Partially persistent dynamic sets for history-sensitive heuristics

R. Battiti

ABSTRACT. Effective heuristic algorithms for combinatorial problems are based on integrating local neighborhood search with *history-sensitive* schemes, where the information collected during the previous search phase is used to direct the future effort. In particular, some algorithms (like *Strict Tabu Search* and *Reactive Tabu Search*) need to detect whether a configuration has already been encountered during the previous phase of the search, either to prohibit repetitions or to determine a prohibition parameter in an adaptive way. This paper analyzes the use of *persistent dynamic sets* for storing and retrieving states and discusses the advantages of this option with respect to popular but less efficient realizations. If the search space is given by L-bit binary strings, the method complexity is $O(L)$ average-case time per search iteration when hashing is used, while the total space for a sequence of t iterations is $O(t)$.

A concrete realization of the abstract data type through a C++ class is presented. The obtained memory occupation and timing results are analyzed for selected "dictionary" benchmark tasks and for the use of the structure to support history–sensitive heuristic algorithms, in particular for the Maximum Clique problem in graphs.

Presented at the Fifth DIMACS Challenge, Rutgers, NJ, 1996. Revised: Oct 10, 1998

1. Introduction

Starting from the late eighties there has been a growing interest in *history-sensitive* heuristics that ameliorate the *Local Search* method. In particular, *Tabu Search* (*TS*) uses *prohibition-based* diversification techniques with the purpose of guiding *Local Search* beyond local optimality [**9**]. The main modifications of *Local Search* are obtained through the *temporary prohibition* (henceforth the term tabu or taboo) of selected moves available at the current point.

While ideas similar to those proposed in *TS* can be found already in the sixties, see for example the *denial* strategy of [**18**] for the Traveling Salesman problem, the current technological context, with the availability of large amounts of memory on standard computers, makes these schemes applicable for the solution of large-scale problems of applicative interest. The renaissance and full blossoming of "intelligent prohibition-based heuristics" is greatly due to the role of F. Glover [**9, 10**], but see also [**12**] for an independent seminal paper. *TS*-based algorithms have been applied

1991 *Mathematics Subject Classification.* ...

Key words and phrases. heuristic algorithms, combinatorial problems, persistent data structures, tabu search, reactive search.

to a growing number of problems in the last years, as it is witnessed by the rapidly growing number of publications. Some popular variants of *TS* algorithms will be described later.

The search space considered is given by L-bit binary strings. The main focus of this paper is on realizing mechanisms for storing *states* during the search and for checking whether a specific state has already been encountered in the past. The availability of the previous states is needed to prevent repetitions in *Strict Tabu Search* and to define in an adaptive way the suitable value of the *prohibition tenure* in *Reactive Tabu Search*, a parameter that determines the prohibited *moves*. In contrast, all previous states are not needed in *Fixed Tabu Search*, where a simple FIFO queue of space $O(L)$ is sufficient to determine the prohibited moves.

The current usage of data structures to support the history-management operations required by *Strict-TS* and *Reactive-TS* algorithms presents two sub-optimal possibilities: for a single search iteration, one method needs $O(1)$ space with a time complexity of $O(\max\{t, L\})$ [**10**], t being the current iteration; the other method needs $O(L)$ space with $O(L)$ time [**5**]. Because a large number of iterations can be executed for relevant large-size tasks (millions are not unusual) the increasing time per iteration required by the first renders the iterations progressively slower, while the memory required by the second can be excessive. In this paper we demonstrate how both goals of average time $O(L)$ per iteration, *and* $O(t)$ space for the entire sequence of t steps can be obtained by integrating *hashing* with *persistent red-black* trees.

To the best of our knowledge this is the first study to address the use of *persistent* data structure to support history-sensitive heuristics. In general, we are not aware of sound experimental studies in the area. This fact explains why comparisons with alternative approaches are lacking in the paper. Of course, this work should be extended in the future to analyze alternative implementations.

In the following sections, first the framework of history-sensitive heuristics is briefly reviewed (Sec. 2), with emphasis on *Tabu Search* principles (Sec. 2.1) and available implementations (Sec. 2.2). Then the novel use of persistent dynamic sets to support history-management in heuristics is presented (Sec. 3).

A statistical analysis of the results obtained by a realization in C++ of the proposed data structures is presented in Sec. 4. In particular the use of the structure to support the *Reactive Local Search* heuristic [**3**] is investigated.

2. Local search and history-sensitive heuristics

Let us now briefly define the main concepts and the notation related to *Local Search* and *Tabu Search*. Let $\mathcal{X}$ be the discrete search space: $\mathcal{X} = \{0,1\}^L$, and let $f : \mathcal{X} \longrightarrow R$ be the function to be minimized. In addition, let $X^{(t)} \in \mathcal{X}$ be the current configuration along the *search trajectory* at iteration t, and $N(X^{(t)})$ the neighborhood of point $X^{(t)}$, obtained by applying a set of basic moves μ_i $(1 \leq i \leq L)$, where μ_i complements the i-th bit x_i of the string: $\mu_i\ (x_1, x_2, ..., x_i, ..., x_L) = (x_1, x_2, ..., 1 - x_i, ..., x_L)$.

$$N(X^{(t)}) = \{X \in \mathcal{X} \ \text{ such that } \ X = \mu_i\ (X^{(t)}), i = 1, ...L\}$$

Local Search starts from a random initial configuration $X^{(0)} \in \mathcal{X}$ and generates a search trajectory where the next point is given by a neighbor of the current solution with a better f value. In detail, we consider a "greedy" form of *Local*

Search, where the *best* neighbor is selected, as follows:

$$Y = \textsc{Best-Neighbor}\ (\ N(X^{(t)})\ ) \tag{1}$$

$$X^{(t+1)} = \begin{cases} Y & \textbf{if} \quad f(Y) < f(X^{(t)}) \\ X^{(t)} & \textbf{if} \quad f(Y) \geq f(X^{(t)}) \quad \textbf{local minimum: search stops} \end{cases} \tag{2}$$

where BEST-NEIGHBOR selects $Y \in N(X^{(t)})$ with the lowest f value and ties are broken randomly. Y in turn becomes the new current configuration if f decreases. Clearly, *Local Search* stops as soon as the first local minimum point is encountered, when no improving moves are available.

History–sensitive heuristics act to continue the search beyond the first local minimum without "forgetting" the part of the search already executed. *Vice versa*, the *memory-less* nature is the key property characterizing Markov chains like *Simulated Annealing* or methods based on repeating runs of *Local Search* after starting from fresh random initial points. In *TS* the history–sensitivity is obtained by generating *prohibitions*: some neighbors are *prohibited*, only a subset $N_A(X^{(t)}) \subset N(X^{(t)})$ of them is *allowed*. A move shares the same prohibition status as that of the neighbor that would be obtained by applying it. The general way of generating the search trajectory that we consider is given by:

$$N_A(X^{(t)}) = \textsc{Allow}(N(X^{(t)}), X^{(0)}, ..., X^{(t)}) \tag{3}$$

$$X^{(t+1)} = \textsc{Best-Neighbor}\ (\ N_A(X^{(t)})\ ) \tag{4}$$

The set-valued function ALLOW selects a subset of $N(X^{(t)})$ in a manner that depends on the entire previous history of the search $X^{(0)}, ..., X^{(t)}$. Let us note that worsening moves can be produced (see eqn. 4), as they must be in order to exit local minima.

2.1. Tabu Search: discrete dynamical systems. There are several vantage points from which to view prohibition-based algorithms like *TS*. By analogy with the concept of *abstract data type* it is useful to separate the abstract concepts and operations of *TS* with respect to the realization with specific data structures. The terminology used in this paper reflects this distinction. Now, an essential abstract concept in *TS* is given by the *discrete dynamical system* of eqn. 3–4 obtained by modifying *Local Search*.

Possibly the simplest form of *TS* is what is called **strict-TS**: a neighbor is prohibited if and only if it has already been visited [**9**]. The term "strict" is chosen to underline the rigid enforcement of its simple prohibition rule:

$$N_A(X^{(t)}) = \{X \in N(X^{(t)}) \quad \text{such that} \quad X \neq \{X^{(0)}, ..., X^{(t)}\}\}$$

Let us note that *strict-TS* is parameter-free. Two additional algorithms can be obtain by introducing a *prohibition tenure*[1] T that determines how long a move will remain prohibited after its execution. The **fixed-TS** algorithm is obtained by fixing T throughout the search [**9**]. A neighbor is allowed if and only if it is obtained from the current point by applying a move that has not been used during the last T iterations. In detail, if $\textsc{LastUsed}(\mu)$ is the last usage time of move μ ($\textsc{LastUsed}(\mu) = -\infty$ at the beginning):

$$N_A(X^{(t)}) = \{X = \mu\ (X^{(t)}) \ \text{ such that } \ \textsc{LastUsed}(\mu) < (t - T)\} \tag{5}$$

[1] The term *prohibition tenure* is preferred with respect to the more traditional *list size* because *list size* refers to a specific implementation.

If T changes with the iteration counter t depending on the search status, the general dynamical system that generates the search trajectory comprises an additional evolution equation for $T^{(t)}$:

$$T^{(t)} = \text{T-REACT}(T^{(t-1)}, X^{(0)}, ..., X^{(t)}) \quad (6)$$

$$N_A(X^{(t)}) = \{X = \mu\ (X^{(t)}) \text{ such that } \text{LASTUSED}(\mu) < (t - T^{(t)})\} \quad (7)$$

$$X^{(t+1)} = \text{BEST-NEIGHBOR}\ (N_A(X^{(t)})) \quad (8)$$

In particular, rules to determine the prohibition tenure by reacting to the repetition of previously-visited configurations have been proposed in the **Reactive-TS** algorithm, *RTS* for short. While the reader is referred to [**5**] for the details, the design principles of *RTS* are that $T^{(t)}$ (in the range $1 \leq T^{(t)} \leq L-2$) increases when repetitions happen, and decreases when repetitions disappear for a sufficiently long search period. *RTS* has been applied to various problems with competitive performance with respect to alternative heuristics like *Fixed-TS*, *Simulated Annealing*, *Neural Networks*, and *Genetic Algorithms*. In particular, state-of-the-art heuristic results have been obtained for the Maximum Clique problem [**3**], Maximum Satisfiability [**4**], and Graph and Hypergraph Partitioning [**1, 2**].

The history-management operations required by *Reactive-TS* are X-INSERT(X, t) to insert a given configuration $X^{(t)}$ into memory, and X-SEARCH(X) to search for a given X. X-SEARCH(X) returns a reference to satellite data associated to the key X if X is present, NULL otherwise. Satellite data in *RTS* are the iteration t of the last visit and the total number of visits.

2.2. Implementations: REM, hashing and digital tree. Let us now consider some existing implementations in order to motivate the new technique. In particular let us focus onto the general requirements for storing and retrieving the past history and for establishing prohibitions. To this end, the binary string X can be represented by the set containing the positions of the bits equal to one (each index is in the range $[1, L]$). Without fear of confusion the same notation will be used for the configuration and for the corresponding set of indices. The computational complexities are worst-case unless stated explicitly.

Strict-TS has been implemented through the *reverse elimination method* (*REM*) [**10**]. Before each iteration is executed, the entire list of moves performed throughout the search is traced in *reverse* order, by starting from the latest moves applied, to determine which moves from the current $X^{(t)}$ would reproduce a previously visited configuration, i.e., which moves are prohibited at iteration t. The space-time complexities of *REM* are of $O(1)$ space and $O(\max\{t, L\})$ time *per iteration*. While the space complexity is optimal, the time complexity per iteration increases with the number of iterations t, leading eventually to $O(t^2)$ space: a fact that makes the application of "pure" *REM* problematic if many iterations are executed.

Alternative implementations are studied in [**5**], where the *hashing* and *radix tree* (or "digital tree") methods are adopted, see [**7**] for a reference textbook. In *Reactive-TS*, the *radix tree* reduces the computational cost per iteration to $O(L)$, and therefore the cost per iteration is *constant* with respect to the number of iterations, but it increases the space requirements to $O(L)$. If *hashing* is used to store the entire configuration, the *average* time complexity is again $O(L)$ while the space cost is $O(L)$ per iteration.

One is therefore confronted with two different scenarios: either constant space is used per iteration but with increasing amounts of time (*REM*), or time constant

with respect to t is used but the storage of the current configuration requires space $O(L)$ (*hashing, digital tree*). Because a very large number of iterations can be executed in practical runs of heuristics one is motivated to study schemes requiring time $O(L)$ constant with respect to t, *and* $O(1)$ space.

In this papers we assume that the answer to the queries should always be correct. It is of interest to note that a different approach consists of permitting a certain probability of *false alarms*: the dictionary structure gives a positive answer to a query about a state, while the state has never been encountered. In this second case an effective strategy already exists that stores only a hashed value [**5, 3**] associated to the state. A query gives a positive answer if and only if the hashed value of the current configuration is equal to the hashed value of a previously visited configuration. Depending on the number of bits one can bound the probability of false alarms (by bounding the probability that two different configurations have the same hashed value) and obtain an efficient realization of the memory structure for history-sensitive heuristics.

3. Persistent dynamic sets to support heuristics

The above desired time and space requirements can in fact be obtained through the use of *persistent dynamic sets*, see [**17, 8**] and references therein. Ordinary data structures are *ephemeral* [**8**] 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-sensitive heuristics considered in this paper, 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 [**8**], a paper that is dedicated to a systematic study of persistence, continuing the previous work of [**16**].

3.1. 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 INSERT(i) and 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. Red-black trees are one of the many search-tree schemes that are balanced in order to guarantee that the basic dynamic-set operations take $O(\log L)$ time. See [**6, 11**] for two seminal papers about red-black trees, and [**7**] for a recent textbook. The binary string can be immediately obtained from the tree by visiting it in symmetric order, in time $O(L)$. INSERT(i) and 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 deletion can be done in $O(1)$ rotations and $O(\log L)$ color changes [**19**]. In addition, the amortized number of color changes per update is $O(1)$, see for example [**15**].

Now, the *REM* method is closely reminiscent of a method studied in [**16**] 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 [**17, 8**]. Let us now summarize from [**17**] how a partially persistent red-black tree

can be realized. An example of the realizations that we consider is presented in Fig. 1.

The trivial way is that of keeping in memory all copies of the ephemeral tree (see the top part of Fig. 1), each copy requiring $O(L)$ space. A more efficient realization is based on *path copying*, independently proposed by many researchers, see [**17**] 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 [**17**]. 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. 1 the NULL pointers are not shown, the colors are correct only for the live tree (the nodes reachable from the rightmost root), and the 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 [**17**] is that the amortized space cost per update operation is $O(1)$. The amortized cost analysis executed in [**17**] for persistent red–black trees uses the *potential* paradigm [**20**].

Let us summarize from [**17**]. The nodes of the data structure are partitioned into two sets: *live* and *dead*. The live nodes form the version of the search tree at the current time. When changes are executed and the current time increases, live nodes can become dead and they will remain in this state forever. All nodes dead at a given time are not affected by future updates. Node-rebalancing operations are executed only on the live part of the structure. The *potential* of the structure is defined to be the number of live nodes minus $1/k$ times the number of free pointers in live nodes (k is the number of extra pointers available at each node, it will be equal to one in our case). The *amortized space cost* of an update operation is defined as the actual number of nodes created plus the net increase in potential. With this definition, if one starts with an empty data structure (with zero potential) the total amortized space cost is an upper bound on the actual number of nodes created. With the given definition of potential, copying a node has an amortized space cost of zero, because a live node with no free pointers becomes dead and

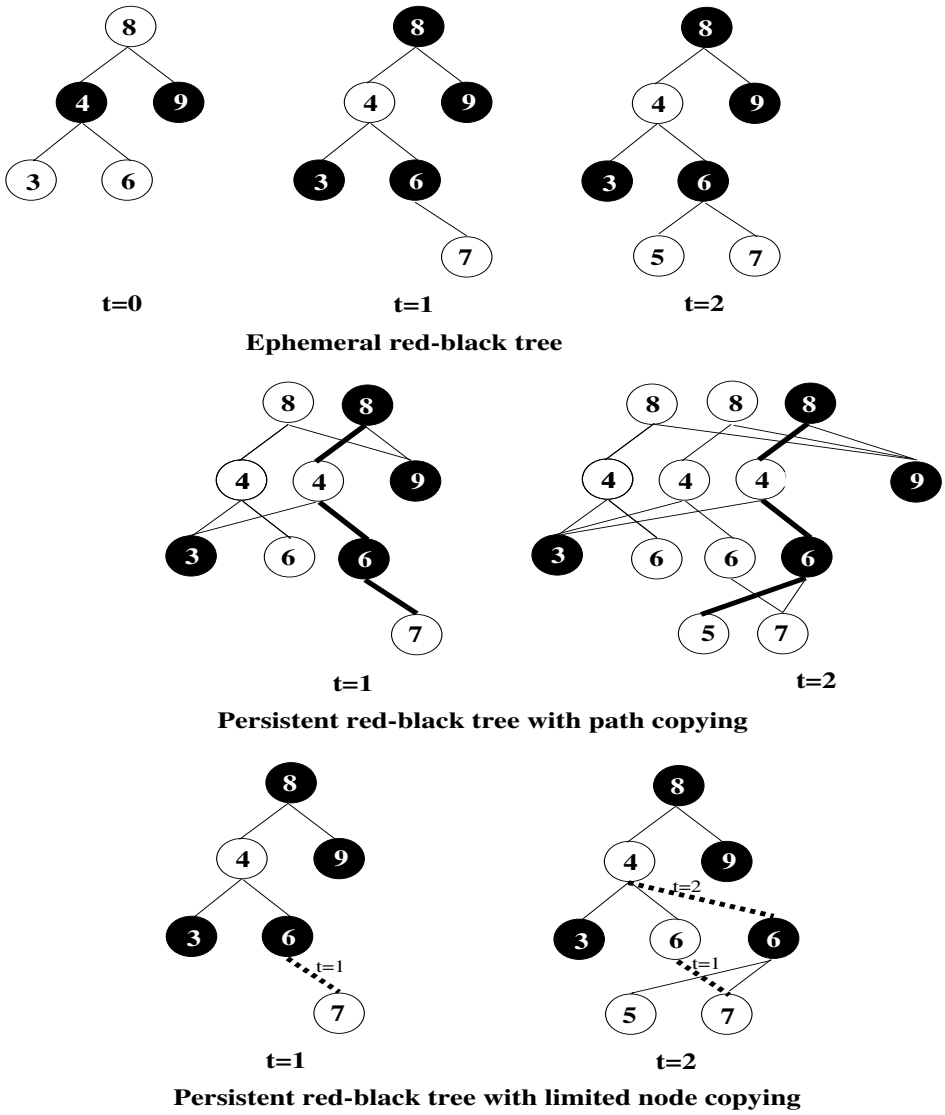

FIGURE 1. How to obtain a partially persistent red-black tree from an ephemeral one (top), containing indices 3,4,6,8,9 at t=0, with subsequent insertion of 7 and 5. Path copying (middle), with thick lines marking the copied part. Limited node copying (bottom) with dashed lines denoting the "extra" pointers with time stamp.

a new live node with k free pointers is created, for a net decrease in potential of one, balancing the new node created. Storing a new pointer in a node has an amortized space cost of $1/k$. The creation of a new node during an insertion has an amortized space cost of one. Because an insertion or deletion requires storing $O(1)$

new pointers not counting node copying, the amortized space cost of an update is $O(1)$. Additional details are present in the cited paper.

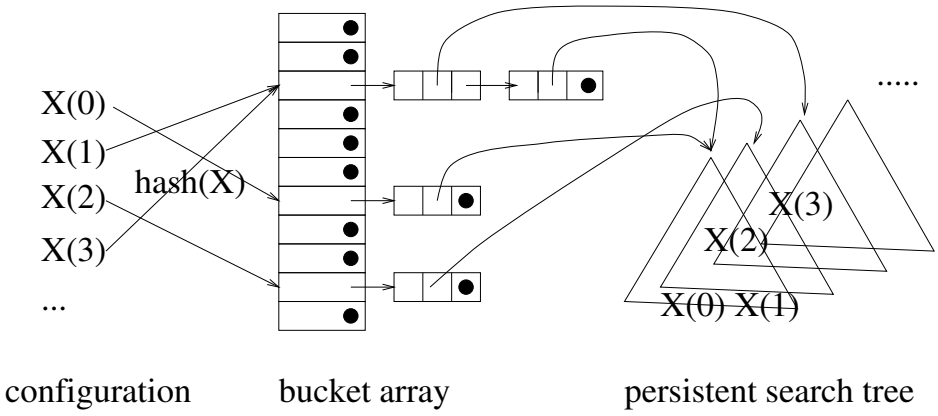

FIGURE 2. Open hashing scheme with persistent sets: a pointer to the appropriate root for configuration $X^{(t)}$ in the persistent search tree is stored in a linked list at a "bucket". Items on the list contain satellite data. The index of the bucket array is calculated from the configuration through a hashing function.

Let us now consider the context of history-sensitive 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. 2. 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 a sequence of t iterations of *Reactive-TS* requires $O(L\ t)$ average-case time and $O(t)$ space for storing and retrieving the past configurations and for establishing prohibitions.

In fact, both the hash table and the persistent red-black tree require $O(1)$ space, amortized for the tree. Let us remember that $O(1)$ amortized space per iteration implies that $O(t)$ is an upper bound on the space needed for a sequence of t iterations. 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 for all neighbors, the above complexity is optimal for the considered application to history-sensitive heuristics.

Let us note that the actual number of bytes used per iteration is very small. If the total number of iterations that will be executed by the search is not known from the beginning, a table with dynamic expansion (see [**7**], Sec. 18.4) can be used for hashing: for example by doubling its size when the load factor becomes equal to one. The space required for t iterations remains $O(t)$. Finally, let us note that the time complexity is pessimistic if the number of nonzero bits in the string is upper-bounded during the search. For example, in a Maximum Clique task, this number is less than or equal to the largest clique size [**3**], which is typically much smaller than L, the number of nodes in the graph.

4. C++ class and experiments

The partially persistent red-black tree previously described has been implemented in the C++ object-oriented language by creating a *class* called `per_rb_tree`, see Fig. 3.

Each node in the tree contains the "extra" pointer with a time stamp, as described in Sec. 3.1, and pointers to the appropriate roots of the partially persistent tree at the different times of its existence are stored in an array.

The "private" functions of the class are not described for brevity reasons. The constructor of the class `per_rb_tree(long max_roots)` takes as argument the maximum number of roots or, equivalently, the maximum life time of the structure. A default value is given if none is specified.

Let us note that the array of pointers is not needed when the pointers to the appropriate roots are stored through hashing, see Fig. 2. It is nonetheless used in the current implementation in order to study the space and time requirements of the persistent red-black tree structure in isolation.

To evaluate the different computational requirements of the ephemeral versus the persistent structure, a public function `enable_persistence()` is used to select the preferred option. The function is called when the user of the class wants to activate the persistency. Before calling the function the data structure implements an ordinary ephemeral red–black tree.

The operations `insert_key(ky_type key)` and `delete_item(per_rb_node* x)` have the usual meaning for dictionary data types: keys are inserted by creating

```
class per_rb_tree
 {
 private:
   ...
 public:
   per_rb_tree(long max_roots=1000000);
   ~per_rb_tree();

   void enable_persistence();

   void insert_key(ky_type key);

   per_rb_node * delete_item(per_rb_node* x);
   void delete_key(ky_type key);

   per_rb_node * lookup(ky_type key, long time= -1);

   void inorder_tree_walk(long time =-1);

   long size(long time =-1);
 }
```

FIGURE 3. C++ class realizing the partially persistent red-black tree.

a new node at the appropriate place in the tree and a pointer to a node is required to delete it. The operation `lookup(ky_type key, long time)` takes as additional parameter the time at which the structure is to be searched. The default value `time = -1` means that one is searching the current *live* version of the structure. The other possible values range from 1 to the current time, equal to the number of modifications (additions or deletions) executed on the structure after its creation. The operation `delete_key(ky_type k)` is realized by first calling `lookup(k)` to get a pointer and then calling `delete_item`, as in: `delete_item(lookup(k))`.

The operation `inorder_tree_walk(long time)` visits the tree at the given time. As usual, `time = -1` means that one is visiting the current structure.

All the experiments have been run on a HP 747i workstation at 100 MHz, with 128 Mb RAM. The GNU `gcc` compiler version 2.7.0 has been used with the `-O2` optimization level.

4.1. Memory usage tests: random numeric keys. In the special case of $k = 1$ the amortized space cost given in [**17**] is 6 for an insertion or deletion. It is therefore of interest to compare the above upper bound with the actual number of nodes in the tree created during its lifetime. The number of nodes is the relevant parameter in the experiments: it is then straightforward to derive the total number of bytes by multiplying that amount by the machine-dependent size of each node.

The first series of experiments is executed on the dictionary tests with random numeric keys proposed by DIMACS and collected by C. McGeoch. The `dimacs_test1` group consists of 27 dictionary trace files. These represent three random trials at each of 3 problem sizes (N=1k, 10k, 100k) with 3 kinds of problems (N inserts followed by N deletes, N inserts followed by N successful lookups, N

inserts followed by N unsuccessful lookups). The `dimacs_test2` group consists of 27 more dictionary trace files for 3 different kinds of problems (N inserts followed by N repetitions of [ins del], N inserts followed by N repetitions of [ins successful del], N inserts followed by N repetitions of [ins unsuccessful delete]). Additional details and informations on the test problems are present in Appendix A of this volume.

The average values of the total number of cells created during the lifetime of the structure (both live cells of the current version and dead cells storing the information about previous versions) are plotted in Fig. 4 and Fig. 5. The growth behavior is well captured as a first approximation by a linear function. Therefore, in order to make the figures more informative and to highlight the differences between the various experiments what is charted is the total number of cells divided by the number of changes executed on the structure (the "time" of the persistent structure). To avoid cluttering the figures with too many details, standard error bars are shown only for one plot, the error bars in the other cases are comparable.

A best fit with a linear function executed on the complete set of data on the linear scale gives the following values for the slope of the growth curve: 2.558 ($\sigma = 0.003$) , for `dimacs_test1`, and 2.541 ($\sigma = 0.001$) for `dimacs_test2`.

The general conclusion that can be derived is that the upper bound of 6 is not tight: the actual number of cells as a function of the number of changes executed grows at a much smaller rate of about 2.5 for each change. What is more of interest is to analyze in detail the behavior of the different experiments. In particular, in the top plots of Fig. 4, one observes a convergence of the curves to a value of $2.68 - 2.69$ cells per change during the insertion phase and a slow decrease toward values of $2.53 - 2.54$ during the deletion phase. A qualitatively similar behavior is present also in Fig. 5.

The fact that insertions are executed in the first part and that deletions are executed in the second one suggests two possible hypotheses: i) more cells per change are created during insertions than during deletions, ii) the number of cells generated is influenced by the size of the live part of the persistent structure. In order to test these hypotheses, the number of *new* cells created during each operation for different sizes of the live structure has been collected on the `id` files. Because the raw data are very noisy, the numbers of new cells created when the size is within a suitable interval have been averaged (e.g., for the `id.a` files, the interval is 100).

Fig. 6 shows the results. Of course, because more points are averaged when the binning interval grows, the standard error on the average decreases for the bottom plots, corresponding to the experiments with more changes executed. It can be observed that there is a significant difference in the number of new cells created for insertions and for deletions (from about 2.7 cells to about 2.4 cells). The size of the data structure also influences the number of new cells but in a counter-intuitive way: in particular less cells are generated during the first deletions, when the size of the "live" structure is larger.

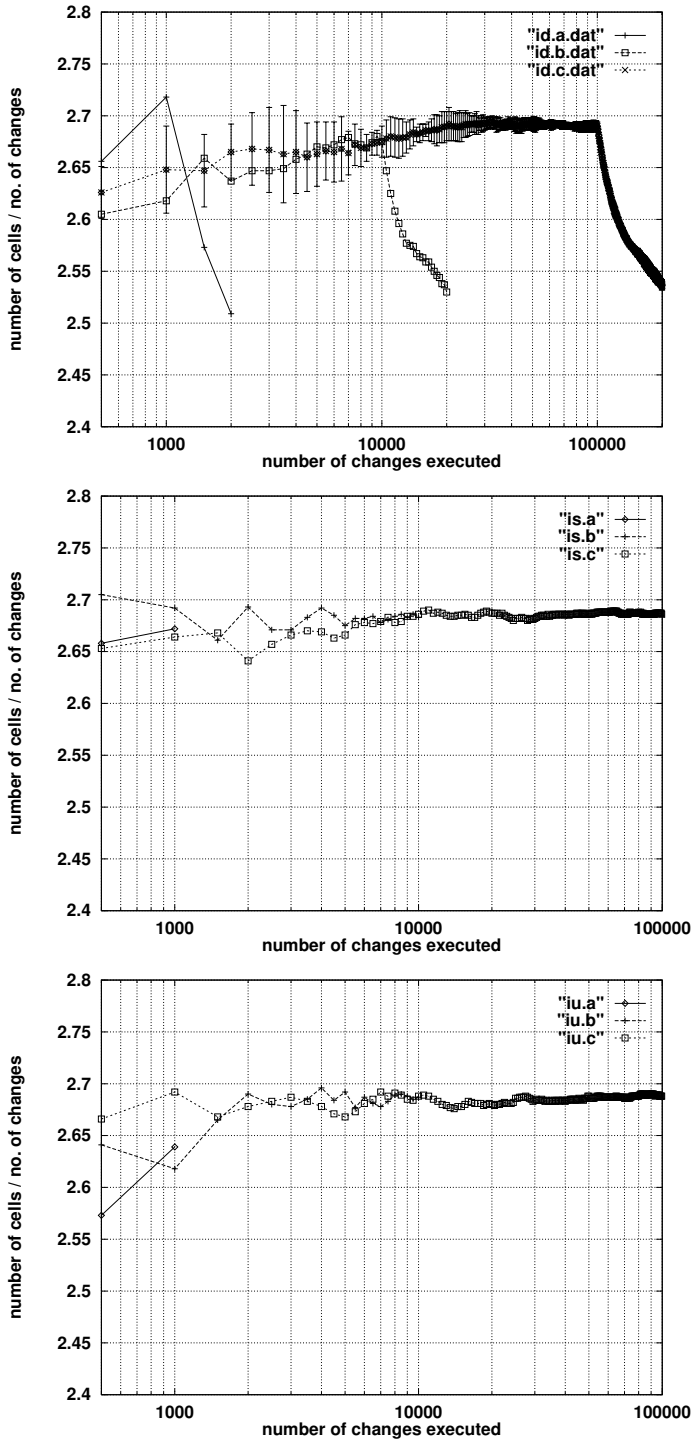

FIGURE 4. Memory usage (cells / number of changes) for DIMACS test1 dictionary tasks. `id` files (top) `is` files (middle) `iu` files (bottom).

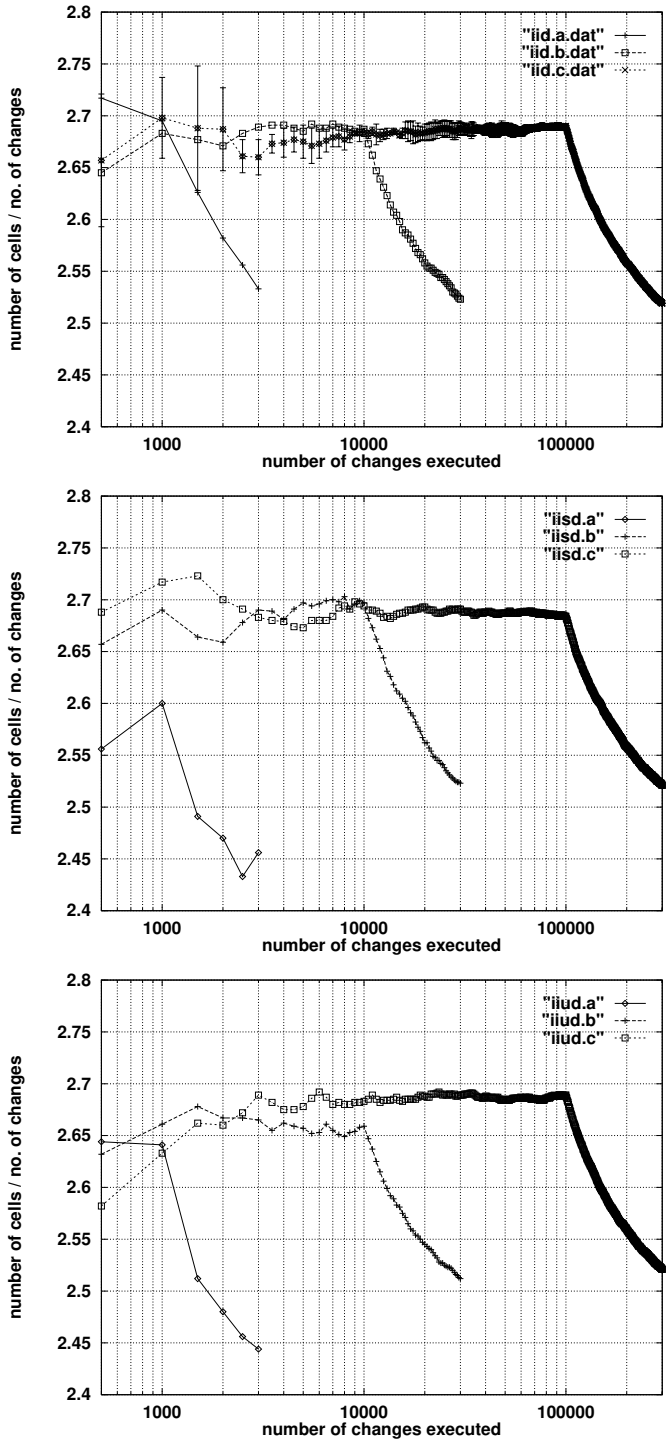

FIGURE 5. Memory usage (cells / number of changes) for DIMACS test2 dictionary tasks. `iid` files (top) `iisd` files (middle) `iiud` files (bottom).

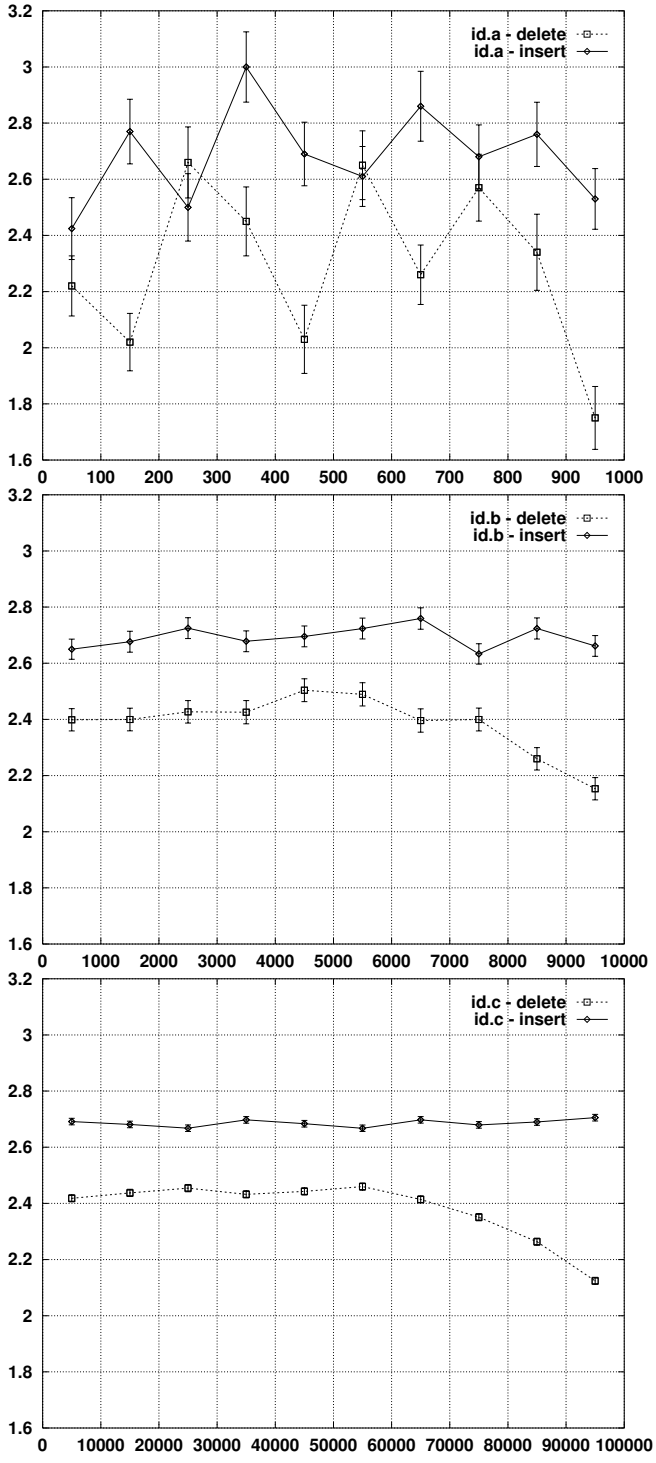

FIGURE 6. Average number of new cells created for each insert or delete operation as a function of the size of the live structure. Tests on `id.a id.b id.c` files, see text for details.

4.2. Memory usage tests: application to Reactive Search. The initial motivation for considering persistent dynamic sets was that of storing and retrieving the past history of *Local Search* based heuristic methods. In particular, we consider the case of a configuration space given by binary strings. Each binary string X is represented by a set containing the positions of the bits equal to one, where each index is in the range $[1, L]$.

The particular case considered in the experiments is that of solving the Maximum Clique problems in graphs with the *Reactive Local Search* method proposed in [**3**], to which the reader is referred for the algorithm details. Let $L = |V|$, the number of vertices. A clique is represented by a binary vector of length L in which a 1 is in position i if and only if the ith vertex is in the clique. In each run, one starts from a "seed" node and generates a search trajectory, where at each iteration a node is either added to or deleted from the current clique. The corresponding INSERT and DELETE operations are registered in a file with the format required by the DIMACS "driver" program, that is then used to test the dictionary data structure.

The memory usage results obtained by using the persistent red-black tree for different instances of the Maximum Clique problem are illustrated in Fig. 7. Standard error bars are shown only for a plot to make the figure more readable. The instances have been used in a benchmark organized by the second DIMACS implementation challenge [**13**], and they are electronically available at: `http://dimacs.rutgers.edu/Challenge/index.html`.

To better show the approximately linear growth, what is plotted is the number of cells divided by the number of changes executed on the structure. Again, it can be noted that the upper bound of 6 cells per change is not tight. Furthermore, the number of cells created after the initial transient phase shows an approximate convergence to values of 2.12 (for the random graphs with density 0.5), and 2.3 (for the random graphs with density 0.5 and for the two additional graphs).

The number of cells generated is lower than that for the dictionary tests. This fact is explained by the very small size of the current clique during the search and therefore by the very small size of the "live" part of the persistent structure. In fact, the heuristic Maximum Clique sizes found in [**3**] are in the range $14 - 16$ for the random graphs with density 0.5 (`DSJC500.5, DSJC1000.5, C2000.5`) in the range $34 - 75$ for the random graphs with density 0.9 (`C125.9, ... , C2000.9`). The `gen400_p0.9_55` graph has a clique of size 55, the `p_hat1500-2` graph has a clique of size 65.

To test the hypothesis that the reduced number of cells created is related to the small size of the live structure, the initial period of the `id` tasks has been analyzed in more detail. In particular, Fig. 8 shows that the number of new cells created during insertions grows rapidly as a function of the size of the live structure and stabilizes only after some hundreds of changes.

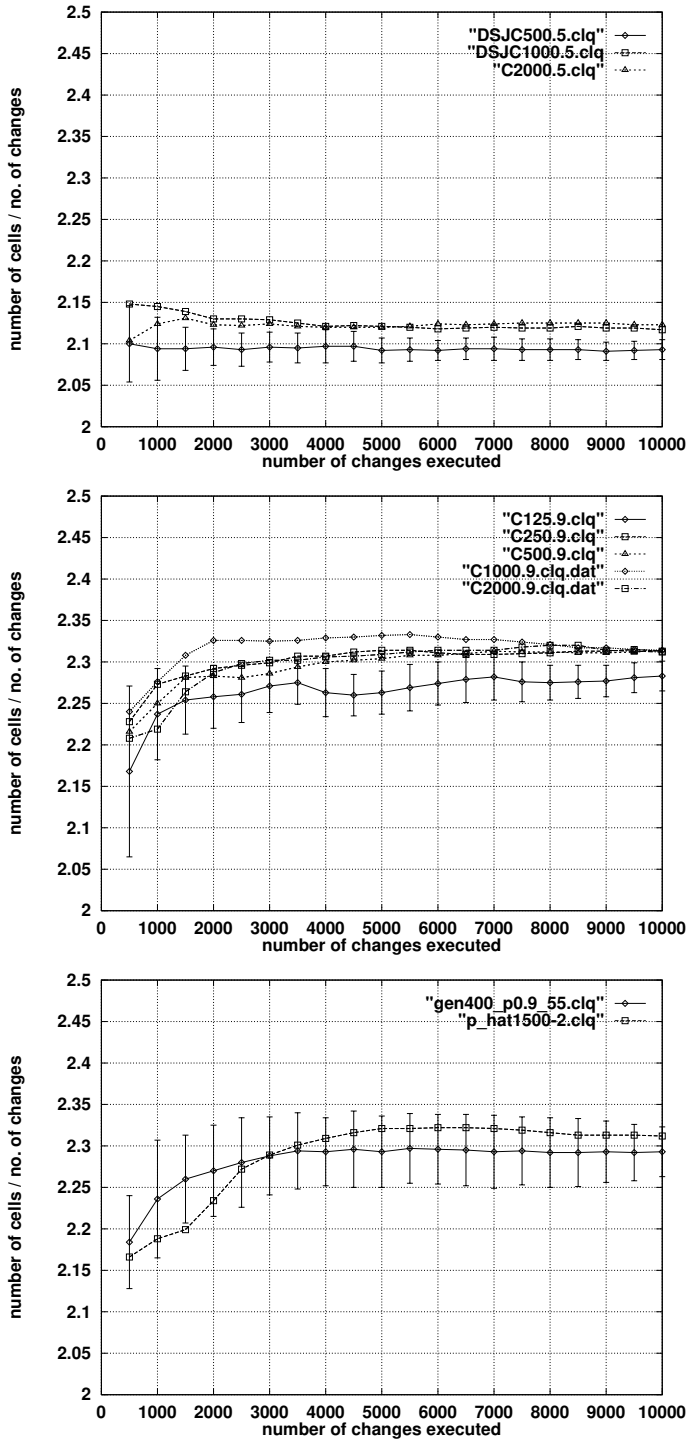

FIGURE 7. Memory usage (no. of cells) when persistent sets are used to support the Reactive *Local Search* algorithm for the MAX-CLIQUE problem.

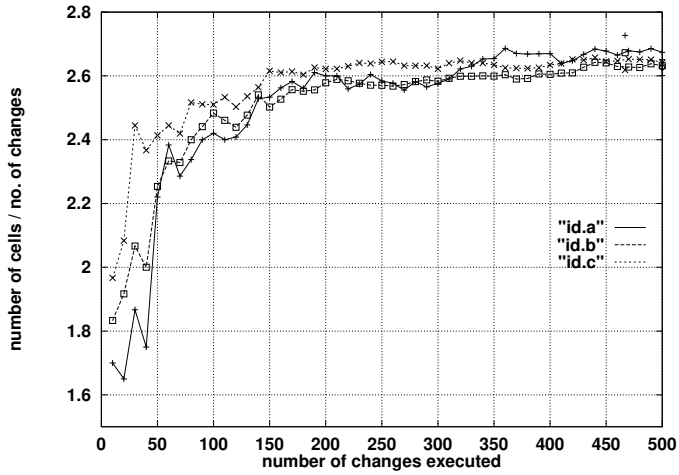

FIGURE 8. Memory usage (no. of cells / no. of changes) as a function of the number of changes: initial period for the `id` tasks.

4.3. Timing tests. Because our main interest is in the above described use of the persistent structure to support *history-sensitive heuristics*, in addition to providing timing results for a subset of the dictionary tests proposed by DIMACS (the same subset that was used in Sec. 4.1 to study the algorithm space complexity) we also provide timing results for the set of Maximum Clique problems described in Sec. 4.2.

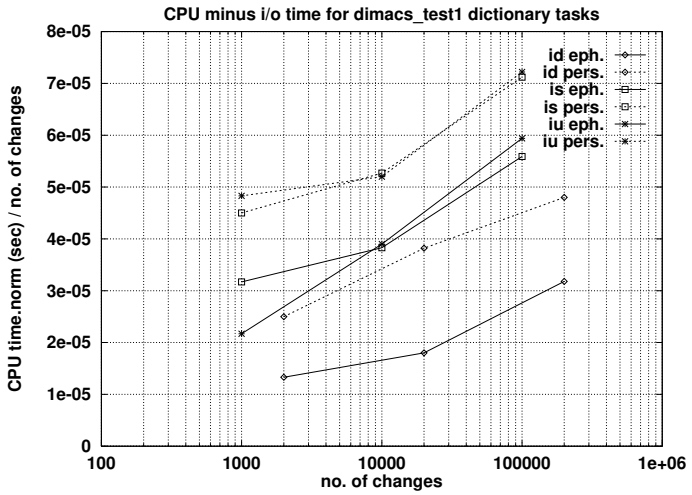

FIGURE 9. CPU time time for DIMACS test1 dictionary tasks as a function of the number of changes. Input-output time has been subtracted.

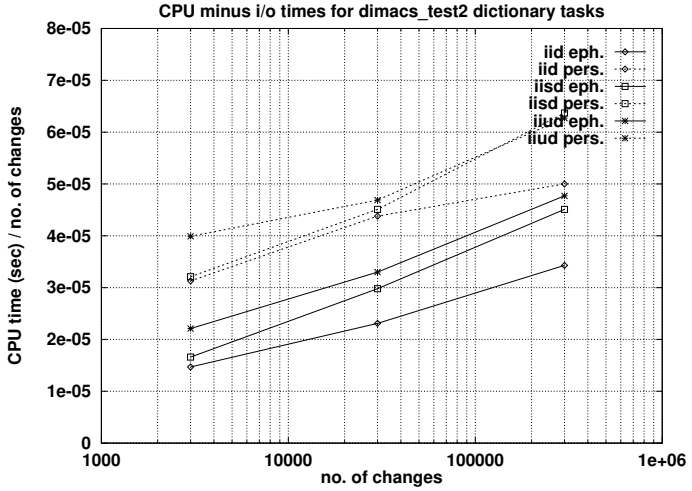

FIGURE 10. CPU time time for DIMACS test2 dictionary tasks as a function of the number of changes. Input-output time has been subtracted.

The CPU times in seconds obtained on the `dimacs_test1` and `dimacs_test2` dictionary tasks are illustrated in Fig. 9 and Fig. 10, respectively, as a function of

the total number of changes executed at the end of each run. The plots correspond to the different tests and to the two versions (ephemeral or persistent) of the red-black tree structure. In order to subtract the input/output times, separate runs are executed where the driver program is calling the same input/output operations without calling any data structure operation. The average input/output time is not negligible: it is 22.5 microseconds per operation. Therefore the individual input-output times of the different tests have been subtracted to get accurate estimates of the CPU times.

The difference in the average time for operation for the different kinds of files is significant, because of the different sequences of operations executed. Therefore, separate averages are presented for the different files. After the input/output time is discounted, the average time per operation in the ephemeral structure is 21 microseconds for the `id` files, 42 microseconds for the `is` files, 40 microseconds for the `iu` files. For the persistent structure the average times in the same order are of 37, 56, and 57 microseconds. The incremental cost to pass from an ephemeral to a persistent red-black tree for the three different kinds of files is therefore of about 76%, 33% and 42%.

The average time per operation in the ephemeral structure is 24 microseconds for the `iid` files, 30 microseconds for the `iisd` files, 34 microseconds for the `iiud` files. The standard error is less than $\sigma = 1$ in all cases. For the persistent structure the average times in the same order are of 41, 47, and 49 microseconds. The incremental cost to pass from an ephemeral to a persistent red-black tree for the three different kinds of files is therefore of about 70%, 56% and 44%.

In general, the CPU time appears to increase more rapidly than a linear function of the number of changes, both because of the size of the data structure, the nature of the changes executed and, possibly, because of cache effects. A detailed analysis of the CPU time as a function of the different parameters and sequences is beyond the scope of this paper.

Graph	ephemeral CPU	persistent CPU	(pers. CPU - eph. CPU) / (eph. CPU)
C125.9.clq.b	11.7	24.4	1.08
C250.9.clq.b	13.1	25.0	0.90
C500.9.clq.b	12.4	25.6	1.06
C1000.9.clq.b	12.4	24.6	0.98
C2000.9.clq.b	12.9	24.2	0.87
DSJC500.5.clq.b	9.8	19.1	0.94
DSJC1000.5.clq.b	9.7	21.5	1.21
C2000.5.clq.b	11.	21.0	0.90
gen400_p0.9_55.clq.b	11.8	24.6	1.08
p_hat1500-2.clq.b	13.5	24.3	0.80

TABLE 1. CPU time per change (in microseconds) for the ephemeral and persistent version on the Maximum Clique problems.

The average CPU time per change for the ephemeral and persistent version on the files produced by Maximum Clique problems (averages on 10 runs for each task) are shown in Table 1. The last column shows the increase in CPU time when passing from an ephemeral to a persistent structure, divided by the ephemeral CPU time. The CPU time is approximately doubled but it remains acceptable for the application to history-sensitive heuristics.

5. Conclusions

History-sensitive heuristics require that the previous history of a *Local–Search–*based process be stored so that it can be accessed in an efficient way. In particular, we consider search problems where the configuration space can be represented by binary strings of length L.

Partially persistent dynamic structures like the persistent red–black tree considered in this paper give an optimal solution to store the entire previous history of a sequence of t iterations with $O(t)$ space requirements. A small number of bytes per iteration, constant with respect to the iteration number and to the binary string dimension, is sufficient to store *all previous versions* of the data structure. The worst-case time complexity per update operation is $O(logL)$.

In addition, hashing can be combined with partially persistent red-black trees so that one can check whether a given configuration has already been encountered during the previous phase of the search, as it is required in *Strict Tabu Search* and *Reactive Tabu Search* algorithms. The experiments confirm the applicability of the approach and the actual average number of cells needed per update operations is much less than the theoretical upper bound of 6: in fact it is upper bounded by about 2.5 in the considered tests. The time penalty incurred in passing from an ephemeral to a persistent structure, after input-output times are discounted, is from about 30% to about 100%.

Of course, the use of persistent data structure in history–sensitive heuristic schemes based on *Local Search* is not limited to *Reactive Search* schemes, but it can be extended to all cases where the information about the previously visited configurations is used to direct the future steps of the search.

To the best of our knowledge this is the first paper to consider the use of *persistent* structures in the framework of history-sensitive heuristics. The work has shown the applicability of the approach but many issues remain open for future research. In particular, a theme of interest is to see whether tighter bounds on the number of cells created during the history of a persistent structure can be derived by considering the detailed realization. A second theme is to see whether more efficient realizations of persistent dictionaries (not limited to the red-black tree) can be derived. Different realizations will also permit to compare the adopted "red-black tree plus hashing" implementations with different data structures.

The complete algorithms developed to make red-black trees persistent are rather complex and not amenable to a brief description. Because of this, the C++ implementation of the persistent red-black trees is available from the author for research purposes.

Acknowledgements

We wish to thank prof. A. Bertossi for reading and commenting a preliminary version of this work and the anonymous referees for their detailed comments.

References

[1] R. Battiti and A. A. Bertossi, Greedy, Prohibition, and Reactive Heuristics for Graph-Partitioning, *IEEE Transactions on Computers*, in press.

[2] R. Battiti, A. A. Bertossi, and R. Rizzi. Randomized Greedy Algorithms for the Hypergraph Partitioning Problem, *Proceedings of the DIMACS Workshop on Randomization Methods in Algorithm Design*, Princeton, Oct 20, 1997. Edited by: P. Pardalos, S.. Rajasekaran, and J. Rolim. American Mathematical Society, 1998, in press

[3] R. Battiti and M. Protasi, Reactive *Local Search* for the Maximum Clique problem, Tech. Rept. TR-95-052, International Computer Science Institute, Berkeley, CA, 1995, *Algorithmica*, to appear.

[4] R. Battiti and M. Protasi, Reactive Search, a history-sensitive heuristic for MAX-SAT, *ACM Journal of Experimental Algorithmics*, **2**, Article 2. Available at URL `http://www.jea.acm.org/` .

[5] R. Battiti and G. Tecchiolli, The Reactive Tabu Search, *ORSA Journal on Computing*, **6(2)** (1994) 126–140.

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

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms* (McGraw-Hill, New York, 1990).

[8] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, Making data structures persistent, in: *Proc. 18th Ann. ACM Symp. on Theory of Computing*, Berkeley, CA (1986) 109–121.

[9] F. Glover, Tabu Search - part I, *ORSA Journal on Computing* **1(3)** (1989) 190–260.

[10] F. Glover, Tabu Search - part II, *ORSA Journal on Computing* **2(1)** (1990) 4–32.

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

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

[13] D. S. Johnson and M. A. Trick, Introduction to the Second DIMACS Challenge: Cliques, Coloring and Satisfiability, in: *Cliques, Coloring, and Satisfiability, Second DIMACS Implementation Challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 26, AMS (1996), 1–7.

[14] S. Lin, Computer Solutions of the Traveling Salesman problems, *BSTJ* **44(10)** (1965) 2245–69.

[15] D. Maier and S. C. Salveter, Hysterical B-trees, *Inform. Process. Lett.* **12 (4)** (1981) 199–202.

[16] M. H. Overmars, Searching in the past II: general transforms, Tech. Rept. RUU-CS-81-9, Dept. of Computer Science, Univ. of Utrecht, Utrecht, The Netherlands, 1981.

[17] N. Sarnak and E. Tarjan, Persistent planar point location using persistent search trees, *Communications of the ACM* **27(7)** (1986) 669–679.

[18] K. Steiglitz and P. Weiner, Some improved algorithms for computer solution of the Traveling Salesman problem, in: *Proc. Sixth Allerton Conf. on Circuit and System Theory*, Urbana, Illinois (1968), 814–21.

[19] R. E. Tarjan, Updating a balanced search tree in $O(1)$ rotations, *Inform. Process. Lett.* **16 (5)** (1983) 253–257.

[20] R. E. Tarjan, Amortized computational complexity, *SIAM J. Alg. Disc. Meth.* **6 (2)** (1985) 306–318.

(R. Battiti) Dipartimento di Matematica, Università di Trento, Via Sommarive 14, 38050 Povo (Trento) Italy

E-mail address, R. Battiti: `battiti@science.unitn.it`