Persistence, Offline Algorithms, and Space Compaction

David Eppstein

Department of Information and Computer Science
University of California, Irvine, CA 92717

Tech. Report 91-54

June 3, 1991

Abstract

We consider dynamic data structures in which updates rebuild a static solution. Space bounds for persistent versions of these structures can often be reduced by using an offline persistent data structure in place of the static solution. We apply this technique to decomposable search problems, to dynamic linear programming, and to maintaining the minimum spanning tree in a dynamic graph. Our algorithms admit trade-offs of update time vs. query time, and of time vs. space.

1 Introduction

We examine the relation between space bounds in static, offline, online, and persistent algorithms. In particular we are interested in online algorithms that reconstruct a static solution after each update. Such an algorithm can be made persistent at a large cost in space. Our main result is that a persistent algorithm can instead be based on an offline solution, resulting in greatly improved space complexity with little loss in time.

We consider data structures which handle operations of two kinds, *updates* and *queries*. In an update, an element is inserted to or deleted from a set V, maintained by the data structure. In a query, we compute the value of some function $f(x, V)$, where x is information specified by the query. A *static* problem allows no updates. In an *online* problem, updates and queries are mixed. In an *offline* problem, the sequence of updates is known in advance; updates are mixed with queries as in the online problem. We could imagine a more restrictive offline problem, in which queries as well as updates are known in advance, but we do not use such problems here.

1.1 Persistence

A *persistent* data structure is one in which updates are performed non-destructively, creating a new version of the data structure without destroying the old version. Queries specify a version of interest, and the appropriate function is computed on the information corresponding to that version, even if further updates have taken place after that version was created.

Persistence has been used for some time [2, 6, 22], and was formalized and classified by Driscoll et al. [8]. In *partial persistence*, old versions of the data structure can be queried, but only the latest version can be updated. The set of versions forms a sequence, with each update increasing the sequence length by one. In *full persistence*, old versions can be both queried and updated. This results in a tree-like history of versions. In *confluent persistence*, introduced by Driscoll, Sleator, and Tarjan [9], updates combine versions, resulting in a DAG-like history. We concentrate on full persistence.

Research in persistence has focused on automatic techniques for making non-persistent data structures persistent [3, 4, 8]. This is important both because of the application of persistence to high-level language design [8], and because in this way new data structures can be made persistent with little effort. However, these techniques can be inefficient, especially in the space they use. When a non-persistent algorithm changes a memory location, the

persistent algorithm must remember both old and new values, so space is bounded by the number of memory changes rather than the non-persistent space bounds. In particular, consider a problem in which updates are much less frequent than queries; it may be appropriate for a non-persistent solution to rebuild the data structure from scratch after each update. The persistent version of this algorithm would maintain each version of the data structure separately, and use exorbitant amounts of space.

We describe a new approach to this problem. Instead of basing our persistent algorithm on the static problem, we base it on an *offline* partially persistent algorithm. This offline problem can typically be solved in similar time bounds to the static problem, but can store many versions of the problem in a single data structure, reducing the total space complexity.

1.2 New Results

Assume we can solve an offline search problem in preprocessing time $P(n)$, query time $Q(n)$, and space $S(n)$. Then we solve the online persistent problem in update time $P(n)$, query time $Q(n)$, and space per update $O(S(n)/n)$. The precise result allows the number of updates to differ from the set size.

Using this technique, we solve any persistent decomposable search problem. Suppose the static problem can be solved in preprocessing time $P(n)$, query time $Q(n)$, and space $S(n)$. Then m updates to a persistent data structure with at most n points in any one version can be performed in time $O(P(k))$ per update, time $O((n/k)Q(k)\log k)$ per query, and space $O((m/k)S(k)\log k)$. These bounds differ from the best non-persistent algorithm by factors of $O(\log k)$ in the query and space bounds.

We next apply our persistence technique to linear programming in three dimensions. The problem set consists of linear constraints; queries describe linear objective functions to be maximized while satisfying the constraints. The static problem can be solved in preprocessing time $O(n \log n)$, query time $O(\log n)$, and space $O(n)$ [10, 20]. We describe in a separate paper algorithms that trade off the update and query times for the fully dynamic problem, and that solve offline, semi-online, and insertion-only problems in polylogarithmic time per operation [13]. We use the offline algorithm, together with an algorithm for combining several static problems, to achieve a dynamic fully persistent algorithm with update time $O(k \log\log k)$, space $O(\log k)$ per update, and query time $O((n/k)\log^2 k \log(n/k))$.

Finally we consider maintaining the minimum spanning tree (MST) in a dynamically changing graph. As usually stated, this problem has no queries,

only updates, and our persistence techniques do not apply. But we can allow queries that test, if a certain update were to occur, what change to the tree would follow, without going to the expense of making that change. The persistent version of this problem is used to find the k best spanning trees of a graph [11, 15, 17, 19]. The problem can be solved by known dynamic MST algorithms, or by sensitivity analysis [23, 24]. The latter approach, if made persistent, uses almost linear time per update, constant time per query, and $O(n)$ space per update. We solve the problem in linear time per update, and polylogarithmic query time and update space.

2 Reduction to Offline Problem

Fix a problem in which updates consist of insertions and deletions from a set. Consider offline problems in which n elements exist in the set, and $m \leq n$ updates occur. Suppose we know an algorithm that answers partially persistent queries in time $Q(n)$, takes time $P_1(n)+P_2(m)$ to build, and takes space $S_1(n)+S_2(m)$. Assume that $P_1 \leq P_2$ and $S_1 \leq S_2$, that P_i and S_i are monotonic, and that they satisfy the smoothness property

$$2F(n) \leq F(2n) \leq c \cdot F(n)$$

for some constant c. We use such an offline algorithm to solve the online fully persistent problem.

Lemma 1. *Let $m = m(n) \leq n/2$. Given an offline algorithm with time and space bounds as above, we can handle the online fully persistent problem in time $Q(n)$ per query, time $P_1(n)+P_2(m)$ per update, and space $O((S_1(n)+S_2(m))/m)$ per update.*

Proof: We use an *Euler tour* of the tree formed by the history of persistent versions; this is a traversal in which each tree node (version) is visited twice, once before and once after all of its children [25]. The first visit corresponds to the operation creating that version, and the second to the reverse update (insert instead of delete and vice versa). We partition the tour into *paths* of length between $m(n)/2$ and $m(n)$, where n is the maximum size of any version in the path. For each path, we maintain an offline data structure for the corresponding sequence of updates. Each update extends the path containing the updated version. If the path would become too long, we split it in two. We rebuild the data structure for the path or paths using the previous data structure's space. □

3 Decomposable Search Problems

We consider problems in which the updates maintain a set V, and queries can be represented as functions $f(x, V)$. Such a problem is called *decomposable* if there is some constant time binary operation $\oplus$ such that, for any partition of V into two disjoint sets $V = A \cup B$, $f(x, V) = f(x, A) \oplus f(x, B)$. Decomposable search problems have many applications in computational geometry, dynamic programming, and other areas. Research on decomposable search problems has concentrated on transforming solutions of the static problem (in which V does not change) into data structures for the dynamic problem; this line of inquiry was initiated by Bentley and Saxe [1], who defined decomposable problems and described several such methods.

Fix a decomposable search problem of interest. Suppose that, given a set with n points, we can build in time $P(n)$ a data structure of size $S(n)$ that can answer queries in time $Q(n)$. The dynamic problem (with both insertions and deletions) can be answered in the same space and query time bounds, but with an update time of $P(n)$, simply by rebuilding the static data structure after each update. This technique uses $O(S(n))$ space. By maintaining the sets in blocks of k points, and rebuilding a single block after each update, the problem can be solved with time bounds $O(P(k))$ per update and $O(nQ(k)/k)$ per query [1]. The space is $O(nS(k)/k)$, which could be a significant reduction from $O(S(n))$ if the static structure uses superlinear space.

Bentley and Saxe described another technique, for insertions only. We maintain a collection of static problems with sizes distinct powers of two. Each query combines $O(\log n)$ problems, in time $O(Q(n) \log n)$. Each update creates a new problem of size one, and then while two problems have the same size merges them to form a larger problem. Each point is in at most one problem of each size, and so the amortized insertion time is $O(P(n) \log n)$. It is not difficult to modify this algorithm to be fully persistent. For completeness we briefly describe this modification.

The difficulty with making this algorithm persistent is that an operation involving the merger of two large static problems may be repeated, at each repetition incurring a large cost. We instead amortize the cost of each merge against operations occuring *after* it in the history tree. Each operation will be charged for at most one merge at each problem size, so such repetitions can not occur.

We again maintain problems of size equal to powers of two, with now at most two problems having the same size. When a third problem of that

size is introduced, we merge the previous two into a larger problem. The merge is done at the point in the history tree at which the second problem was created; proportionally many insertions must have taken place between that point and the update causing the merge. Any repetition of the update that caused the merge will see the merge as having already happened, and not cause it to happen again. As in [21], computation of merged static data structures may be performed incrementally, to make the update times worst case rather than amortized.

We examine persistent algorithms for dynamic insertions and deletions. These algorithms could recompute the appropriate static problem as in the non-persistent algorithm, and maintain all old solutions to static problems. However this results in a space penalty: the space for the $P(n)$ update time algorithm is $O(S(n))$ in the non-persistent case, but becomes $O(nS(n))$ when we make the algorithm persistent. Following lemma 1, we use an offline algorithm to reduce this space complexity.

Lemma 2. *An offline partially persistent decomposable search problem, with m updates and n non-updated points, can be solved in preprocessing (update) time $O(P(n)+P(m)\log m)$, space $O(S(n)+S(m)\log m)$, and query time $O(Q(n)\log m)$, where there are m updates and n points in any version of the problem.*

Proof: The algorithm is similar to that of Bentley and Saxe [1] and Dobkin and Suri [7].

We compute one static problem for the non-updated points; this takes time $P(n)$ and space $S(n)$. We maintain a *segment tree* of the lifetimes of each updated point. This is a complete balanced binary tree with m leaves, corresponding to update events, ordered by the times of the events. Each point is stored at $O(\log m)$ tree nodes, which together cover all leaves corresponding to the times at which that point exists. We construct a static data structure for the points stored at each node.

Each query combines the results of the static data structure and $O(\log m)$ nodes on the path from the root of the segment tree to the leaf corresponding to the query version. Any point existing at the query time will be contained in exactly one of the queried data structures, so each query is answered correctly. □

Theorem 1. *We can perform a fully persistent sequence of insertions, deletions, and queries in a decomposable search problem, in update time*

$O(P(n))$, query time $O(Q(n)\log n)$, and space $O((S(n)\log n)/n)$ per update.

Proof: We apply lemma 1 to the offline algorithm of lemma 2, by breaking up the Euler tour of the history tree into segments of $m = O(n/\log n)$ changes, and maintaining an offline data structure for each segment. □

If $S(n) = \Omega(n^{1+\epsilon})$, we can let $m = \Theta(n)$ and reduce the space bound to $O(S(n)/n)$. Without assumption, we can reduce the update time to $O(k)$ for full persistence with insertions and deletions, at the expense of increasing the query time by a factor of $O(n/k)$. Here k is an arbitrary parameter; but for simplicity of exposition we start with the case that k is fixed.

Theorem 2. *We can perform fully persistent insertions and deletions in time $O(P(k))$ per update, time $O((n/k)Q(k)\log k)$ per query, and space $O((S(k)\log k)/k)$ per update.*

Proof: We partition the inserted points into sets, all but one of which contain k points. The remaining set, which we call the *small* set, can have fewer members. We use the algorithm of the previous section separately for each such set. To perform an insertion, we insert the point into the small set, if one exists, or create a new small set otherwise. To perform a deletion from a set that is not small, we also move a point from the small set by deleting it and re-inserting it, so that at most one set remains small. If there is no small set the set undergoing deletion becomes small. When we delete the last point from the small set we remove the set as well. □

Now allow k to be a function of n. The algorithm above depends on all sets having the same size. If that desired size changes, we do not have time to make all sets have the new size. Instead, during each update, along with the operations performed above, we move $O(1)$ points between the small set and the sets with size furthest away from k. With some smoothness assumptions, k cannot change rapidly, so all set sizes will remain close to k. Therefore we achieve the same bounds as in the theorem above. A rigorous analysis of this algorithm will be given in the full paper.

4 Linear Programming in Three Dimensions

We next apply our persistence technique to linear programming in three dimensions. Here the problem set consists of linear constraints, and queries

describe linear objective functions to be maximized by a point satisfying the constraints. We can also handle certain nonlinear problems including finding the minimum circle containing a dynamic planar point set.

The static problem can be solved in query time $Q(n) = O(\log n)$, using a linear size data structure built in time $O(n \log n)$ [10, 20]. We describe in a separate paper a method for solving queries in the intersection of k static problems, in expected time $O(k \log k \log n + \sqrt{k} \log k \log^3 n)$ [13]. We can therefore treat this problem as decomposable: the answers to k queries can not be combined, but the data structures supporting the queries can be combined, with $O(\log k)$ overhead. We can trade off between the update and query times for the fully dynamic problem, and solve offline, semi-online, and insertion-only problems in polylogarithmic time per operation, analogously to decomposable searching problems [13].

Theorem 3. *We can solve persistent dynamic three-dimensional linear programming with update time $O(k \log\log k)$, update space $O(\log k)$, and expected query time $O((n/k) \log^2 k \log(n/k))$, for any $1 \leq k \leq n/\log^3 n$.*

Proof: We divide the constraints into sets of size k, and within each set use a segment tree as in lemma 2. We will choose paths of length $m = \Theta(k/\log k)$, so we also will have unchanged constraints for each path. We divide these static constraints into $O(\log k)$ disjoint subsets of size $O(k/\log k)$. Each update modifies $O(1)$ sets, by rebuilding the segment tree and $O(1)$ subsets of static constraints. This takes time $O(k + m \log m \log\log m) = O(k \log\log k)$ [13] and uses space $O(k)$. Each time a path splits, we copy the static subsets and create $O(1)$ new subsets, in time $O(k)$.

Each query combines $O((n/k) \log k)$ static problems, containing $O(k)$ constraints each. The restriction on k lets us avoid the $O(\sqrt{k} \log k \log^3 n)$ term in the time for combining the problems. □

5 Dynamic Minimum Spanning Trees

We consider maintaining the minimum spanning tree (MST) in a dynamic edge-weighted graph. Each time an MST edge is deleted or has its weight increased, it may be removed from the MST and replaced by a different edge; each time an edge is inserted or has its weight decreased, it may be added to the MST, replacing an edge already there. We allow queries that report the effects of an update without actually performing it.

This problem can be solved in $O(\sqrt{m})$ per update [15] (here m denotes the number of edges), and therefore the same time per query. For planar graphs the time can be reduced to $O(\log n)$ [14, 18]; the same is true for offline updates [12] but the latter algorithm does not support online queries.

The persistent problem has been used in branch-and-bound algorithms for finding the k best spanning trees of a graph [11, 15, 17, 19]. The application uses $O(k)$ updates and $O(kn)$ queries, so queries should be faster than updates. The problem can be solved with persistent versions of the dynamic MST algorithms mentioned above, or by re-tracing paths through the history tree with non-persistent algorithms [15], but a better approach for the application is to calculate, after each update, the effects of all possible queries. This method is known as *sensitivity analysis*; it can be performed in almost linear time per update [23, 24], after which each query can be looked up in constant time. If made persistent, sensitivity analysis requires $O(n)$ space per update to store the replacement edges for each MST edge. The non-MST replacements would seem to take $O(m)$ space, but this can be reduced to $O(n)$ using an algorithm based on lowest common ancestors.

As before, we improve the space bounds by using an offline data structure. We find each MST edge replacement by computing a set of $O(\log n)$ *candidate* replacement edges. We test whether each candidate is not already in the MST, and whether it correctly replaces the queried edge. We select the remaining candidate with the smallest weight; this will be the smallest weight replacement in the queried version of the graph.

Our approach is similar to the update-only offline algorithm in [12]. We start with a block of updates. By removing some edges in the graph, and contracting others, we reduce the graph graph to a size proportional to the number of updates. We split the updates into two subsequences, and continue recursively. At the bottom of the recursion, problems have constant size, and can be answered by brute force. As we reduce the graph, we retain information that lets us compute and test replacement candidates corresponding to each level of the reduction.

Let the edges initially in graph G, or inserted into it by some update, be partioned into three sets, S, T, and U. U contains the edges involved in some update. T contains the MST of $G - U$. S contains the remaining edges. Call a vertex of the graph a *bystander* if it is adjacent to one of the edges in U. Call a vertex a *link* if it has degree two in T and it is not a bystander. The following facts are not difficult to prove.

1. No edge in S will ever be an MST edge.

2. If some MST edge in U is queried, it is replaced by an edge in $T \cup U$.

3. If some MST edge e in T is queried, and its replacement is an edge in S, it will be the unique such edge in the MST of $G - U - \{e\}$.

4. If $e \in T$ is not on any path in T between two bystanders, then e will always be an MST edge, and can only be replaced by an edge in S.

5. Suppose e_1, e_2, ..., e_k form a path of edges in T, connected by link vertices, with e_i having the largest weight in the path. Then for any $j \neq i$, e_j will always be an MST edge. Whenever e_i is also an MST edge, the replacement for e_j is either an edge in S or it is the same as the replacement for e_i. Whenever e_i is not an MST edge, the replacement for e_j is either an edge in S or it is e_i itself.

We compute the candidate replacement edges for fact (3) using sensitivity analysis. Facts (1-3) above show that, once this information has been collected, the edges in S play no further part in the computation, and we can safely remove them from the graph, leaving only sets T and U.

We can then contract any edge satisfying fact (4), and any edge other than e_i satisfying fact (5). Along with the candidate replacements from set S, we store for each contracted edge e_j the corresponding heavy edge e_i. Once the edges are contracted, the uncontracted edges of T form a tree in which each vertex that is not a bystander has degree at least three. The number of edges in this tree is proportional to the number of bystanders, and there are two bystanders adjacent to every updated edge; therefore we have reduced the problem on the original graph to a new problem in which the size of the graph is proportional to the number of updates.

Let G' be the graph formed by throwing out the edges in S and contracting certain edges in T, as described above. We split the sequence of updates in two, and reduce G' recursively for each subsequence.

To find candidate replacements for an MST edge e, we test whether e is in sets S, T or U above. If it is in S it can not be in the MST, and we return the empty set. If it is in U, the candidates are found recursively in G' for the subsequence of updates containing the queried version. If the edge is in T, but is not contracted, the recursive candidates are joined by the replacement in S computed by sensitivity analysis. If it is contracted by fact (4), the candidate is simply the replacement from S. If it contracted by fact (5), the candidates are the replacement from S, the heavy edge e_i on the same path, and the recursively computed replacement edges for e_i.

We test a candidate replacement (u, v) for MST edge e by checking whether e is on the MST path from u to v. If e is not contracted, the result is computed recursively in the compressed graph G'. If e is contracted by fact (4), the test can be answered with a data structure based on lowest common ancestors. If e is contracted by fact (5), let e_i be the heavy edge on the contracted path; then depending on how u and v are placed relative to the path, e is between u and v either exactly when e_i is between them or exactly when e_i is *not* between them. So in constant time we can reduce the problem to one in G', and testing each candidate takes time $O(\log n)$.

Finally, we replace non-MST edges (u, v) by the heaviest edge on the MST path between u and v. We find $O(\log n)$ candidates; these need no testing and we simply choose the heaviest. If the edge is not contracted, it will be found recursively in G'. If u and v are contracted to a common point, we can find the heaviest edge between them using lowest common ancestors. If the heaviest edge is contracted by fact (4), the candidate is the heaviest edge between u or v and the bystander into which it is contracted. If the heaviest edge is contracted by fact (5), u or v must be a link on a contracted path. Let e_i be the uncontracted, heavy edge on the path. We select as a candidate the heaviest path edge e_j in the opposite direction from e_i. If e_j is not between u and v, then e_i will be between them, and e_j will not be chosen because the true replacement will be heavier than it.

Theorem 4. *We can persistently maintain the MST of a dynamically changing graph, and answer edge replacement queries, in time $O(m)$ per update, space $O(\log n)$ per update, time $O(\log n)$ per query of a non-MST edge, and time $O(\log^2 n)$ per query of an MST edge.*

Proof: We divide the history tree into paths of $\Theta(m/\log n)$ updates. We maintain a sorted list of the edge weights involved in the versions of each path; this takes $O(m)$ space and update time per path, and allows us to compute MSTs and perform sensitivity analysis in linear time [16].

In each path, we recursively split the sequence of updates into subsequences, maintaining for each subsequence a compressed graph of proportional size, as described above. The first level of the recursion takes space $O(m)$; the remaining $O(\log n)$ levels each take space $O(m/\log n)$. The total structure can be computed in time and space $O(m)$.

Each MST replacement query traces through this recursive decomposition of update sequences and compressed graphs, in time $O(\log n)$, and produces a list of $O(\log n)$ candidate replacements. Each candidate can be tested in time $O(\log n)$, so the total time per query is $O(\log^2 n)$. □

6 Conclusions

We have developed space-efficient persistent algorithms for decomposable search problems, for dynamic linear programming, and for minimum spanning trees in dynamic graphs. Our algorithms are based on a novel reduction from the online persistent problem to an offline partially persistent version of the same problem.

Some open problems remain. Perhaps the logarithmic factors that separate our algorithms from the best non-persistent algorithms can be eliminated. Our algorithms for decomposable problems and linear programming allow tradeoffs between update and query time, while preserving low space complexity; can this be done for the MST problem?

References

[1] J.L. Bentley and J.B. Saxe. Decomposable searching problems I: static-to-dynamic transformation. *J. Algorithms* 1 (1980) 301–358.

[2] B. Chazelle. How to search in history. *Inf. & Control* 64 (1985) 77–99.

[3] P.F. Dietz. Fully persistent arrays. 1st Worksh. Algorithms and Data Structures, Springer-Verlag LNCS 382 (1989) 67–74.

[4] P.F. Dietz and R. Raman. Persistence, amortization and randomization. 2nd ACM-SIAM Symp. Discrete Algorithms (1991) 78–88.

[5] D.P. Dobkin and D.G. Kirkpatrick. Fast detection of polyhedral intersection. 9th Int. Colloq. Automata, Languages, and Programming, Springer-Verlag LNCS 140 (1982) 154–165.

[6] D.P. Dobkin and J.I. Munro. Efficient uses of the past. *J. Algorithms* 6 (1985) 455–465.

[7] D.P. Dobkin and S. Suri. Dynamically computing the maxima of decomposable functions, with applications. 30th IEEE Symp. Found. Comput. Sci. (1989) 488–493.

[8] J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan. Making data structures persistent. *J. Comput. Sys. Sci.* 38 (1989) 86–124.

[9] J.R. Driscoll, D.D. Sleator, and R.E. Tarjan. Fully persistent lists with catenation. 2nd ACM-SIAM Symp. Discrete Algorithms (1991) 89–99.

[10] H. Edelsbrunner and H. Maurer. Finding extreme points in three dimensions and solving the post-office problem in the plane. *Inf. Proc. Lett.* 21 (1985) 39–47.

[11] D. Eppstein. Finding the k best spanning trees. 2nd Scand. Worksh. Algorithm Theory, Springer-Verlag LNCS 447 (1990) 38–47.

[12] D. Eppstein. Offline algorithms for dynamic minimum spanning tree problems. Manuscript, 1991.

[13] D. Eppstein. Dynamic three-dimensional linear programming. Manuscript, 1991.

[14] D. Eppstein, G.F. Italiano, R. Tamassia, R.E. Tarjan, J. Westbrook, and M. Yung. Maintenance of a minimum spanning forest in a dynamic planar graph. 1st ACM-SIAM Symp. Discrete Algorithms (1990) 1–11.

[15] G.N. Frederickson. Data structures for on-line updating of minimum spanning trees. *SIAM J. Comput.* 14 (1985) 781–798.

[16] M.L. Fredman and D.E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. 31st IEEE Symp. Found. Comput. Sci. (1990) 719–725.

[17] H.N. Gabow. Two algorithms for generating weighted spanning trees in order. *SIAM J. Comput.* 6 (1977) 139–150.

[18] H.N. Gabow and M. Stallman. Efficient algorithms for graphic matroid intersection and parity. 12th Int. Conf. Automata, Languages, and Programming, Springer-Verlag LNCS 194 (1985) 210–220.

[19] N. Katoh, T. Ibaraki, and H. Mine. An algorithm for finding k minimum spanning trees. *SIAM J. Comput.* 10 (1981) 247–255.

[20] D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.* 12 (1983) 28–35.

[21] M.H. Overmars and J. van Leeuwen. Dynamization of decomposable searching problems yielding good worst case bounds. 5th GI Fachtagung Theoretische Informatik, Springer-Verlag LNCS 104 (1981) 224–233.

[22] N. Sarnak and R.E. Tarjan. Planar point location using persistent search trees. *C. ACM* 29 (1986) 669–679.

[23] R.E. Tarjan. Applications of path compression on balanced trees. *J. ACM* 26 (1979) 690–715.

[24] R.E. Tarjan. Sensitivity analysis of minimum spanning trees and shortest path trees. *Inf. Proc. Lett.* 14 (1982) 30–33.

[25] R.E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. 7th IEEE Symp. Found. Comput. Sci., New York (1984) 12–20.