the flexibility to deal with related problems through an internal feedback loop that considers the previous history of the search.

In the following, we shall call *reaction* the act of modifying some algorithm parameters in response to the search algorithm's behavior *during* its execution, rather than between runs. Therefore, a *reactive heuristic* is a technique with the ability of tuning some important parameters during execution by means of a machine learning mechanism.

It is important to notice that such heuristics are intrinsically history-dependent; thus, the practical success of this approach in some cases raises the need of a sounder theoretical foundation of non-Markovian search techniques.

1.1 Machine learning for automation and full documentation

process where experiments are designed in a focussed way, with the support of statistical estimation (parameter identification) tools.

Because of its familiarity with algorithms, the Computer Science community masters a very powerful tool for describing processes so that they can be reproduced even by a (mechanical) computer. In particular, the Machine Learning community, with significant influx from Statistics, developed in the last decades a rich variety of "design principles" that can be used to develop machine learning methods and algorithms.

It is therefore appropriate to consider whether some of these design principles can be profitably used in the area of parameter tuning for heuristics. The long-term goal is that of completely *eliminating the human* *intervention* in the tuning process. This does not imply higher unemployment rates in the CS community, on the contrary, the researcher is now loaded with a heavier task: the algorithm developer must aim at transferring his expertise into the algorithm itself, a task that requires the *exhaustive description of the tuning phase* in the algorithm.

Let us note that the algorithm "complexity" will increase as a result of the process, but the price is worth paying if the two following objectives are reached:

Complete and unambiguous documentation. The algorithm (and the research paper based on the algorithm) becomes self-contained and its quality can be judged independently from the designer or specific user. This requirement is particularly important from the scientific point of view, where objective evaluations are crucial. The recent introduction of software archives (in some cases related to scientific journals) further simplifies the test and *simple re-use* of heuristic algorithms.

Automation. The time-consuming tuning phase is now substituted by an automated process. Let us note that only the final user will typically benefit from an automated tuning process. On the contrary, the algorithm designer faces a longer and harder development phase, with a possible preliminary phase of exploratory tests, followed by the above described exhaustive documentation of the tuning process when the algorithm is presented to the scientific community.

Although formal learning frameworks do exist in the CS community (notably, the PAC learning model [67, 46]) one should not reach the conclusion that these models can be simply adapted to the new context. On the contrary, the theoretical framework of computational learning theory and machine learning is very different from that of heuristics. For example, the definition of a "quality" function against which the learning algorithm has to be judged is complex. In addition, the abundance of negative results in computational learning should warn about excessive hopes.

Nonetheless, as a first step, some of the principles and methodology used in machine learning can be used in an analogic fashion to develop "reactive heuristics."

1.2 Asymptotic results are irrelevant for optimization

Scientists, and also final users, might feel uneasy working with non-Markovian techniques because they don't benefit from the deep and wide theoretical background that covers Markovian algorithms. However, asymptotic convergence results of many Markovian search algorithms, such as Simulated Annealing [48], are often irrelevant for their application to optimization. As an example, a comparison of Simulated Annealing and Reactive Search has been presented in [12, 13].

In any finite-time approximation one must resort to approximations of the asymptotic convergence. In Simulated Annealing, for instance, the "speed of convergence" to the stationary distribution is determined by the second largest eigenvalue of the transition matrix. The number of transitions is at least *quadratic* in the size of the solution space [1], which is typically exponential in n.

When using a time-varying temperature parameter, it can happen (e.g., TSP problem) that the complete enumeration of all solutions would take less time than approximating an optimal solution with arbitrary precision by SA [1].

In addition, repeated local search [31], and even random search [23], have better asymptotic results. According to [1] "approximating the asymptotic behavior of SA arbitrarily closely requires a number of transitions that for most problems is typically larger than the size of the solution space [...] Thus, the SA algorithm is clearly unsuited for solving combinatorial optimization problems to optimality." Of course, practical utility of SA has been shown in many applications, in particular with fast cooling schedules, but then the asymptotic results are not directly applicable. The optimal finite-length annealing schedules obtained on specific simple problems do not always correspond to those intuitively expected from the limiting theorems [63].

2 Reactive Search applied to Tabu Search (RTS)

In this Section, we illustrate the potential of Reactive Search by installing a reaction mechanism on the prohibition period T of a Tabu Search [35] algorithm. For more comprehensive descriptions of Tabu Search, the Reader can refer to the Bibliography.

2.1 Prohibition-based diversification: Tabu Search

The Tabu Search meta-heuristic is based on the use of *prohibition-based* techniques and "intelligent" schemes as a complement to basic heuristic algorithms like local search, with the purpose of guiding the basic heuristic *beyond local optimality*. It is difficult to assign a precise date of birth to these principles. For example, ideas similar to those proposed in TS can be found in the *denial* strategy of [62] (once common features are detected in many suboptimal solutions, they are forbidden) or in the opposite *reduction* strategy of [49] (in an application to the Travelling Salesman Problem, all edges that are common to a set of local optima are fixed). In very different contexts, prohibition-like strategies can be found in *cutting planes* algorithms for solving integer problems through their Linear Programming relaxation (inequalities that cut off previously obtained fractional solutions are generated) and in branch and bound algorithms (subtrees are not considered if the leaves cannot correspond to better solutions). For many examples of such techniques, see the textbook [57].

The renaissance and full blossoming of "intelligent prohibition-based heuristics" starting from the late eighties is greatly due to the role of F. Glover in the proposal and diffusion of a rich variety of meta-heuristic tools [35, 36], but see also [39] for an independent seminal paper. A growing number of TS-based algorithms has been developed in the last years and applied with success to a wide selection of problems [37]. It is therefore difficult, if not impossible, to characterize a "canonical form" of TS, and classifications tend to be short-lived. Nonetheless, at least two aspects characterize many versions of TS: the fact that TS is used to complement local (neighborhood) search, and the fact that the main modifications to local search are obtained through the *prohibition* of selected moves available at the current point. TS acts to continue the search beyond the first local minimizer without wasting the work already executed, as it is the case if a new run of local search is started from a new random initial point, and to enforce appropriate amounts of diversification to avoid that the search trajectory remains confined near a given local minimizer.

In our opinion, the main competitive advantage of TS with respect to alternative heuristics based on local search like Simulated Annealing (SA) [48] lies in the intelligent use of the past history of the search to influence its future steps.

For a generic search space $\mathcal{X}$, let $X^{(t)} \in \mathcal{X}$ be the current configuration and $N(X^{(t)}) \subseteq \mathcal{X}$ its neighborhood. In prohibition-based search (Tabu Search) some of the neighbors can be *prohibited*, and let the subset $N_A(X^{(t)}) \subseteq N(X^{(t)})$ contain the *allowed* ones. The general way of generating the search trajectory that we consider is given by:

$$X^{(t+1)} = \textsc{Best-Neighbor}\left(N_A(X^{(t)})\right) \tag{1}$$

$$N_A(X^{(t+1)}) = \textsc{Allow}\left(N(X^{(t+1)}), X^{(0)}, \ldots, X^{(t+1)}\right) \tag{2}$$

The set-valued function Allow selects a subset of $N\big(X^{(t+1)}\big)$ in a manner that depends on the search trajectory $X^{(0)}, \ldots, X^{(t+1)}$.

This general framework allows several specializations. In many cases, the dependence of Allow on the entire search trajectory introduces too many constraints on the next move, causing the search path to avoid an otherwise promising area, or even prohibiting all neighbors. It is therefore advisable to reduce the amount of constraints by limiting the Allow function to the latest T configurations (where the parameter T is often called the *prohibition period*), so that equation (2) becomes

$$N_A(X^{(t+1)}) = \textsc{Allow}\left(N(X^{(t+1)}), X^{(t')}, \ldots, X^{(t+1)}\right), \qquad t' = \max\{0, t-T+1\} \tag{3}$$

A practical example for equation (3) is a function Allow that forbids all moves that have been performed within the last T iterations. For instance, let us assume that the feasible search space $\mathcal{X}$ is the set of binary strings with a given length L: $\mathcal{X} = \{0, 1\}^L$ (this case shall be considered also in the example of Section 4). In this case, a practical neighborhood of configuration $X^{(t)}$ is given by the L configurations that differ from $X^{(t)}$ by a single entry. In such case, a simple prohibition scheme may allow a move if and only if it changes an entry which has remained fixed for the previous T iterations. In other words, after an entry has been changed, it shall remain *frozen* for the following T steps.

It is apparent that the choice of the right prohibition period T is crucial in order to balance the amount of *intensification* (small T) and *diversification* (large T).

2.2 Reaction on Tabu Search parameters

Some problems arising in TS that have been investigated in Reactive Search papers are:

1. the determination of an appropriate prohibition T for the different tasks,
2. the robustness of the technique for a wide range of different problems,
3. the adoption of minimal computational complexity algorithms for using the search history.

The three issues are briefly discussed in the following sections, together with the reaction-based methods proposed to deal with them.

2.2.1 Self-adjusted prohibition period

In RTS the *prohibition period* T is determined through feedback (i.e., *reactive*) mechanisms during the search. At the beginning, we let $T = 1$ (the inverse of a given move is prohibited only at the next step). During the search, T increases only when there is *evidence* that diversification is needed, and it decreases when this evidence disappears. In detail: the evidence that diversification is needed is signalled by the repetition of previously visited configurations. For this purpose, all configurations found during the search are stored in memory. After a move is executed, the algorithm checks whether the current configuration has already been found and reacts accordingly (T increases if a configuration is repeated, T decreases if no repetitions occurred during a sufficiently long period).

By means of this self-adjustment algorithm, T is not fixed during the search, but it is determined in a dynamic way depending on the *local structure* of the search space. This is particularly relevant for "inhomogeneous" tasks, where the statistical properties of the search space vary widely in the different regions (in these cases a fixed T would be inappropriate).

An example of the behavior of T during the search is illustrated in Fig. 1, for a Quadratic Assignment Problem task [11]. T increases in an exponential way when repetitions are encountered, it decreases in a gradual manner when repetitions disappear.

2.2.2 The escape mechanism

based on prohibitions is not sufficient to avoid long cycles. As an example, when operating on binary strings of length L, the prohibition T must be less than the length of the string, otherwise all moves are eventually prohibited; therefore, cycles longer than $2 \times L$ are still possible. In addition, even if "limit cycles" (endless cyclic repetitions of a given set of configurations) are avoided, the first reactive mechanism is not sufficient to guarantee that the search trajectory is not confined in a limited region of the search space. A "chaotic trapping" of the trajectory in a limited portion of the search space is still possible (the analogy is with *chaotic attractors* of dynamical systems, where the trajectory is confined in a limited portion of the space, although a limit cycle is not present).

For both reasons, to increase the robustness of the algorithm a second more radical diversification step (*escape*) is needed. The *escape* phase is triggered when too many configurations are repeated too often [11]. A simple *escape* consists of a number of random steps executed starting from the current configuration (possibly with a bias toward steps that bring the trajectory away from the current search region).

With a stochastic escape, one can easily obtain the *asymptotic convergence* of RTS: in fact, in a finite search space *escape* is activated infinitely often; if the probability for a point to be reached after escaping is different from zero for all points, eventually all points will be visited - clearly including the globally optimal points. The detailed investigation of the asymptotic properties and finite-time effects of different *escape* routines to enforce long-term diversification is an open research area.

2.3 Implementation of history-sensitive techniques

The efficiency and competitiveness of history-based reaction mechanisms strongly depend on the detailed data structures used in the algorithms and on the consequent realization of the needed operations. Different data structures can possess widely different computational complexities so that attention should be spent on this subject before choosing a version of Reactive Search that is efficient on a particular problem.

Reactive-TS can be implemented through a simple list of visited configurations, or with more efficient hashing [70, 11] or radix tree [11] techniques. At a finer level of detail, hashing can be realized in different ways. If the entire configuration is stored (see also Fig. 2) an exact answer is obtained from the memory lookup operation (a repetition is reported if and only if the configuration has been visited before). On the contrary, if a "compressed" item is stored, like a hashed value of a limited length derived from the configuration, the answer will have a limited probability of *false positives* (a repetition can be reported even if the configuration is new, because the compressed items are equal by chance – an event called "collision"). Experimentally, small collision probabilities do not have statistically significant effects on the use of Reactive-TS as heuristic tool, and hashing versions that need only a few bytes per iteration can be used.

2.3.1 Fast algorithms for using the search history

The storage and access of the past events is executed through the well-known hashing or radix-tree techniques in a CPU time that is approximately *constant* with respect to the number of iterations. Therefore the overhead caused by the use of the history is negligible for tasks requiring a non-trivial number of operations to evaluate the cost function in the neighborhood.

An example of a memory configuration for the hashing scheme is shown in Fig. 2. From the current configuration ϕ one obtains an index into a "bucket array." The items (configuration or hashed value or derived quantity, last time of visit, total number of repetitions) are then stored in linked lists starting from the indexed array entry. Both storage and retrieval require an approximately constant amount of time if: i) the number of stored items is not much larger than the size of the bucket array, and ii) the *hashing function* scatters the items with a uniform probability over the different array indices. More precisely, given a hash table with m slots that stores n elements, a load factor $\alpha = n/m$ is defined. If collisions are resolved by chaining searches take $O(1+\alpha)$ time, on the average.

2.3.2 Persistent dynamic sets

Persistent dynamic sets are proposed to support memory–usage operations in history-sensitive heuristics in [6, 7].

Ordinary data structures are *ephemeral* [28] because when a change is executed the previous version is destroyed. Now, in many contexts like computational geometry, editing, implementation of very high level programming languages, and, last but not least, the context of history-based heuristics, multiple versions of a data structure must be maintained and accessed. In particular, in heuristics one is interested in *partially persistent* structures, where all versions can be accessed but only the newest version (the *live* nodes) can be modified. A review of *ad hoc* techniques for obtaining persistent data structures is given in [28] that is dedicated to a systematic study of persistence, continuing the previous work of [56].

Hashing combined with persistent red-black trees The basic observation is that, because *tabu search* is based on local search, configuration $X^{(t+1)}$ differs from configuration $X^{(t)}$ only because of the addition or subtraction of a single index (a single bit is changed in the string). Let us define the operations $\textsc{Insert}(i)$ and $\textsc{Delete}(i)$ for inserting and deleting a given index i from the set. As cited above, configuration X can be considered as a set of indices in $[1, L]$ with a possible realization as a balanced red-black tree, see [18, 38] for two seminal papers about red-black trees. The binary string can be immediately obtained from the tree by visiting it in symmetric order, in time $O(L)$. $\textsc{Insert}(i)$ and $\textsc{Delete}(i)$ require $O(\log L)$ time, while at most a single node of the tree is allocated or deallocated at each iteration. Rebalancing the tree after insertion or

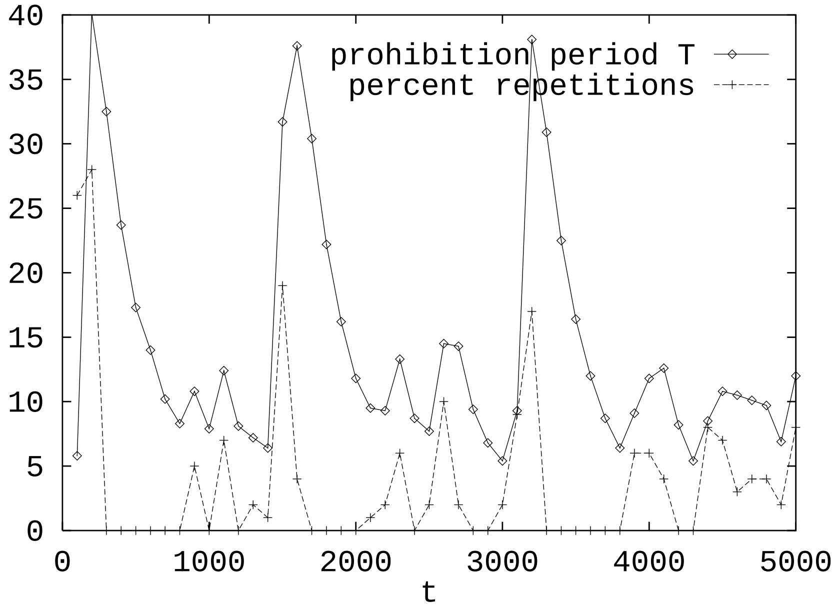

Figure 1: Dynamics of the the prohibition period T on a QAP task.

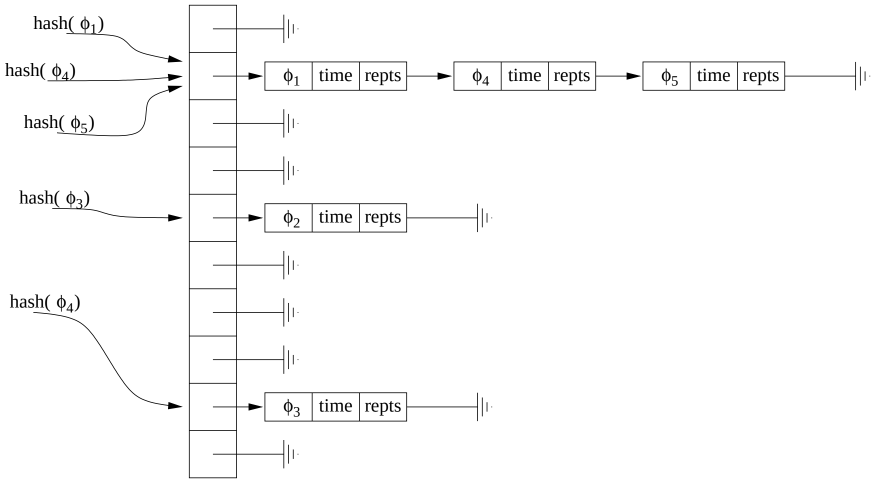

Figure 2: Open hashing scheme: items (configuration, or compressed hashed value, etc.) are stored in "buckets". The index of the bucket array is calculated from the configuration.

deletion can be done in $O(1)$ rotations and $O(\log L)$ color changes [65]. In addition, the amortized number of color changes per update is $O(1)$, see for example [50].

Now, the Reverse Elimination Method [35, 36, 26] (a technique for the storage and analysis of the ordered list of all moves performed throughout the search) is closely reminiscent of a method studied in [56] to obtain partial persistence, in which the entire update sequence is stored and the desired version is rebuilt from scratch each time an access is performed, while a systematic study of techniques with better space-time complexities is present in [60, 28]. Let us now summarize from [60] how a partially persistent red-black tree can be realized. An example of the realizations that we consider is presented in Fig. 3.

The trivial way is that of keeping in memory all copies of the ephemeral tree (see the top part of Fig. 3), each copy requiring $O(L)$ space. A smarter realization is based on *path copying*, independently proposed by many researchers, see [60] for references. Only the path from the root to the nodes where changes are made is copied: a set of search trees is created, one per update, having different roots but *sharing* common subtrees. The time and space complexities for Insert(i) and Delete(i) are now of $O(\log L)$.

The method that we will use is a space-efficient scheme requiring only linear space proposed in [60]. The approach avoids copying the entire access path each time an update occurs. To this end, each node contains an additional "extra" pointer (beyond the usual left and right ones) with a time stamp.

When attempting to add a pointer to a node, if the extra pointer is available, it is used and the time of the usage is registered. If the extra pointer is already used, the node is copied, setting the initial left and right pointers of the copy to their latest values. In addition, a pointer to the copy is stored in the last parent of the copied node. If the parent has already used the extra pointer, the parent, too, is copied. Thus copying proliferates through successive ancestors until the root is copied or a node with a free extra pointer is encountered. Searching the data structure at a given time t in the past is easy: after starting from the appropriate root, if the extra pointer is used the pointer to follow from a node is determined by examining the time stamp of the extra pointer and following it iff the time stamp is not larger than t. Otherwise, if the extra pointer is not used, the normal left-right pointers are considered. Note that the pointer direction (left or right) does not have to be stored: given the search tree property it can be derived by comparing the indices of the children with that of the node. In addition, colors are needed only for the most recent (live) version of the tree. In Fig. 3 `null` pointers are not shown, colors are correct only for the live tree (the nodes reachable from the rightmost root), extra pointers are dashed and time-stamped.

The worst-case time complexity of Insert(i) and Delete(i) remains of $O(\log L)$, but the important result derived in [60] is that the amortized space cost per update operation is $O(1)$. Let us recall that the total amortized space cost of a sequence of updates is an upper bound on the actual number of nodes created.

Let us now consider the context of history-based heuristics. Contrary to the popular usage of persistent dynamic sets to search past versions at a specified time t, one is interested in checking whether a configuration has already been encountered in the previous history of the search, at *any* iteration.

A convenient way of realizing a data structure supporting X-Search(X) is to combine *hashing* and *partially persistent dynamic sets*, see Fig. 4. From a given configuration X an index into a "bucket array" is obtained through a hashing function, with a possible incremental evaluation in time $O(1)$. Collisions are resolved through chaining: starting from each bucket header there is a linked list containing a pointer to the appropriate root of the persistent red-black tree and satellite data needed by the search (time of configuration, number of repetitions).

As soon as configuration $X^{(t)}$ is generated by the search dynamics, the corresponding persistent red-black tree is updated through Insert(i) or Delete(i). Let us now describe X-Search($X^{(t)}$): the hashing value is computed from $X^{(t)}$ and the appropriate bucket searched. For each item in the linked list the pointer to the root of the past version of the tree is followed and the old set is compared with $X^{(t)}$. If the sets are equal, a pointer to the item on the linked list is returned. Otherwise, after the entire list has been scanned with no success, a `null` pointer is returned.

In the last case a new item is linked in the appropriate bucket with a pointer to the root of the live version of the tree (X-Insert(X, t)). Otherwise, the last visit time t is updated and the repetition counter is incremented.

After collecting the above cited complexity results, and assuming that the bucket array size is equal to the maximum number of iterations executed in the entire search, it is straightforward to conclude that each iteration of *reactive-TS* requires $O(L)$ average-case time and $O(1)$ amortized space for storing and retrieving the past configurations and for establishing prohibitions.

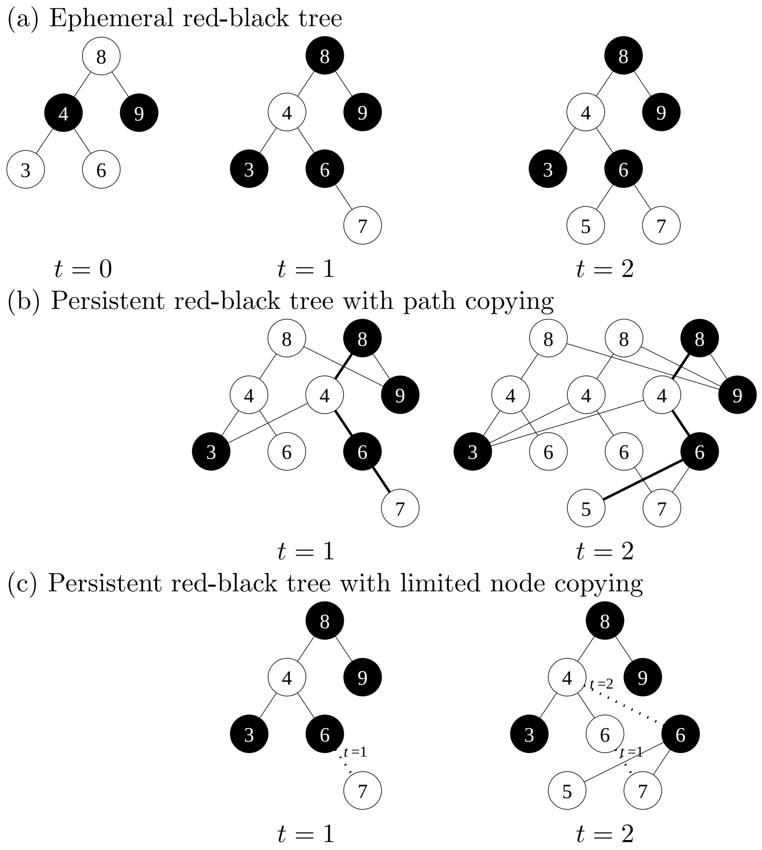

Figure 3: How to obtain a partially persistent red-black tree from an ephemeral one (top), containing indices 3,4,6,8,9 at t=0, with subsequent insertion of 7 and 5. Path copying (middle), with thick lines marking the copied part. Limited node copying (bottom) with dashed lines denoting the "extra" pointers with time stamp.

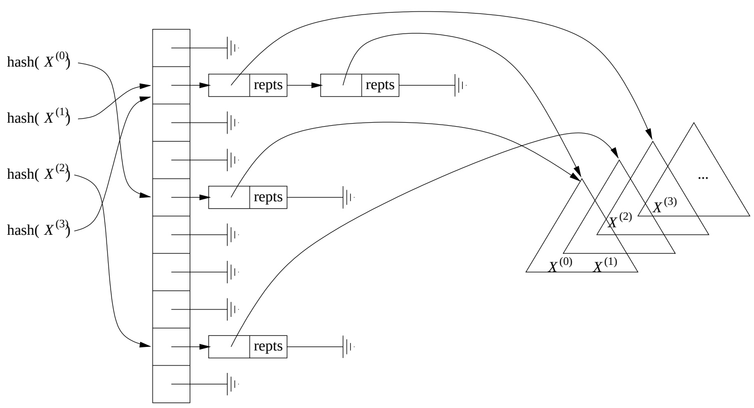

Figure 4: Open hashing scheme with persistent sets: a pointer to the appropriate root for configuration $X^{(t)}$ in the persistent search tree is stored in a linked list at a "bucket". Items on the list contain satellite data. The index of the bucket array is calculated from the configuration through a hashing function.

In fact, both the hash table and the persistent red-black tree require $O(1)$ space (amortized for the tree). The worst-case time complexity per iteration required to update the current $X^{(t)}$ is $O(\log L)$, the average-case time for searching and updating the hashing table is $O(1)$ (in detail, searches take time $O(1+\alpha)$, α being the load factor, in our case upper bounded by 1). The time is therefore dominated by that required to compare the configuration $X^{(t)}$ with that obtained through X-SEARCH($X^{(t)}$), i.e., $O(L)$ in the worst case. Because $\Omega(L)$ time is needed during the neighborhood evaluation to compute the f values, the above complexity is optimal for the considered application to history-based heuristics.

3 Wanted: a theory of history-sensitive heuristics

Randomized Markovian local search algorithms have enjoyed a long period of scientific and applicative excitement, in particular see the flourishing literature on Simulated Annealing [48, 44]. SA generates a *Markov chain*: the successor of the current point is chosen stochastically, with a probability that does not depend on the previous history (standard SA does not learn). A consequence is that the "trapping" of the search trajectory in an attractor cannot be avoided: the system has no memory and cannot detect that the search is localized. Incidentally, the often cited asymptotic convergence results of SA are unfortunately irrelevant for the application of SA to optimization. In fact, repeated local search [31], and even random search [23] has better asymptotic results.

History-sensitive techniques in local search contain an internal *feedback loop* that uses the information derived from the past history to influence the future behavior. In the cited prohibition-based diversification techniques one can, for example, decide to increase the diversification when configurations are encountered again along the search trajectory [11, 10]. It is of interest that state-of-the-art versions of Simulated Annealing incorporate "temporal memory" [33]. The non-Markovian property is a mixed blessing: it permits heuristic results that are much better in many cases, but makes the theoretical analysis of the algorithm difficult.

Therefore one has an unfortunate chasm: on one side there is an abundance of mathematical results derived from the theory of Markov processes, but their relevance to optimization is dubious, on the other side there is mounting evidence that simple *machine learning* or history-sensitive schemes can augment the performance of heuristics in a significant way, but the theoretical instruments to analyze history-sensitive heuristics are lacking.

The practical success of history-sensitive techniques should motivate new search streams in Mathematics and Computer Science for their theoretical foundation.

4 Applications of Reactive Search and the Maximum Clique example

Reactive Search principles have been used for example for the problem of Quadratic Assignment [11], training neural nets and control problems [14], vehicle-routing problems [24, 55, 21], structural acoustic control problems [47], special-purpose VLSI realizations [2], graph partitioning [16], electric power distribution [66], maximum satifiability [8], constraint satisfaction [9, 54], optimization of continuous functions [15, 22], traffic grooming in optical networks [17], maximum clique [10], real-time dispatch of trams in storage yards [69], increasing internet capacity [32]. Because of space limitation we consider with some detail only the application to the maximum clique problem.

The Maximum Clique (MC) problem is NP-hard, and strong negative results have been shown about its approximability [3, 25]. In particular, if P $\neq$ NP, MC is *not approximable* within $n^{1/4-\epsilon}$ for any $\epsilon > 0$, n being the number of nodes in the graph [19], and it is not approximable within $n^{1-\epsilon}$ for any $\epsilon > 0$, unless coRP = NP [40].

These theoretical results stimulated a research effort to design efficient heuristics for this problem, and computational experiments to demonstrate that optimal or close approximate values can be efficiently obtained for significant families of graphs [45, 58].

In particular, a new *reactive* heuristic (Reactive Local Search or RLS) is proposed for the Maximum Clique problem in [10]. The present description is a summarized version of the cited paper.

The experimental efficacy and efficiency [10] of RLS is strengthened by an analysis of the complexity of a single iteration. It is possible to show that the worst-case cost is $O(\max\{n, m\})$ where n and m are the number of nodes and edges, respectively. In practice, the cost analysis is pessimistic and the measured number of operations tends to be a small constant times the average degree of nodes in $\overline{G}$, the complement of the original graph.

4.1 Reactive Local Search for Max-Clique

The Reactive Local Search (RLS) algorithm for the maximum clique problem takes into account the particular neighborhood structure of MC. This is reflected in the following two facts: a single reactive mechanism is used to determine the prohibition parameter T, and an explicit restart scheme is added so that all possible configurations will eventually be visited, even if the search space is not connected by using the basic local-search moves. Both building blocks of RLS use the past history of the search (set of visited configurations) to influence the choice.

The admissible search space $\mathcal{X}$ is the set of all cliques in a graph G defined over a vertex set V. Let us recall that a clique is a subset X of V such that all pairs of nodes in X are connected by an edge. The function to be maximized is the clique size $f(X) = |X|$, X being the current clique, and the neighborhood $M(X)$ consists of all cliques that can be obtained from X by adding or dropping a single vertex (*add* or *drop* moves).

At a given iteration, the neighborhood set $M(X)$ is partitioned into the set of *prohibited* neighbors and the set of *allowed* ones. As soon as a vertex is moved (added or removed from the current clique), changing its status is prohibited for the next T iterations; it is allowed otherwise. With a slight abuse of terminology, the terms *allowed* and *prohibited* shall also be applied to vertices.

The top-level description of RLS is shown in Fig. 5. First (lines 2–5) the relevant variables and structures are initialized: they are the iteration counter t, the prohibition period T, the time t_T of the last change of T, the last restart time t_R, the current clique X, the largest clique I_b found so far with its size k_b, and the iteration t_b at which it is found. The set S shall be used by function Best-Neighbor and contains the set of eligible nodes to improve the current clique (initially, the current clique is empty, and no node is prohibited, so all nodes in V are eligible); the role of array *deltaS* shall be explained in Section 4.1.1. Then the loop (lines 6–15) continues to be executed until a satisfactory solution is found or a limiting number of iterations is reached.

The function History-Reaction searches for the current clique X in memory, inserts it if it is a new one, and adjusts the prohibition T through feedback from the previous history of the search.

Then the best neighbor is selected and the current clique updated (line 8). The iteration counter is incremented. If a better solution is found, the new solution, its size and the time of the last improvement are saved (lines 10–11). A restart is activated after a suitable number of iterations are executed from the last improvement and from the last restart (lines 12–13).

The prohibition period T is equal to one at the beginning, because in this manner one avoids coming back to the just abandoned clique. Nonetheless, let us note that RLS behaves exactly as local search in the first phase, as long as only new vertices are added to the current clique X, and therefore prohibitions do not have any effect. The difference starts when a maximal clique with respect to set inclusion is reached and the first vertex is dropped.

4.1.1 Choice of the best neighbor

The function Best-Neighbor is described in Fig.6. Given a current clique X, let us define as S the vertex set of possible additions, i.e., the vertices that are adjacent to all nodes of X. $G(S)$ is the subgraph induced by S. Finally, if $j \in X$, *deltaS*$[j]$ is the number of vertices adjacent to all nodes of X but j. A vertex v is *prohibited* at iteration t iff it satisfies *lastMoved*$[v] \geq (t - T^{(t)})$, where *lastMoved*$[v]$ is the last iteration at which it has been added to or dropped from the current clique. *lastMoved* is a vector used to determine the allowed vertices, see lines 3 and 11.

The best neighbor is chosen in stages with this overall scheme: first an allowed vertex that can be added to the current clique is searched for (lines 3–7). If none is found, an allowed vertex to drop is searched for

Global variables and data structures	
t	Time (iteration counter)
t_T	Time of last period change
S	Nodes in $V \setminus X$ adjacent to all nodes in X
deltaS$[j]$	Nodes in $V \setminus S$ adjacent to all nodes in $X \setminus \{j\}$
k_b	Cardinality of best configuration
lastMoved$[v]$	Time of last movement concerning node v

Local variables	
T	Prohibition period
t_R	Time of last restart
X	Current configuration
I_b	Best configuration
t_b	Time of best configuration

procedure REACTIVE-LOCAL-SEARCH
  $t \leftarrow 0$; $T \leftarrow 1$; $t_T \leftarrow 0$; $t_R \leftarrow 0$
  $X \leftarrow \emptyset$; $I_b \leftarrow \emptyset$; $k_b \leftarrow 0$; $t_b \leftarrow 0$
  S $\leftarrow$ V ; $\forall j \in V$ *deltaS*$[j] \leftarrow \emptyset$
  $\forall j \in V$ *lastMoved*$[j] \leftarrow -\infty$
  repeat
    $T \leftarrow$ HISTORY-REACTION (X, T)
    $X \leftarrow$ BEST-NEIGHBOR (X)
    $t \leftarrow t + 1$
    if $|X| > k_b$
      $I_b \leftarrow X$; $k_b \leftarrow |X|$; $t_b \leftarrow t$
    if t - $\max\{t_b, t_R\} > 100\ k_b$
      $t_R \leftarrow t$; RESTART
  until k_b is acceptable
    or maximum number of iterations reached

Figure 5: RLS Algorithm: Pseudo-Code Description.

Parameters	
X	Configuration to be changed

Local variables	
type	Type of move (`addMove` or `dropMove`)
v	Node to be added or removed

function BEST-NEIGHBOR (X)
  type $\leftarrow$ `notFound`
  if {allowed nodes $\in S\} \neq \emptyset$
    type $\leftarrow$ `addMove`
    maxDegAllowed $\leftarrow$ maximum degree in $G(S)$
    $v \leftarrow$ random allowed $w \in S$
      with $\deg_{G(S)}(w) =$ *maxDegAllowed*
  if *type* $=$ `notFound` **and** $X \neq \emptyset$
    type $\leftarrow$ `dropMove`
    if {allowed $v \in X\} \neq \emptyset$
      maxDeltaS $\leftarrow \max_{\text{allowed}_{j \in X}}$ *deltaS*$[j]$
      $v \leftarrow$ random allowed $w \in X$
        with *deltaS*$[w] =$ *maxDeltaS*
    else
      $v \leftarrow$ random $w \in X$
  if *type* $=$ `notFound`
    type $\leftarrow$ `addMove`
    $v \leftarrow$ random $w \in V$
  INCREMENTAL-UPDATE $(v$, *type*$)$
  if *type* $=$ `addMove` **return** $X \cup \{v\}$
    else **return** $X \setminus \{v\}$

Figure 6: RLS Algorithm: the function BEST-NEIGHBOR.

(lines 10–13). Finally, if no allowed moves are available, a random vertex in X is dropped if X is not empty (line 15), a random vertex in V is added in the opposite case (lines 16–18).

Ties among *allowed* vertices that can be added are broken by preferring the ones with the largest degree in $G(S)$ (line 7); a random selection is executed among vertices with equal degree in $G(S)$.

Ties among *allowed* vertices that can be dropped are broken by preferring those causing the largest increase $|S^{(t+1)}| - |S^{(t)}|$ where $S^{(t)}$ is the set S at iteration t (line 13). Again, a random selection is then executed if this criterion selects more that one winner.

4.1.2 Reaction and periodic restart

The function History-Reaction is illustrated in Fig. 7. The prohibition T is minimal at the beginning ($T = 1$), and is then determined by two competing processes: T increases when the current clique comes back to one already found, it decreases when no cliques are repeated in a suitable period. In detail: the current clique X is searched in memory, by utilizing hashing techniques (line 1). If X is found, a reference Z is returned to a data structure containing the last visit time (line 2). If the repetition interval R is sufficiently short, cycles are discouraged by increasing T (lines 5-7). If X is not found, it is stored in memory with the time t when it was encountered (line 9). If T remained constant for a number of iterations greater than $10k_b$, and therefore no clique is repeated during this interval, it is decreased (lines 10-12). Increases and decreases, with a minimal change of one unit plus upper and lower bounds, are realized by the two following functions:

$$\begin{aligned} \text{Increase}(T) &= \min\{\max\{T \cdot 1.1, T+1\}, n-2\} \\ \text{Decrease}(T) &= \max\{\min\{T \cdot 0.9, T-1\}, 1\} \end{aligned}$$

The routine Restart is similar to that in [61]. If there are vertices that have never been part of the current clique during the search, i.e., that have never been moved since the beginning of the run, one of them with maximal degree in V is randomly selected (lines 3-5 in Fig. 8). If all vertices have already been members of X in the past, a random vertex in V is selected (line 7). Data structures are updated to reflect the situation of $X = \emptyset$, then the incremental update is applied and the vertex v is added.

4.2 Complexity per iteration

The computational complexity of each iteration of RLS is the sum of a term caused by the usage and updating of reaction-related structures and a term caused by the local search part: neighborhood evaluation and generation of the next clique.

Let us first consider the reaction-related part. The overhead per iteration incurred to determine the prohibitions is $O(|M(X)|)$, $M(X)$ being the neighborhood, that for updating the last usage time of the chosen move is $O(1)$, that to check for repetitions, and to update and store the new *hashing value* of the current configuration has an average complexity of $O(1)$, if an incremental *hashing* calculation is applied.

In our case the single-iteration RLS complexity is dominated by the neighborhood evaluation. This evaluation requires an efficient update of the sets S and SMinus plus the computation of the degrees of the vertices in the induced subgraph $G(S)$ (used in function Best-Neighbor, Fig. 6, line 6). It is therefore crucial to consider incremental algorithms, in an effort to reduce the complexity. In our algorithm, the sets S and SMinus are maintained by the routine Incremental-Update that is used in the function Best-Neighbor and in the procedure Restart. The limited space of this extended abstract force us to omit the detailed description of Incremental-Update and of the related data structures. The following theorem is proved in the full paper:

Theorem 1 *The incremental algorithm for updating X, S and* Sminus *during each iteration of RLS has a worst case complexity of $O(n)$. In particular, if vertex v is added to or deleted from S, the required operations are $O(\deg_{\overline{G}}(v))$.*

Let us note that the actual multiplicative constant is very small and that the algorithm tends to be faster for dense graphs where the average degree $\deg_{\overline{G}}(v)$ in the complement graph can be much smaller than n.

Finally, the computation of the vertex degrees in the induced subgraph $G(S)$ costs at most $O(m)$ by the following trivial algorithm. All the edges are inspected, if both end-points are in S, the corresponding degrees are incremented by 1. In practice the degree is not computed from scratch but it is updated incrementally with a much lesser computational effort: in fact the maximum number of nodes that enter or leave S at a given iteration is at most $\deg_{\overline{G}}(v)$, v being the just moved vertex. Therefore the number of operations performed is at most $O(\deg_{\overline{G}}(v) \cdot |S^{(t+1)}|)$. Because the search aims at maximizing the clique X, the set S tends to be very small (at some steps empty!) after a first transient period, and the dominant factor is the same $O(\deg_{\overline{G}}(v))$ factor that appears in the above theorem.

Putting together all the complexity considerations the following Corollary is immediately implied:

Corollary 1 *The worst-case complexity of a single iteration is* $O(\max\{n, m\})$.

Experimental results of RLS on the benchmark suite for the MC problem by the organizers of the Second DIMACS Implementation Challenge [45], are analyzed in [10]

5 Related reactive approaches

Because Reactive Search is rooted in the automation of the algorithm tuning process by embodying the typical experimental cycle followed by heuristic algorithm designers, it is not surprising to see related ideas and principles arising in various areas. For space limitation we list in this section a limited selection of interesting related papers.

Probabilistic methods that explicitly maintain statistics about the search space by creating models of the good solutions found so far are considered for example in [59, 4].

Implicit models of the configuration space are built by a population of searchers in Genetic Algorithms, where learning comes in the form of "survival of the fittest" and generation of new sampling points depends on the previous evolution, see for example [64]. The issue of controlling parameters of an evolutionary algorithm, including adaptive and "self-adaptive" techniques is considered in [30], while fitness landscape analysis for the choice of appropriate operators in "memetic" algorithms (combining genetic algorithms with local search) is considered in [52].

In the area of stochastic optimization, which considers noise in the evaluation process, memory based schemes for validation and tuning of function approximators are used in [53, 29].

Guided local search aims at exploiting the problem and search-related information to effectively guide local search heuristics [68]. Evaluation functions for global optimization and Boolean satisfiability are learnt in [20].

Learning mechanisms with biological motivations are used in Ant Colony Optimization based on feedback, distributed computation, and the use of a constructive greedy heuristic. For example "pheromone trail" information to perform modifications on solutions for the quadratic assignment problem is considered in [27, 34].

Dynamic local search, which increases penalties of some solution components to move the tentative solution away from a given local minimum can also be considered as a form of learning based on the previous history of the search. An example is the dynamic local search algorithm on the MAXSAT problem in [43], see also [42] for additional references about stochastic local search methods including adaptive versions.

In the area of computer systems management, "autonomic" systems are based on self-regulating biological systems (http://www.research.ibm.com/autonomic/) which have many points of contact with Reactive Search.

References

[1] E. H. L. Aarts, J.H.M. Korst, and P.J. Zwietering, *Deterministic and randomized local search*, Mathematical Perspectives on Neural Networks (M. Mozer P. Smolensky and D. Rumelhart, eds.), Lawrence Erlbaum Publishers, Hillsdale, NJ, 1995.

[2] G. Anzellotti, R. Battiti, I. Lazzizzera, G. Soncini, A. Zorat, A. Sartori, G. Tecchiolli, and P. Lee, *TOTEM: a highly parallel chip for triggering applications with inductive learning based on the reactive tabu search*, International Journal of Modern Physics C **6** (1995), no. 4, 555–560, Postscript available at http://rtm.science.unitn.it/~battiti/battiti-publications.html.

[3] G. Ausiello, P. Crescenzi, and M. Protasi, *Approximate solution of NP optimization problems*, Theoretical Computer Science **150** (1995), 1–55.

[4] S. Baluja and S. Davies, *Using optimal dependency trees for combinatorial optimization: Learning the structure of the search space*, Proceedings of the Fourteenth International Conference on Machine Learning (San Mateo, CA.) (D.H. Fisher, ed.), Morgan Kaufmann Publishers, 1997, pp. 30–38.

[5] R. S. Barr, B. L. Golden, J. P. Kelly, M. G. C. Resende, and W. Stewart, *Designing and reporting on computational experiments with heuristic methods*, Journal of Heuristics **1** (1995), no. 1, 9–32.

[6] R. Battiti, *Time- and space-efficient data structures for history-based heuristics*, Tech. Report UTM-96-478, Dip. di Matematica, Univ. di Trento, 1996.

[7] ______, *Partially persistent dynamic sets for history-sensitive heuristics*, Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Challenges (D. S. Johnson M. H. Goldwasser and C.C. McGeoch, eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 59, American Mathematical Society, 2002, pp. 1–21.

[8] R. Battiti and M. Protasi, *Reactive search, a history-sensitive heuristic for MAX-SAT*, ACM Journal of Experimental Algorithmics **2** (1997), no. ARTICLE 2, http://www.jea.acm.org/.

[9] ______, *Reactive local search techniques for the maximum k-conjunctive constraint satisfaction problem*, Discrete Applied Mathematics **96-97** (1999), 3–27.

[10] ______, *Reactive local search for the maximum clique problem*, Algorithmica **29** (2001), no. 4, 610–637.

[11] R. Battiti and G. Tecchiolli, *The reactive tabu search*, ORSA Journal on Computing **6** (1994), no. 2, 126–140, Postscript available at http://rtm.science.unitn.it/˜battiti/battiti-publications.html.

[12] ______, *Simulated annealing and tabu search in the long run: a comparison on QAP tasks*, Computer and Mathematics with Applications **28** (1994), no. 6, 1–8, Postscript available at http://rtm.science.unitn.it/˜battiti/battiti-publications.html.

[13] ______, *Local search with memory: Benchmarking RTS*, Operations Research Spektrum **17** (1995), no. 2/3, 67–86, Postscript available at http://rtm.science.unitn.it/˜battiti/battiti-publications.html.

[14] ______, *Training neural nets with the reactive tabu search*, IEEE Transactions on Neural Networks **6** (1995), no. 5, 1185–1200, Postscript available at http://rtm.science.unitn.it/˜battiti/battiti-publications.html.

[15] ______, *The continuous reactive tabu search: blending combinatorial optimization and stochastic search for global optimization*, Annals of Operations Research – Metaheuristics in Combinatorial Optimization **63** (1996), 153–188, Postscript available at http://rtm.science.unitn.it/˜battiti/battiti-publications.html.

[16] Roberto Battiti and Alan Albert Bertossi, *Greedy, prohibition, and reactive heuristics for graph partitioning*, IEEE Transactions on Computers **48** (1999), no. 4, 361–385.

[17] Roberto Battiti and Mauro Brunato, *Reactive search for traffic grooming in WDM networks*, Evolutionary Trends of the Internet, IWDC2001, Taormina (S. Palazzo, ed.), Lecture Notes in Computer Science LNCS 2170, Springer-Verlag, September 2001, pp. 56–66.

[18] R. Bayer, *Symmetric binary B-trees: Data structure and maintenance algorithms*, Acta Informatica **1** (1972), 290–306.

[19] M. Bellare, O. Goldreich, and M. Sudan, *Free bits, PCP and non-approximability — towards tight results*, Proc. 36th Ann. IEEE Symp. on Foundations of Comput. Sci., IEEE Computer Society, 1995, pp. 422–431.

[20] J. A. Boyan and A. W. Moore, *Learning evaluation functions for global optimization and boolean satisfability*, In Proc. of 15th National Conf. on Artificial Intelligence (AAAI) (AAAI Press, ed.), 1998, pp. 3–10.

[21] Olli Brysy, *A reactive variable neighborhood search for the vehicle-routing problem with time windows*, INFORMS Journal on Computing **15** (2003), no. 4, 347–368.

[22] Rachid Chelouah and Patrick Siarry, *Tabu search applied to global optimization*, European Journal of Operational Research **123** (2000), 256–270.

[23] T.-S. Chiang and Y. Chow, *On the convergence rate of annealing processes*, SIAM Journal on Control and Optimization **26** (1988), no. 6, 1455–1470.

[24] W.C. Chiang and R.A. Russell, *A reactive tabu search metaheuristic for the vehicle routing problem with time windows*, INFORMS Journal on Computing **9** (1997), 417–430.

[25] P. Crescenzi and V. Kann, *A compendium of NP optimization problems*, Tech. report, 1996, electronic notes: http://www.nada.kth.se/˜viggo/problemlist/compendium.html.

[26] F. Dammeyer and S. Voss, *Dynamic tabu list management using the reverse elimination method*, Annals of Operations Research **41** (1993), 31–46.

[27] M. Dorigo, V. Maniezzo, and A. Colorni, *Ant system: optimization by a colony of cooperating agents*, IEEE Transactions on Systems, Man and Cybernetics, Part B **26** (1996), no. 1, 29–41.

[28] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, *Making data structures persistent*, Proceedings of the 18th Annual ACM Symposium on Theory of Computing (Berkeley, CA), ACM, May 28-30 1986.

[29] Artur Dubrawski and Jeff Schneider, *Memory based stochastic optimization for validation and tuning of function approximators*, Conference on AI and Statistics, 1997.

[30] A.E. Eiben, R. Hinterding, and Z. Michalewicz, *Parameter control in evolutionary algorithms*, IEEE Transactions on Evolutionary Computation **3** (1999), no. 2, 124–141.

[31] A.G. Ferreira and J. Zerovnik, *Bounding the probability of success of stochastic methods for global optimization*, Computers Math. Applic. **25** (1993), 1–8.

[32] Bernard Fortz and Mikkel Thorup, *Increasing internet capacity using local search*, Computational Optimization and Applications **29** (2004), no. 1, 13–48.

[33] Bennet L. Fox, *Simulated annealing: Folklore, facts, and directions*, Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing (H. Niederreiter and P. J.-S. Shiue, eds.), Springer-Verlag, 1995.

[34] L. M. Gambardella, E. D. Taillard, and M. Dorigo, *Ant colonies for the quadratic assignment problem*, Journal of the Operational Research Society **50** (1999), no. 2, 167–176.

[35] F. Glover, *Tabu search - part i*, ORSA Journal on Computing **1** (1989), no. 3, 190–260.

[36] ______, *Tabu search - part ii*, ORSA Journal on Computing **2** (1990), no. 1, 4–32.

[37] ______, *Tabu search: Improved solution alternatives*, Mathematical Programming, State of the Art (J. R. Birge and K. G. Murty, eds.), The Univ. of Michigan, 1994, pp. 64–92.

[38] L. J. Guibas and R. Sedgewick, *A dichromatic framework for balanced trees*, Proc. of the 19th Ann. Symp. on Foundations of Computer Science, IEEE Computer Society, 1978, pp. 8–21.

[39] P. Hansen and B. Jaumard, *Algorithms for the maximum satisfiability problem*, Computing **44** (1990), 279–303.

[40] J. Hastad, *Clique is hard to approximate within* $n^{1-\epsilon}$, Proc. 37th Annual Symposium on Foundations of Computer Science, IEEE Computer Society, 1996, pp. 627–636.

[41] J.N. Hooker, *Testing heuristics: We have it all wrong*, Journal of Heuristics **1** (1995), no. 1, 33–42.

[42] H. H. Hoos and T. Stuetzle, *Stochastic local search: Foundations and applications*, Morgan Kaufmann, 2005.

[43] F. Hutter, D.A.D. Tompkins, and H.H. Hoos, *Scaling and probabilistic smoothing: Efficient dynamic local search for SAT*, Proc. CP-02, LNCS, vol. 2470, Springer Verlag, 2002, pp. 233–248.

[44] D. S. Johnson, C. R. Aragon, L.A. McGeoch, and C. Schevon, *Optimization by simulated annealing: an experimental evaluation; part ii, graph coloring and number partitioning*, Operations Research **39** (1991), no. 3, 378–406.

[45] D.S. Johnson and M. Trick (Eds.), *Cliques, coloring, and satisfiability: Second DIMACS implementation challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 26, American Mathematical Society, 1996.

[46] M. J. Kearns and U. V. Vazirani, *An introduction to computational learning theory*, The MIT Press, Cambridge, Massachusetts, 1994.

[47] Rex K. Kincaid and Keith E. Laba1, *Reactive tabu search and sensor selection in active structural acoustic control problems*, Journal of Heuristics **4** (1998), no. 3, 199–220.

[48] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi, *Optimization by simulated annealing*, Science **220** (1983), 671–680.

[49] S. Lin, *Computer solutions of the travelling salesman problems*, Bell Systems Technical J. **44** (1965), no. 10, 2245–69.

[50] D. Maier and S. C. Salveter, *Hysterical B-trees*, Information Processing Letters **12** (1981), no. 4, 199–202.

[51] Catherine C. McGeoch, *Toward an experimental method for algorithm simulation*, INFORMS Journal on Computing **8** (1996), no. 1, 1–28.

[52] P. Merz and B. Freisleben, *Fitness landscape analysis and memetic algorithms for the quadratic assignment problem*, IEEE Transactions on Evolutionary Computation **4** (2000), no. 4, 337–352.

[53] Andrew W. Moore and Jeff Schneider, *Memory-based stochastic optimization*, Advances in Neural Information Processing Systems (David S. Touretzky, Michael C. Mozer, and Michael E. Hasselmo, eds.), vol. 8, The MIT Press, 1996, pp. 1066–1072.

[54] K. Nonobe and T. Ibaraki, *A tabu search approach for the constraint satisfaction problem as a general problem solver*, European Journal of Operational Research (1998), no. 106, 599–623.

[55] Ibrahim H. Osman and Niaz A. Wassan, *A reactive tabu search meta-heuristic for the vehicle routing problem with back-hauls*, Journal of Scheduling **5** (2002), no. 4, 287–305.

[56] M. H. Overmars, *Searching in the past ii: general transforms*, Tech. report, Dept. of Computer Science, Univ. of Utrecht, Utrecht, The Netherlands, 1981.

[57] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization, algorithms and complexity*, Prentice-Hall, NJ, 1982.

[58] P. M. Pardalos and J. Xu, *The maximum clique problem*, Journal of Global Optimization **4** (1994), 301–328.

[59] M. Pelikan, D.E. Goldberg, and F. Lobo, *A survey of optimization by building and using probabilistic models*, Computational Optimization and Applications **21** (2002), no. 1, 5–20.

[60] N. Sarnak and R. E. Tarjan, *Planar point location using persistent search trees*, Communications of the ACM **29** (1986), no. 7, 669–679.

[61] P. Soriano and M. Gendreau, *Tabu search algorithms for the maximum clique problem*, Tech. Report CRT-968, Centre de Recherche sur les Transports, Universite de Montreal, Canada, 1994.

[62] K. Steiglitz and P. Weiner, *Algorithms for computer solution of the traveling salesman problem*, Proceedings of the Sixth Allerton Conf. on Circuit and System Theory, Urbana, Illinois, IEEE, 1968, pp. 814–821.

[63] P. N. Strenski and Scott Kirkpatrick, *Analysis of finite length annealing schedules*, Algorithmica **6** (1991), 346–366.

[64] G. Syswerda, *Simulated crossover in genetic algorithms*, Foundations of Genetic Algorithms (D.L. Whitley, ed.), Morgan Kaufmann Publishers, San Mateo, CA, 1993, pp. 239–255.

[65] R. E. Tarjan, *Updating a balanced search tree in $O(1)$ rotations*, Information Processing Letters **16** (1983), 253–257.

[66] Sakae Toune, Hiroyuki Fudo, Takamu Genji, Yoshikazu Fukuyama, and Yosuke Nakanishi, *Comparative study of modern heuristic algorithms to service restoration in distribution systems*, IEEE TRANSACTIONS ON POWER DELIVERY **17** (2002), no. 1, 173–181.

[67] L. G. Valiant, *A theory of the learnable*, Communications of the ACM **27** (1984), no. 11, 1134–1142.

[68] Christos Voudouris and Edward Tsang, *Guided local search and its application to the traveling salesman problem*, European Journal of Operational Research **113** (1999), 469–499.

[69] T. Winter and U. Zimmermann, *Real-time dispatch of trams in storage yards*, Annals of Operations Research (2000), no. 96, 287–315.

[70] D. L. Woodruff and E. Zemel, *Hashing vectors for tabu search*, Annals of Operations Research **41** (1993), 123–138.

Local variables	
Z	Hash item (or pointer)
$lastVisit[Z]$	Time of last visit of item Z
R	Time since last visit

```
function HISTORY-REACTION (X, T)
    Z ← HASH-SEARCH (X)
    if Z ≠ null
        R ← t - lastVisit[Z]
        lastVisit[Z] ← t
        if R < 2(n-1)
            t_T ← t
            return INCREASE(T)
    else
        HASH-INSERT (X, t)
    if t - t_T > 10 k_b
        t_T ← t
        return DECREASE(T)
    return T
```

Figure 7: RLS Algorithm: routine HISTORY-REACTION.

Local variables	
L	Nodes in V not moved yet

```
procedure RESTART
    T ← 1 ; t_T ← t
    if ∃v ∈ V (lastMoved[v] = −∞)
        L ← {w ∈ V: lastMoved[v] = −∞}
        v ← random vertex with maximum deg_G(V)(v) in L
    else
        v ← random vertex ∈ V
    relevant data structures are reinitialized
    INCREMENTAL-UPDATE (v, addMove)
    X ← {v}
```

Figure 8: RLS Algorithm: routine RESTART.