Implementation and Experimental Evaluation of Graph Connectivity Algorithms using LEDA*

Panagiota Fatourou[†] *Paul Spirakis*[‡] *Panagiotis Zarafidis*[§] *Anna Zoura*[§]

Abstract

In this paper we describe robust and efficient implementations of two graph connectivity algorithms. The implementations are based on the LEDA library of efficient data types and algorithms [18]. Moreover, we provide experimental evaluations of the implemented algorithms and we compare their performance to other graph connectivity algorithms currently implemented in LEDA.

The first algorithm is the Karp and Tarjan algorithm [16] for finding the connected components of an undirected graph. The algorithm achieves to find the connected components of a graph $G = \langle V, E \rangle$ in $O(|V|)$ expected time. This is the *first* expected-time algorithm for the static graph connectivity problem implemented in LEDA. The experimental evaluation of the algorithm proves that the algorithm performs well in practice, and establishes that theoretical and experimental results converge.

The standard procedure provided by LEDA for finding the connected components of a graph, called COMPONENTS has running time $O(|V| + |E|)$. We have compared the performance of Karp and Tarjan's algorithm to the one of COMPONENTS and we have proved that there exists a wide class of graphs (those that they are dense) that the performance of the first algorithm dramatically improves upon the one of the second.

The second implemented algorithm is the Nikoletseas, Reif, Spirakis and Yung *polylogarithmic* algorithm [20] for dynamic graph connectivity. The algorithm can cope with any random sequence of three kinds of operations: (1)Property-Query(`parameter`): returns true if and only if the property holds; (2) Insert(x, y): inserts a new edge joining vertices x to y; (3) Delete(x, y): deletes the edge (x, y). The experimental evaluation of the algorithm proves that it is very efficient for particular classes of graphs. Comparing the performance of the implemented algorithm to the one of other dynamic connectivity algorithms implemented in LEDA, we conclude that the algorithm always performs better than all these algorithms for dense random graphs and random sequences of operations. Moreover, it works very efficiently even for sparse random graphs when the sequence of operations is long.

---

*This work has been supported in part by the European Union's ESPRIT Long Term Research Project ALCOM-IT (contract # 20244).

[†]**Contact author.** Department of Computer Engineering and Informatics, University of Patras, Patras, Greece & Computer Technology Institute, Patras, Greece. Postal address: Computer Technology Institute, P. O. Box 1122, 26110 Patras, Greece, Fax. +30 61 993 973, Tel. +30 61 960 328. E-mail: `faturu@cti.gr`

[‡]Department of Computer Engineering and Informatics, University of Patras, Patras, Greece & Computer Technology Institute, Patras, Greece. E-mail: `spirakis@cti.gr`

[§]Department of Computer Engineering and Informatics, University of Patras, Patras, Greece. E-mail: `{zarafidi, zoura}@ceid.upatras.gr`

1 Introduction

In many areas of computer science, graph algorithms play an important role: problems modeled as graphs are solved by computing a property of the graph. If the underlying problem instance changes incrementally, algorithms are needed that quickly compute the property in the modified graph. A problem is called fully dynamic when both insertions and deletions of edges take place in the underlying graph. The goal of a dynamic graph algorithm is to update the solution after a change, doing so more efficiently than re-computing it at that point from scratch. To be precise, a dynamic algorithm is a data structure that supports the following operations: (1) insert an edge e; (2) delete an edge e; (3) test if the graph fulfills a certain property, e.g. are two given vertices connected? The adaptivity requirements (insertions/deletions of edges) usually make dynamic algorithms more difficult to design and analyze than their static counterparts.

Graph connectivity is one of the most basic problems with numerous applications and various algorithms in different settings. The area of dynamic graph algorithms has been a blossoming field of research in the last years, and it has produced a large body of algorithmic techniques [4, 8, 11, 12, 15, 20]. Most of the results [8, 11, 12, 14] on efficient fully dynamic structures for general graphs were based on clustering techniques. This has led to solutions of an inherent time bound of $O(n^{\epsilon})$, for some $\epsilon < 1$, since the key problem encountered by these techniques is that the algorithm must somehow balance the work investing in maintaining the component of the cluster structure, and the work on the cluster structure (connecting the components).

Nikoletseas, Reif, Spirakis and Yung presented in [20] a fully dynamic graph connectivity algorithm with *polylogarithmic* execution time per update. Their algorithm can cope with any random sequence of three kinds of operations:

Property-Query(parameter): Returns true if and only if the property holds (or returns a subgraph as a witness to the property). For a connectivity query (u,v), a true answer means that the vertices u, v are in the same connected component.

Insert(x,y): Insert a new edge joining x to y (assuming that $\{x, y\} \notin E$).

Delete(x,y): Deletes the edge x,y (assuming $\{x, y\} \in E$).

In the previous operations it is assumed that random updates (insertions and deletions) have equal probability 1/2 and the edges to be deleted/inserted are chosen randomly. The algorithm admits an amortized expected cost of $O(\log^3 n)$ time per update operation with high probability and an amortized expected cost of $O(1)$ per query.

Another interesting work, published at about the same period, is by M. Henzinger and V. King [15]. Their work presents a technique for designing dynamic algorithms with *polylogarithmic* time per operation and applies this technique to the dynamic connectivity, bipartiteness, and cycle equivalence problem. The resulting algorithms are Las-Vegas type randomized algorithms. The connectivity algorithm achieves $O(\log^3 n)$ update time and $O(\log n)$ query time for both random and non-random sequences of updates.

The area of dynamic graph connectivity algorithms is reach of theoretical results. However, it seems that the implementation and experimental studies of this kind of algorithms are on the opposite poor. It is only recently that some of these algorithms have been implemented in C++ and tested under the LEDA Extension Package Dynamic Algorithms (LEPDGA) [17]. To the best of our knowledge, the only results towards this direction are two papers by G. Amato, G. Cattaneo and G. Italiano [2], and D. Alberts, G. Cattaneo and G. Italiano [1], respectively. In the first of these works, they implemented and tested Frederickson's algorithms [12], and compared them to other dynamic algorithms. In the second work, the dynamic minimum spanning tree based on sparsification by Eppstein, Galil, Italiano and Spencer [7] and the dynamic connectivity based on the paper by M. Henzinger and V. King [15] were tested and compared. For random inputs the algorithm by Henzinger and King is the fastest. For non-random inputs sparsification was the fastest algorithm for small sequences, while for medium and large sequences of updates, the Henzinger and King's algorithm was faster.

In this paper, we implement and experimentally evalaute two graph connectivity algorithms. The first one, namely GCOMPONENTS, is a linear expected-time algorithm for the static connectivity problem. The algorithm, proposed by R. Karp and R. Tarjan [16], achieves to find the connected components of a graph $G = (V, E)$ in $O(n)$ expected time, where $n = |V|$. It is worth pointing out that all deterministic algorithms for the same problem work in $O(n + m)$ execution time, where $n = |E|$. The only algorithm for this problem provided by LEDA is such a deterministic algorithm and thus, it has $O(n+m)$ execution time. Apparently, the theoretical results imply that GCOMPONENTS improves over COMPONENTS by an additive factor of m. Our work proves that this theoretical result can be verified experimentally. We prove through a sequence of experiments that the performance time of GCOMPONENTS on dense random graphs is extremely faster than the one of COMPONENTS.

The second work, namely NRSY, is the Nikoletseas, Reif, Spirakis and Yung algorithm for dynamic connectivity. We implement the algorithm in C++ using LEDA and evaluate its performance. More specifically, our results imply that the algorithm behaves efficiently in case of dense random graphs, as stated by theory. Comparing the performance of the implemented algorithm to the performance of other dynamic connectivity algorithms implemented in LEDA [1], we found that the algorithm performs better than those algorithms for random graphs with at least $O(n \ln n)$ edges, where n is the number of nodes in the graph. For such graphs and for random sequences of operations, our implementation is proved the fastest; it is faster even than the Henzinger and King algorithm, which has been considered the fastest implemented algorithm for such inputs thus far (see e.g., [1]). For sparser graphs, our implementation is also very efficient in case the sequence of updates is not too short.

All our implementations are written in C++ and are based on LEDA. The source codes are available at the following URL: `http://www.ceid.upatras.gr/~faturu/projects.htm`

The rest of our paper is organized as follows. Section 2 describes how Karp and Tarjan's algorithm for finding the connected components of a graph has been implemented, and provides experimental results on its performance. Section 3 concentrates on the implementation and performance evaluation of the NRSY dynamic connectivity algorithm.

2 The Karp and Tarjan's Algorithm

2.1 Algorithm Description

In this section, we present the main ideas of the randomized algorithm of Karp and Tarjan, described in [16], for finding the connected components of a graph.

Consider any graph $G = \langle V, E \rangle$ and let $n = |V|$ and $m = |E|$. The algorithm finds the connected components of G in $O(n)$ expected time. The algorithm proceeds in two stages. The first stage, called **sampling**, finds a *giant* connected sub-graph of G as follows:

(1) Let $E_0 = \emptyset$.

(2) If $|E \setminus E_0| < n$, let $D = E \setminus E_0$. Otherwise, let D be a set of n distinct edges randomly selected from $E \setminus E_0$.

(3) Let $E_0 = E_0 \cup D$.

(4) Use DFS (Depth First Search [6]) to find the connected components of $G_0 = (V, E_0)$. If $E = E_0$ or G_0 has a connected component of at least θn, were $\theta > 1/2$ is a constant, vertices, stop. Otherwise, return to Step (2).

The second stage of the algorithm, called **cleanup**, finishes the grouping of vertices into components and is executed if in the sampling stage a giant component is found. This stage uses the standard depth-first search procedure with a few small changes. Initially, all vertices in the giant component of G_0 are marked `giant`, while the remaining vertices are unmarked. An unmarked vertex is selected as a start vertex and a depth first search is initiated from this vertex. If an edge leading to the giant component is found, the search is immediately aborted and all vertices reached during the search are marked `giant`. If the search finishes without reaching the giant component, all vertices reached are marked as being in a new component. The process of carrying out such a search from an unmarked vertex is repeated until no unmarked vertices remain.

The design of the algorithm, as well as its good performance is based upon the following three observations: (1) there is a well-known depth-first search procedure [6] that finds the connected components of an undirected graph in $O(m)$ worst-case time; (2) Erdos and Renyi [9] have shown that there exists a constant $\theta > 1/2$ such that with probability tending to one as n tends to infinity, a random graph with n vertices and n edges has a giant connected component with at least θn vertices; (3) there is no need to examine all the edges of a graph in order to find its connected components; it suffices to exhaust the adjacency lists of the vertices in all but one of the components and to look at enough edges in the remaining component to verify that it is connected.

Karp and Tarjan have proved that the algorithm computes the connected components of any graph in $O(|V|)$ expected time; moreover, the probability that the running time exceeds $c|V|$, where $c > 0$ is a constant, tends to zero exponentially fast as $|V|$ tends to infinity.

2.2 Implementation

For the implementation of the algorithm, the LEDA library has been used. More specifically, the graph data structure provided by LEDA, as well as useful operations implemented on graphs (e.g., node or edge arrays, iterators, etc.) have been particularly helpful for the implementation of the algorithm.

A function, called GCOMPONENTS, has been implemented. GCOMPONENTS takes as arguments a graph $G = \langle V, E \rangle$ and a float θ, and computes the connected components of the underlying undirected graph, i.e., for every node $v \in V$, an integer `compnum`$[v]$ from $[0 \ldots$`num_of_components`$-1]$ is returned, where `num_of_components` is the number of connected components of G and v belongs to the i-th connected component if and only if `compnum`$[v] = i$. GCOMPONENTS returns a boolean, indicating the existence or not of a giant component with θn vertices in G.

The implementation of GCOMPONENTS is implied directly by the algorithm of Karp and Tarjan [16]. The two stages of the algorithm are indicated by the call of two procedures, called sampling and cleanup. Procedure sampling is called first and if a giant component of the graph G exists, cleanup is executed. The main idea of sampling is that a new graph $G_0 = (V, E_0)$ is incrementally created, by adding edges of G. Initially, E_0 is empty, while as many nodes as in G are inserted in V_0. A node array is used for direct mapping of G_0 nodes to G nodes. Procedure sampling executes a while loop. During each iteration of the loop, a set of at most $|V|$ distinct edges are inserted in E_0 and the steps of the first stage, as described in Section 2.1 are executed. The existing procedure COMPONENTS of LEDA is used for finding the connected components of G_0. Since the running time of COMPONENTS is $O(n + m)$ and G_0 has $O(n)$ edges, the running time of COMPONENTS on G_0 is $O(n)$. Procedure `cleanup` simply executes a depth-first search procedure starting from every `UNMARKED` node in order to share all remaining nodes in the proper connected components.

The code described above has been compiled and tested under the C++ compiler g++. In order to testify its correctnesss a flexible main function has been created and a lot of executions have been performed for graphs with different set of nodes and edges. The connected components calculated by GCOMPONENTS have been compared with those calculated by COMPONENTS. For graphs with small node/edge sets this comparison was straightforward. However, such comparisons can not be easily performed when the number of nodes becomes large. In this case, we used proper UNIX scripts for comparing the results produced by the two algorithms.

2.3 Performance Evaluation

The time performance of GCOMPONENTS has been compared to the one of COMPONENTS. COMPONENTS performs slightly better for sparse graphs, that is, for graphs with a small set of edges. Apparently, this is natural since if $|E| \in O(|V|)$, the $O(|V| + |E|)$ running time of COMPONENTS becomes $O(|V|)$, while additionaly the implementation of COMPONENTS is simpler than the one of GCOMPONENTS. However, GCOMPONENTS performs dramatically better than COMPONENTS for dense graphs, that is, graphs with large sets of edges (e.g., $|E| \in O(|V| \log |V|)$ or $|E| \in O(|V|^2)$).

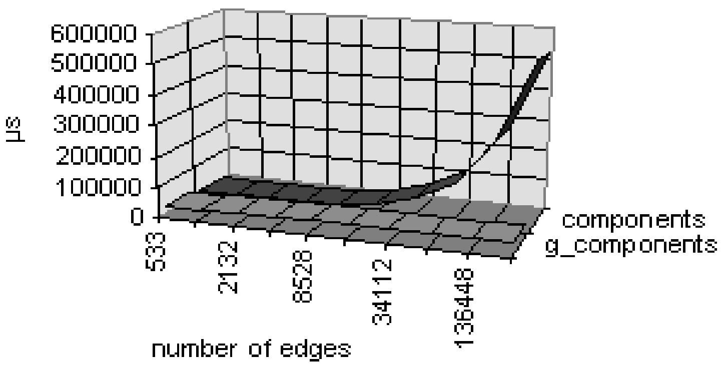

Figure 1: The execution times of GCOMPONENTS and COMPONENTS for a graph with 1600 nodes.

For example, on a graph of 200 nodes, COMPONENTS performs slightly better than GCOMPONENTS if the number of edges is less than 5000. For denser graphs GCOMPONENTS becomes better than COMPONENTS. For large graphs, e.g., graphs with 5000 nodes and at least 500000 edges, there exists a tremendeous difference on the performance of the two algorithms (for instance, the execution time of COMPONENTS is up to several seconds, while GCOMPONENTS terminates after at most a few milliseconds). For even larger/denser graphs, the execution time of COMPONENTS becomes really huge (more than a few minutes), while GCOMPONENTS behaves in an extremely faster way.

The experimental evaluation of GCOMPONENTS implies that for graphs with fixed number of nodes, the algorithm has almost the same running time independent of the number of edges in the graph. Thus, the experiments verify the theoretical result that the running time of GCOMPONENTS does not depend on the number of edges in the graph. Another interesting observation is that the performance of GCOMPONENTS is linearly increasing with n, as stated by the theoretical results.

Figure 1 presents the performance of both algorithms for a graph of 1600 nodes[1]. All our experiments took place on an `ULTRA ENTERPRISE 3000 SUN SPARC` station with two processors working at 167`MHz` with 256 Mbytes of memory.

[1] More diagrams demonstrating the above observations are provided at the attached appendix.

3 The NRSY Algorithm

3.1 Description

In this section, we describe the dynamic connectivity algorithm of Nikoletseas, Reif, Spirakis and Yung (NRSY), presented in [20].

The algorithm alternates between two epochs, namely the *activation* epoch and the *retirement* epoch. As long as the algorithm is in the activation epoch, each operation is performed in at most polylogarithmic time. Queries are performed in constant time. However, in order to perform these operations fast, the algorithm may examine many edges at each delete operation, which means that after a small number of delete operations the algorithm has probably looked at the entire graph in which case it cannot use the fact that the graph is random. If this happens, the algorithm switches to the retirement epoch, where any dynamic connectivity algorithm presented in the literature, with expected update time at most $O(n)$ should be used for performing a specific number of operations, probably in a higher cost than a polylogarithmic one. However, the number of operations performed during a retirement epoch is not too large, so that the amortized cost per operation of the algorithm has been proved to be polylogarithmic.

Initially, the algorithm uses a slightly modified version of the linear expected-time algorithm for finding connected components by Karp and Tarjan [16] (see Section 2) to compute a forest of spanning trees for the original graph. We call the above procedure of computing the forest of spanning trees a *total reconstruction*. We say that a total reconstruction is *successful* if it achieves to find a giant component of the graph. It is worth pointing out that a total reconstruction is an expensive operation, since all data structures of the algorithm should be re/initiated.

A graph activation epoch is entered after a successful total reconstruction. The activation epoch lasts at most $c_1 n^2 \log n$ operations, where $c_1 > 1$ is a constant. However, the epoch may end before all these operations are performed. This happens when a deletion operation disconnects the spanning tree of the giant component and the attempted fast reconnection fails. A graph retirement epoch starts when the previous graph activation epoch ends. However, the algorithm may start by a retirement epoch if the initial total reconstruction is not successful. During the course of an execution of a retirement epoch, the algorithm does not maintain any data structure. It simply uses some other algorithm for dynamic connectivity (any algorithm presented in the literature with update time at most $O(n)$) to perform the operations. A graph retirement epoch lasts at least $c_2 n \log^2 n$ operations, where $c_2 > 1$ is a constant. At the end of a graph retirement epoch, a total reconstruction is attempted. If it is successful, a new graph activation epoch is entered. Otherwise, the graph retirement epoch continues for another set of $c_2 n \log^2 n$ operations. The above procedure is repeated until a successful total reconstruction occurs.

During an activation epoch the algorithm maintains a forest of spanning trees, one tree per connected component of the graph. Moreover, the algorithm categorizes all edges of the graph into three classes: (1) pool edges, (2) tree edges, and (3) retired edges.

During an activation insertion operation, if the edge joins vertices of the same tree, the edge is marked as a retired edge and the appropriate data structures are updated; otherwise, the edge joins vertices of different trees, and the component name of the smaller tree is updated. During an activation deletion operation, if the edge is a pool or a retired edge, the edge is deleted by all data structures. Otherwise, if the edge is a tree edge of a small tree and the deletion of the edge disconnects the corresponding graph component, the tree is split into two small trees and the smaller of the two trees is relabeled. If on the oposite, a reconnection is possible, the tree is reconnected. If the edge is an edge of the tree of the giant component, a specific procedure, called NeighborhoodSearch, is executed.

Let the deleted edge be $e = \langle u, v \rangle$. Moreover, let $T(u)$, $T(v)$ be the two pieces in which the tree is split by the removal of e, such that $u \in T(u)$ and $v \in T(v)$. NeighborhoodSearch executes a sequence of phases. A phase starts when a new node is visited. NeighborhoodSearch starts two BreadthFirstSearch procedures, the one out of u and the other out of v and executes them in an interleaved way; that is, the algorithm visits nodes by executing one step by each BFS. The visit of a node, independently of which of the two searches reaches the node, indicates the start of a new phase. In any phase an attempt for reconnection occurs. If this reconnection is achieved, the phase returns success; otherwise, it returns failure. The total number of nodes visited by BFS procedures equals $c_3 \log n$, where $c_3 > 1$ is a constant. After this number of nodes have been visited, NeighborhoodSearch ends, independently of whether it achieves to reconnect the giant tree or not. If any phase returns success, NeighborhoodSearch also returns success. If more than one phases returns success, the edge which is "closer" to the root is selected for the reconnection. Thus, NeighborhoodSearch may: (1) finish with no success, in which case the algorithm undergoes total reconstruction and the current activation epoch ends. NeighborhoodSearch reports failure in this case; (2) the nodes connected to u may be exhausted before the search finishes, so that the one of the two components is a small component of $O(\log n)$ nodes; let this small component be $G(u)$. We search all edges emanating from nodes of $G(u)$ and if a reconnection is impossible, the giant component is just disconnected to a still-giant component and a midget component of $O(\log n)$ nodes. In this case the midget component is renamed, while NeighborhoodSearch reports success; (3) several phases may report a number of successes. In this case, the algorithm chooses for reconnection the edge which is "closer" to the root of the tree and reconnects the two pieces.

We now describe what happens during a phase initiated by the visit of some node w. Assume that w is reached due to the execution of the BFS initiated at node u, so that $w \in T(u)$. Roughly speaking, the algorithm checks if a random chosen (if any exists) pool edge out of node w reconnects the two pieces of the giant tree and if this reconnection is a "good" one. A good reconnection is one that does not increase the diameter of the tree. If a good reconnection is found, procedure Phase reports success; otherwise, it reports failure.

Graph activation epochs are partitioned into edge deletion intervals of $c_4 \log n$ deletions each, where $c_4 > 1$ is a constant. All edges marked as retired during an edge deletion interval are returned into the pool after a delay of one more subsequent edge deletion interval. Edges of small components are not reactivated by the algorithm.

Nikoletseas, Reif, Spirakis and Yung have proved that the amortized expected time of algorithm NRSY for update operations on the graph is $O(\log^3 n)$ with high probability; moreover, they have proved that the query time of their algorithm is $O(1)$.

3.2 Implementation

The algorithm has been implemented as a data structure (a class, called `nrsy_connectivity`, in C++). For its implementation the Library of Efficient Data types and Algorithms (LEDA) has been used. The public methods of class `nrsy_connectivity` that are efficiently supported are: `insert_new_edge`, `delete_edge` and all queries ("global" or not). A few more public methods are provided for collecting statistics, as well as for demonstrating how the algorithm works (that is, a graphical user interface).

In the activaton epoch, each node maintains both a set of pool edges and a priority queue of retired edges incident to it. Each node is labeled by the identification of the component it belongs.

A forest of spanning trees is maintained during the execution of activaton epochs of the algorithm. Each node of the tree maintains a pointer to its parent, its left and right children (pointers `lc`, `rc`, respectively), as well as to its left and right siblings (pointers `ls`, `rs`, respectively). If a node does not have children pointers `lc`, `rc` are `nil`. If a node does not have right or left sibling, the corresponding pointer points to the node's parent. The trees are constructed in $O(n)$ time by using a slightly modified version of Karp and Tarjan's algorithm, presented in Section 2. For random dense graphs, these trees have diameter of expected length logarithmic on the number of nodes in the graph.

Clearly, the way that the nodes of each tree are connected allows fast traversal of parts of the tree (which is required e.g., for updating labels during insert/delete operations). It is worth pointing out that an inserted edge that connects nodes of two different trees may cause an *evert* operation to take place in one of the two trees, so that some particular node to become the new root of this tree. In this case, all nodes starting from the new root up to the old root of the tree should be updated. However, since the diameter of the tree is of logarithmic length, such operations can be performed in logarithmic time.

Each node maintains the size of its subtree (that is, the subtree emanated by the node). When insertions/deletions that change the forest of the spanning trees occur, the smallest of the two trees is updated (e.g., the size and the label of the nodes of the smallest tree are updated).

Retired edges are maintained in the retired priority queues of the nodes that they are incident. A counter measuring the number of operations in each epoch is maintained and when an edge becomes retired, its priority takes the value of this counter. Reactivation of retired edges occurs only before the deletion of *tree* edges, that is, edges that disconnect one of the spanning trees in the forest. More specifically, reactivations occur only before the execution of procedure NeighborhoodSearch, since for this procedure it is important several pool edges to exist. Reactivation is performed if the difference of the priority of a retired edge from the current value of the counter expressing the number of operations performed in the current epoch is larger than $\log n$.

Algorithm	Insert	Delete	Query
`nrsy_connectivity`	$O(\log^3 n)$	$O(\log^3 n)$	$O(1)$
`dc_henzinger_king`	$O(\log^3 n)$	$O(\log^3 n)$	$O(\log n)$
`dc_simple`	$O(n+m)$	$O(n+m)$	$O(1)$

Figure 2: Theoretical bounds on the performance of each algorithm

During the retirement epoch, the algorithm by Henziger and King (that is, the data structure `dc_henzinger_king` of LEP-DGA [17]) is used for performing each operation.

The correctness of the implementation has been checked by comparing the answers to the queries in sequences of random operations among NRSY and other already implemented algorithms in LEDA. Additionally, we have performed tests with random updates, and every time a specified number of updates had been done, we checked whether all possible queries were answered according to the component labels of the vertices. These labels were computed by the static algorithm COMPONENTS or GCOMPONENTS.

3.3 Performance Evaluation

In this section, we present the results of comparative experiments with NRSY and other algorithms currently implemented in LEDA, using different types and sizes of inputs. More specifically, we measure the performance of the following dynamic connectivity data structures: `nrsy_connectivity`, `dc_henzinger_king` and dc_simple. The second is the algorithm by M. Henzinger and V. King [1, 15], while the third is a simple approach to dynamic connectivity [17]. Algorithm dc_simple maintains and recomputes if necessary component labels for the vertices and edge labels for the edges in a current spanning forest. Inserting or deleting edges which do not belong to the current spanning forest takes constant time, while the deletion/insertion of forest edges takes time proportional to the size of the affected component(s) (thus, $O(|V|+|E|)$ in the worst case). Figure 2 summarizes information about the performance of the above three algorithms. The last two algorithms have been implemented in LEDA and their performance has been studied experimentally in [1].

Since NRSY algorithm guarantees good performance only on random inputs, we concentrate our study only on random sequences of updates on random graphs. We are mainly interested in the average case performance of the algorithm on such inputs. The random inputs consisted of random graphs with different densities, and sequences of random updates.

We conducted a series of tests for graphs on $n = 50, 100, 300, 500, 750$, and 1000 vertices and different sizes of edge sets. More specifically, for each such n we tested the performance of the above algorithms on graphs with $m = n/2$, n, $n \ln n$, $n^{3/2}$ and $n^2/4$ edges. For every pair of values for n and m we did five experiments and took averages. In every experiment, the same input was used for all algorithms. Since the NRSY algorithm may executes several total reconstructions during its execution, we didn't measure seperately the times that the algorithm spent in preprocessing, that is, for building the data structure for the initial graph, and processing, that is for performing the sequence of operations. We measure the performance of the algorithms for long update sequences e.g., of 100000 operations, for medium sequences e.g., of 10000 operations and for short sequences e.g., of 1000 operations. We did all experiments on an `ULTRA ENTERPRISE 3000 SUN SPARC` station with two processors working at 167`MHz` with 256 Mbytes of memory.

The most difficult inputs for the NRSY algorithm are inputs with m equal or slightly larger to n. In this case, the theory of random graphs implies that the graph is with high probability disconnected, but it may exist some component of size in the order of $n^{2/3}$. Thus, a total reconstruction with high probability does not succeed to find a giant component, or if it finds one it is very highly expected that it will become disconnected in the next few operations. Thus, either the algorithm does not run in activation epoch, or it gets in activation epoch and after a few operations it has to run a total reconstruction again. Recall that a total reconstruction is an expensive operation. Even if the algorithm gets and remains in the activation epoch for some operations, almost all deletions/insertions are expected to be tree updates. Thus, in this case most of the operations to be performed are expensive.

Difficult inputs appear also to be those where $m < n$. In this case, the theory of random graphs implies that with high probability the graph consists of several components of size $O(\log n)$. Thus, with very high probability a total reconstruction will not succeed. At this point the algorithm uses the `dc_henzinger_king` data structure to execute the retirement epoch. If a total reconstruction succeeds after the retirement epoch the algorithm gets into an activaton epoch and so on. We expect that `nrsy_connectivity` and `dc_henzinger_king` perform in a similar way in this case. However, `nrsy_connectivity` periodically performes a total reconstruction and checks if a giant component exists in the graph. Since a total reconstruction is expensive, it would be also natural if `nrsy_connectivity` behaves in a slightly slower way than `dc_henzinger_king` in this case.

For inputs with $m > n$, that is, if $m \in O(n \ln n)$ or $m \in O(n^2)$, the theory of random graphs implies that a giant component exists in the graph, so that `nrsy_connectivity` should spend most of its time in activation epochs. Moreover, most of the updates are non-tree updates, so that `nrsy_connectivity` should perform them very fast (in $O(1)$ time). We further expect that the behavior of the algorithm should be better for long/medium sequences of operations than for small sequences, since the theoretical results hold asymptotically.

The experiments that we perform prove all the above observations. More specifically, for long sequences of updates, our implementation is proved to be very fast, faster than the Henzinger and King's algorithm in most cases. For medium sequences of operations our implementation has similar performance to the one of `dc_henzinger_king` for values of m close to n, while it is better for denser graphs. For short sequences of updates, the performance of our implementation becomes worse than the other two algorithm for sparse graphs. Recall that this is expectable, since the algorithm alternates between retirement epochs and total reconstructions, while all operations performed are with high probability expensive operations. Notice that this bad behavior of the algorithm in this case does not contradict the theoretical results, since all bounds provided in [20] have been derived assuming dense random input graphs. Moreover,

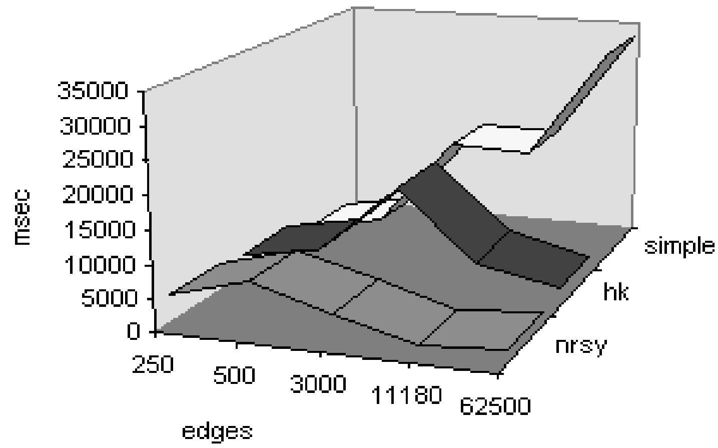

Figure 3: Performance of NRSY, Henzinger-King and Simple for a random graph of 500 nodes and a sequence of 10000 random operations.

even in the case of short sequences, our implementation has better performance than the other two algorithms for dense graphs (that is, if m is at least $O(n \ln n)$).

For the behavior of the other two algorithms we derive similar observations as those presented in [1]. Roughly speaking, `dc_henzinger_king` benefits for small components as they appear for $m < n$; moreover, the most difficult inputs are again those for which m is close to n, while the algorithm gets better and better if the number of edges increases (still remaining worse than `nrsy_connectivity` though). For `dc_simple`, if m increases the expected number of tree updates decreases, but each such tree update becomes very expensive. In theory, these two effects cancel out, so that it appears that the expected running of `dc_simple` is $O(n)$ per update.

The performance of `dc_henzinger_king` has been compared in [1] to the one of a sequence of other algorithms, among others sparsification [8], and it has been proved that `dc_henzinger_king` is the fastest algorithm among all for random inputs. Since NRSY performs better than Henzinger and King's algorithm for such inputs (especially if the graph is dense) we conclude that `nrsy_connectivity` is the fastest algorithm for random sequences of updates on dense random graphs, implemented under LEDA thus far. Moreover, the performance of NRSY is very good (comparable or better than the one of Henzinger and King's algorithm) even for non-dense random graphs, if the sequence of updates is long.

We didn't test the performance of `nrsy_connectivity` for non-random inputs, since we believe that the results would not be of theoretical interest. Recall that theory does not guarantee any bound on the update time of `nrsy_connectivity` for non-random inputs on non-random graphs. Some of the above observations are illustrated in Figure 3[2].

[2] More diagrams are provided at the attached appendix.

4 Discussion

We have described elegant, robust and efficient implementations of two graph connectivity algorithms, based on the LEDA C++ library of efficient data types and algorithms. Moreover, we have shown with experimental data that these algorithms are fast and efficient in practice.

We are currently working on testing the performance of modified versions of the NRSY algorithm. One such direction is, for example, to let the algorithm remain in the activation epoch till the giant component is disconnected, instead of alternating between the two epochs after the execution of a specific number of operations. We believe that such modifications may yield better experimental results.

References

[1] D. Alberts, G. Cattaneo and G. Italiano, "An Empirical Study of Dynamic Algorithms," *ACM Journal of Experimental Algorithmics,* Vol. 2, No. 5, pp. 192–201, 1997.

[2] G. Amato, G. Cattaneo and G. Italiano, "Experimental Analysis of Dynamic Minimum Spanning Tree Algorithms," *Proceedings of the 8th Annual ACM–SIAM Symposium on Discrete Algorithms,* pp. 314–323, January 1997.

[3] G. Amato, G. Cattaneo, G. F. Italiano, U. Nanni and C. Zaroliagis, "A Software Library of Dynamic Graph Algorithms," *Proceedings of the Workshop on Algorithms and Experiments (ALEX'98),* pp. 129–136, February 1998.

[4] D. Alberts and M. Henzinger, "Average Case Analysis of Dynamic Graph Algorithms," *Proceedings of the 6th Annual ACM–SIAM Symposium on Discrete Algorithms,* pp. 312–321, 1995.

[5] Bela Bollobas, *Random Graphs,* Academic Press, 1985.

[6] T. Cormen, C. Leiserson and R. Rivest, *Introduction to Algorithms,* MIT Press/McGraw-Hill Book company, 1990.

[7] D. Eppstein, Z. Galil, G. Italiano and T. Spencer, "Separator Based Sparsification for Dynamic Planar Graph Algorithms," *Journal of Computer and System Science,* Vol. 52, No. 1, pp. 3–27, 1996.

[8] D. Epstein, Z. Galil, G. Italiano and A. Nissenweig, "Sparsification - A Technique for Speeding Up Dynamic Graph Algorithms," *Journal of ACM,* Vol. 44, pp. 669–696, 1997.

[9] P. Erdos and A. Renyi, "On the evolution of random graphs," Magyar Tud. Akad. Math. Kut. Int. Kozl., 5, pp. 17-61, 1960.

[10] P. Fatourou, P. Spirakis, P. Zarafidis and A. Zoura, "Implementation and Experimental Evaluation of Graph Connectivity Algorithms using LEDA," `http://students.ceid.upatras.gr/~faturu/projects.htm`

[11] G. Frederickson, "Data Structures for On-Line Updating of Minimum Spanning Trees," *SIAM Journal of Computing,* Vol. 14, pp. 781–798, 1985.

[12] G. Frederickson, "Ambivalent data structures for dynamic 2-edge connectivity and k-smallest spanning trees," *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science (FOCS'91),* pp. 632-641, 1991.

[13] D. Frigioni, T. Miller, U. Nanni, G. Pasqualone, G. Schaefer and C. Zaroliagis, "An Experimental Study of Dynamic Algorithms for Directed Graphs," *Proceedings of the 6th Annual European Symposium on Algorithms (ESA'98),* pp. 368–380, August 1998.

[14] Z. Galil and G. Italiano, "Fully dynamic algorithms for edge-connectivity problems," *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing (STOC'91),* pp. 317–327, May 1991.

[15] M. Henzinger and V. King, "Randomized Graph Algorithms with Polylogarithmic Time per Operation," *Proceedings of the 27th Annual ACM Symposium on Theory of Computing (STOC'95),* pp. 519–527, 1995.

[16] R. Karp and R. Tarjan, "Linear Expected Time for Connectivity Problems," *Journal of Algorithms,* Vol. 1, pp. 374–393, 1980.

[17] "LEDA Extension Package Dynamic Graph Algorithms", User Manual, Version 3.8, 1998.

[18] K. Mehlhorn and S. Naher, "LEDA, A platform for combinatorial and geometric computing " *Communications of the ACM*, 1995.

[19] K. Mehlhorn and S.Naher, *The LEDA Platform of Combinatorial and Geometric Computing,* Cambridge University Press, 1999.

[20] S. Nikoletseas, J. Reif, P. Spirakis and M. Yung, "Stochastic Graphs Have Short Memory: Fully Dynamic Connectivity in Poly-Log Expected Time," *Proceedings of the 22nd International Colloquium on Automata, Languages, and Programming (ICALP'95),* pp. 159-170, 1995.

Appendix

A The Karp and Tarjan's Algorithm

A.1 Performance Evaluation

We present a set of diagrams demonstrating that GCOMPONENTS performs better than COMPONENTS in case of graphs with large edge sets. On the other hand, in case of sparse graphs, the running time of GCOMPONENTS is slightly inferior to the one of COMPONENTS.

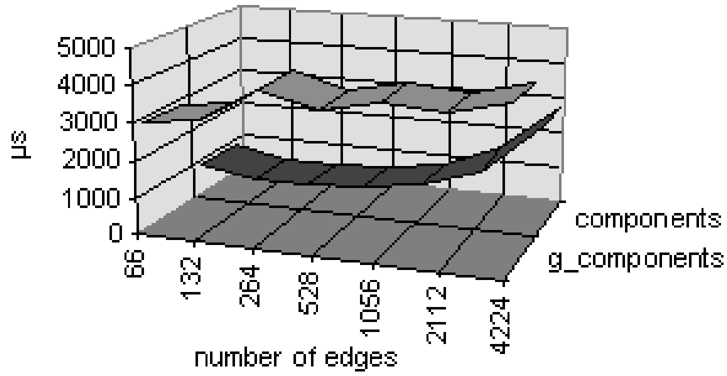

Figure 4: The execution time of GCOMPONENTS and COMPONENTS for a graph with 200 nodes. Notice that in this case, COMPONENTS performs better than GCOMPONENTS. However, the difference is not more than a few milliseconds. Moreover, as the number of edges increases, the running time of COMPONENTS becomes larger. More specifically, the running time of COMPONENTS becomes worse than the one of GCOMPONENTS for more than 5000 edges.

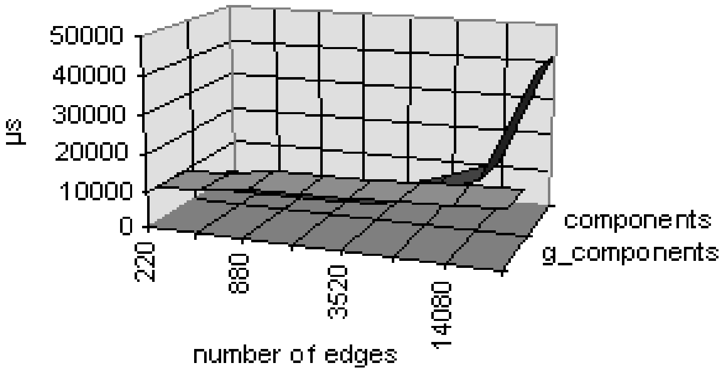

Figure 5: The execution times of GCOMPONENTS and COMPONENTS for a graph with 660 nodes. Obviously, GCOMPONENTS performs better than COMPONENTS if the graph has a large set of edges. Notice that GCOMPONENTS has almost constant running time, which does not depend on the number of edges in the graph. Apparently, this observation is the experimental analog to the theoretical result that the expected execution time of Karp and Tarjan's algorithm is $O(n)$, thus independent of m.

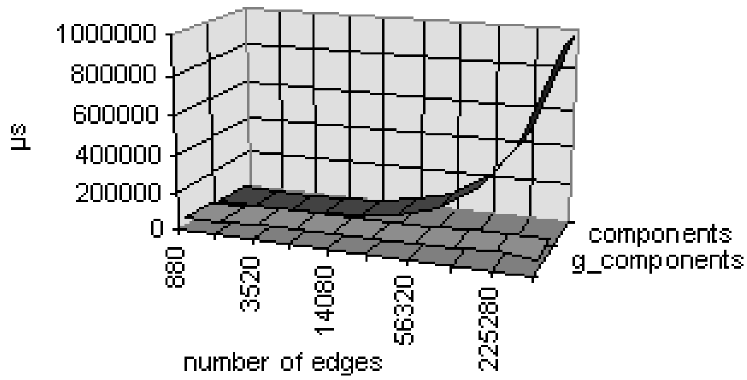

Figure 6: The execution times of GCOMPONENTS and COMPONENTS for a graph with 2640 nodes. Notice that the execution time of COMPONENTS is about $1000ms$, while the one of GCOMPONENTS remains almost constantly under $200ms$.

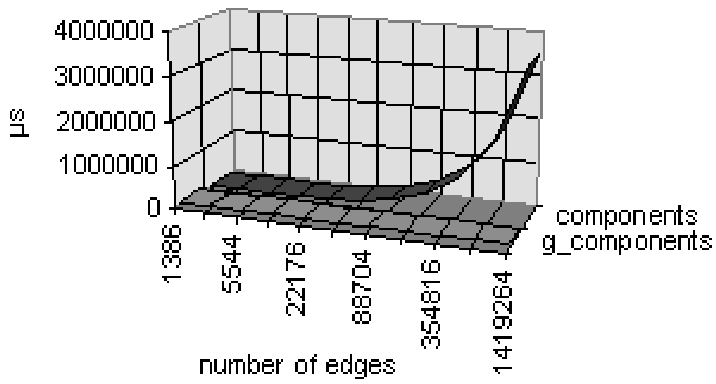

Figure 7: The execution times of GCOMPONENTS and COMPONENTS for a graph with 4160 nodes. It is obvious that there exists a tremendeous difference between the two algorithms for such large graphs. The execution time of COMPONENTS is up to 4 seconds, while the one of GCOMPONENTS is less than $200ms$. For larger graphs, the execution time of COMPONENTS becomes really huge (more than a few minutes), while GCOMPONENTS behaves in an extremely faster way.

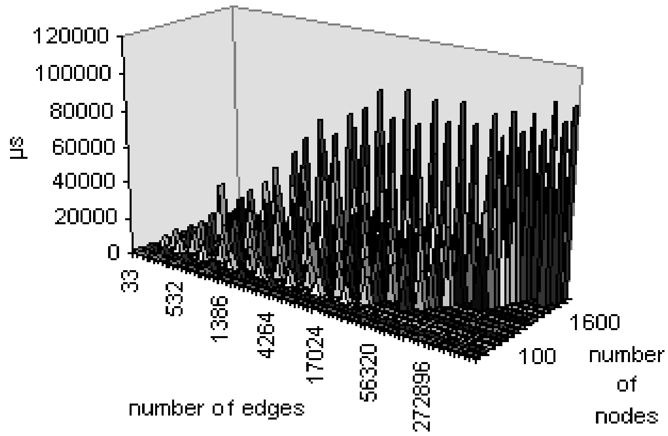

Figure 8: Execution time of GCOMPONENTS for different pairs ⟨`number of nodes`, `number of edges`⟩. For each such pair a bar in the diagram exists. Notice that the execution time of GCOMPONENTS increases with n.

B The NRSY Algorithm

B.1 Description-Implementation

Figure 9 presents C like pseudocode for Algorithm NRSY. Figure 10 describes the actions taken in the activation epoch for an insertion operation. Figure 11 describes the actions taken in the activation epoch for a deletion operation. Procedure NeighborhoodSearch is presented in Figure 12. Pseudocode for procedure NeighborhoodPhase appears in Figure 13.

In Figure 14 edge e is to be deleted. Notice that an edge e' reconnects the two subtrees $T(u), T(v)$ only if $w \in T(u)$ and the path from w' to the root of the tree traverses edge e (which is the case in Figure 14), or if $w \in T(v)$ and the path from w' to the root does not traverse edge e.

```
Algorithm NRSY:
{
    operation_cnt = 0;
    if (a successfull total reconstruction occurs)
        epoch = ACTIVATION;
    else
        epoch = RETIREMENT;
    while (1) {
        switch(operation) {

            case insertion:
                if (epoch == ACTIVATION)
                    call ActivationInsert;
                else
                    Perform it by running any dynamic connectivity algorithm
                    with (expected) update time at most O(n);
                break;

            case deletion:
                if (epoch == ACTIVATION)
                    call ActivationDelete;
                else
                    Perform it by running any dynamic connectivity algorithm
                    with (expected) update time at most O(n);
                break;

            case query:
                if (epoch == ACTIVATION)
                    call ActivationQuery;
                else
                    Perform it by running any dynamic connectivity algorithm
                    with (expected) query time at most O(n);
                break;
        }
        operation_cnt++;
        if (epoch == ACTIVATION && operation_cnt > c_1 n^2 log n) {
            operation_cnt = 0;
            epoch = RETIREMENT;
        }
        else if (epoch == RETIREMENT && operation_cnt > c_2 n log^2 n) {
            operation_cnt = 0;
            if (a successful total reconstruction occurs)
                epoch = ACTIVATION;
        }
    }
}
```

Figure 9: Epochs of Algorithm NRSY.

```
Insertion of an edge:
{
    if (the edge joins vertices of the same tree) {
        the edge is marked as a retired edge;
        the appropriate data structures are updated;
    }
    else
        update the root and component name of the smaller tree;
}
```

Figure 10: Insertion of an edge during the Activation Epoch of Algorithm NRSY.

```
Deletion of an edge:
{
    if ((the edge is a pool edge) or (the edge is a retired edge))
        the edge is deleted by all data structures;
    else if (the edge is a tree edge of a small tree) {
        if (the deletion of the edge disconnects the corresponding graph component) {
            the tree is split into two small trees;
            the smaller of the two trees is relabeled;
        }
        else
            the tree is reconnected;
    }
    else if (the edge is an edge of the tree of the giant component)
        call NeighborhoodSearch;
}
```

Figure 11: Deletion of an edge during the Activation Epoch of Algorithm NRSY.

```
NeighborhoodSearch:
{
    to_do = FAILURE;
    for (turn = 0,i = 0; i < c3 log n; turn = 1 -turn, i++) {
        if (turn)
            Let w be next vertex visited by BFS started from u;
        else
            Let w be next vertex visited by BFS started from v;
        if (w == nil) {
            if (no phase has return success till now)
                to_do = SMALL_TREE;
        }
        else  {
            cal NeighborhoodPhase(w);
            if (NeighborhoodPhase reports success) {
                the edge that reconnects the two pieces of the
                    tree and appropriate distances are maintained;
                to_do = SUCCESS;
            }
        }
    }
    switch (to_do) {
        case SMALL_TREE:
            if (reconnection fails)
                split the giant tree to a still giant tree
                    and a logarithmic size small tree;
            return SUCCESS;
            break;
        case SUCCESS:
            choose the reconnection that is closer to the root
                and reconnect the two pieces;
            return SUCCESS;
            break;
        default:
                return FAILURE;
    }
}
```

Figure 12: Pseudocode for Procedure NeighborhoodSearch.

```
NeighborhoodPhase(node w):
{
    call ActivateRetirementEdges;

    if (there is no pool edge out of w)
        return FAILURE;

    a pool edge e' = ⟨w, w'⟩ is chosen randomly;
    if ((root R belongs to the subtree containing w && !(path from w' to R uses e)) ||
        (!(R belongs to the subtree containing w) && path from w' to R uses e)) {
        if (the distance of the new path is appropriate) {
            the edge and the distance is maintained;
            return SUCCESS;
        }
    }
    call Retire(e);
    return FAILURE;
}
```

Figure 13: Pseudocode for Procedure NeighborhoodPhase.

B.2 Performance Evaluation

We present a set of diagrams demonstrating the performance of the following data structures: `nrsy_connectivity`, `dc_henzinger_king` and `dc_simple` for graphs with $n = 50, 100, 300, 500$, 750 and 1000 nodes. For each such n, we have measured the performance of the algorithms for $m = n/2, n, n \ln n, n^{3/2}$ and $n^2/4$.

Section B.3 presents our experiments for long sequences of 100000 updates. Our experiments for medium sequences of 10000 updates are presented in Section B.4, while Section B.5 presents diagrams for the case of short sequences of 1000 updates.

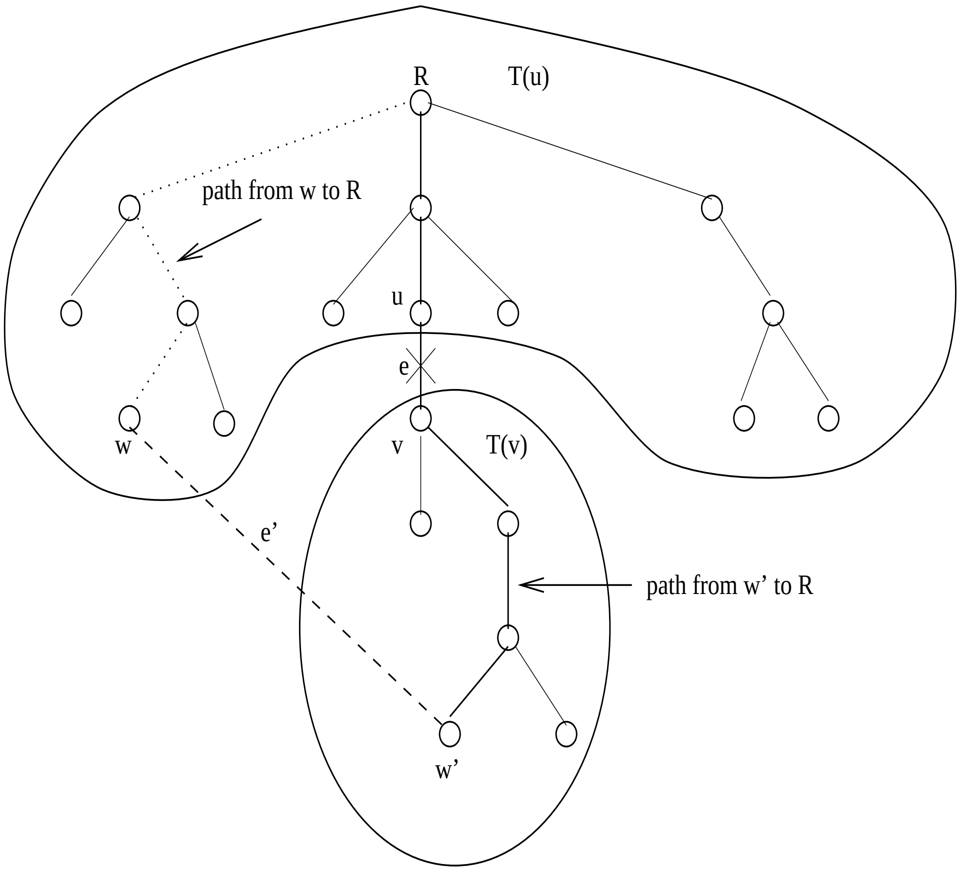

Figure 14: Notice that $R \in T(u)$, while edge e' reconnects the two pieces of the tree if the path from w' to R uses the edge e.

B.3 Long Sequences of Random Updates

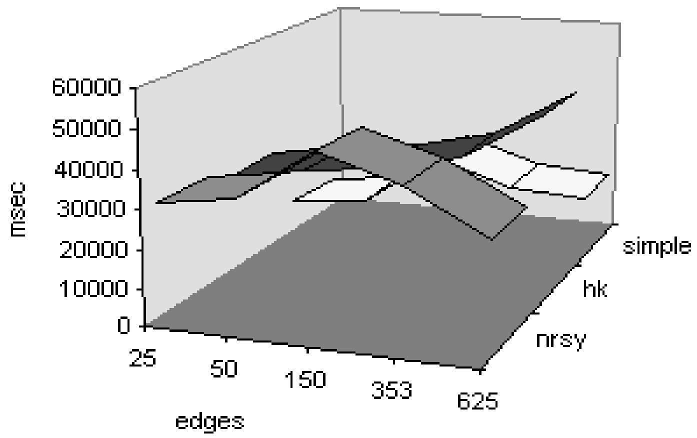

Figure 15: Performance of NRSY, Henzinger-King and Simple for a random graph of 50 nodes and a sequence of 100000 random operations.

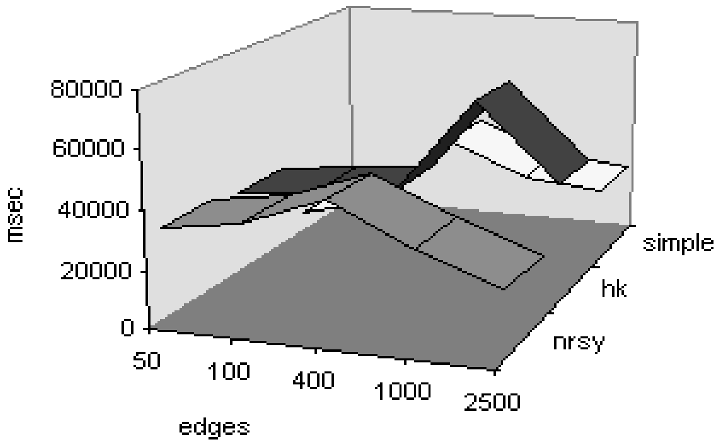

Figure 16: Performance of NRSY, Henzinger-King and Simple for a random graph of 100 nodes and a sequence of 100000 random operations.

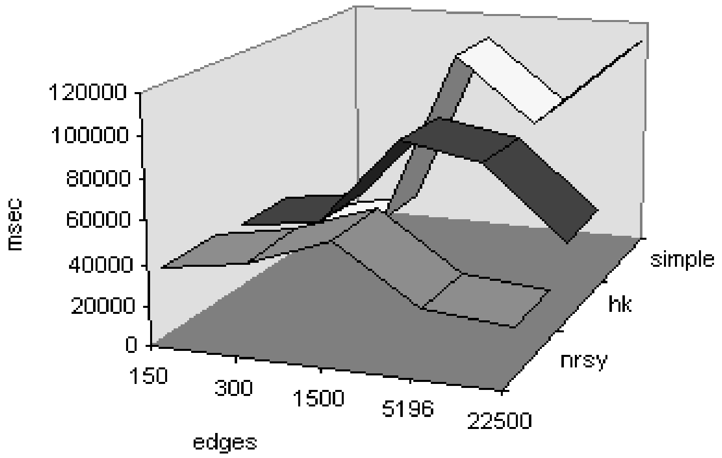

Figure 17: Performance of NRSY, Henzinger-King and Simple for a random graph of 300 nodes and a sequence of 100000 random operations.

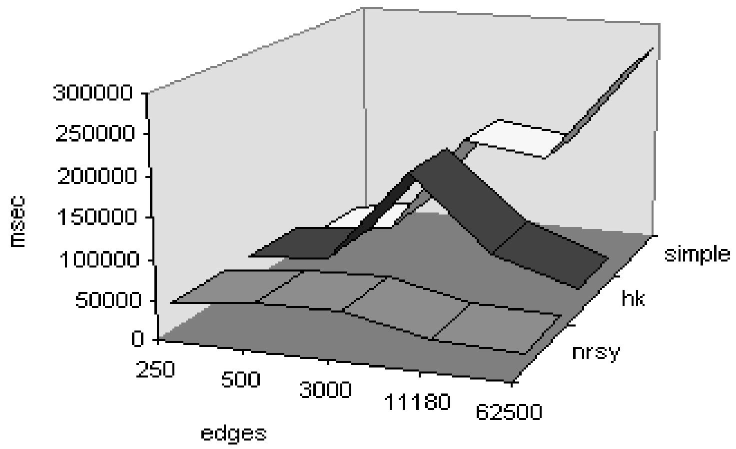

Figure 18: Performance of NRSY, Henzinger-King and Simple for a random graph of 500 nodes and a sequence of 100000 random operations.

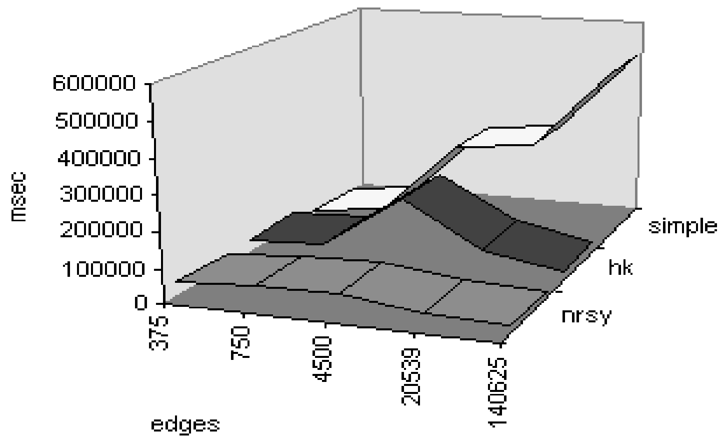

Figure 19: Performance of NRSY, Henzinger-King and Simple for a random graph of 750 nodes and a sequence of 100000 random operations.

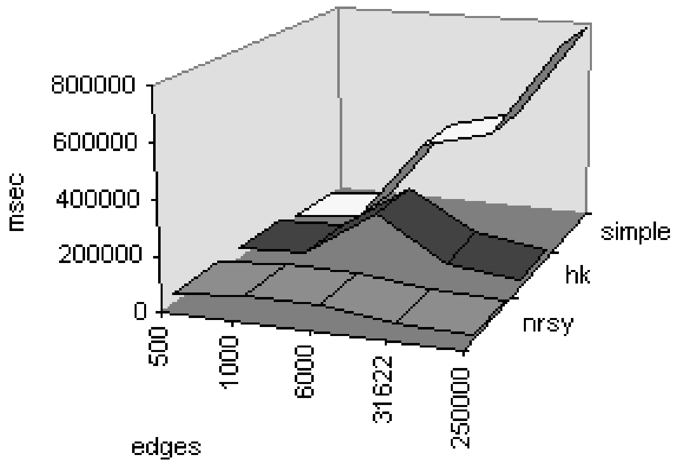

Figure 20: Performance of NRSY, Henzinger-King and Simple for a random graph of 1000 nodes and a sequence of 100000 random operations.

B.4 Medium Sequences of Random Updates

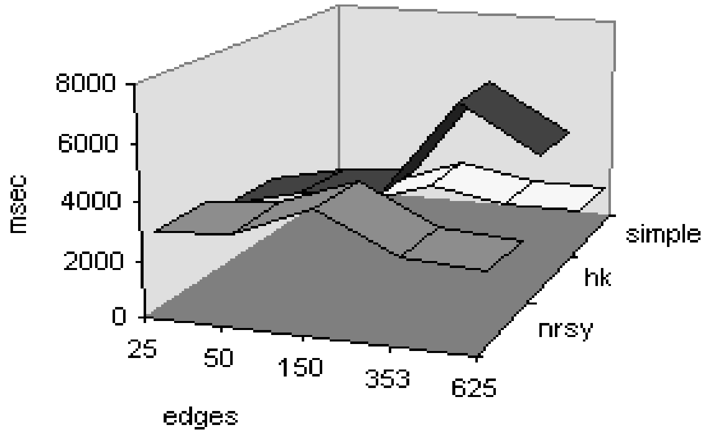

Figure 21: Performance of NRSY, Henzinger-King and Simple for a random graph of 50 nodes and a sequence of 10000 random operations.

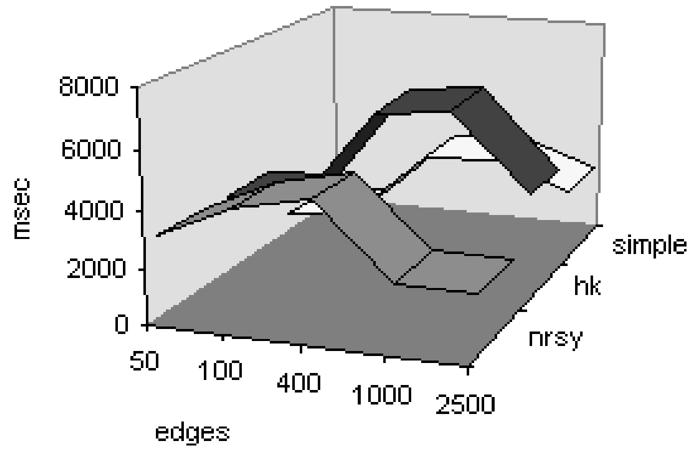

Figure 22: Performance of NRSY, Henzinger-King and Simple for a random graph of 100 nodes and a sequence of 10000 random operations.

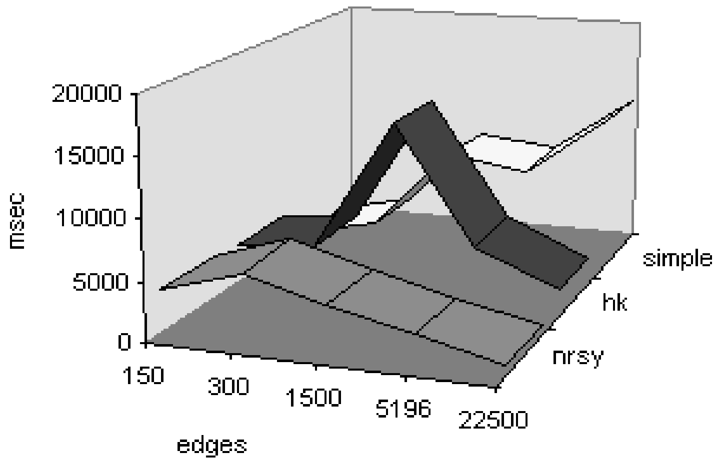

Figure 23: Performance of NRSY, Henzinger-King and Simple for a random graph of 300 nodes and a sequence of 10000 random operations.

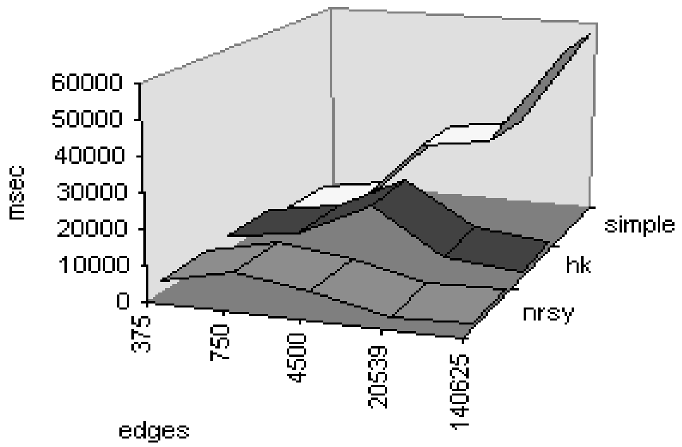

Figure 24: Performance of NRSY, Henzinger-King and Simple for a random graph of 750 nodes and a sequence of 10000 random operations.

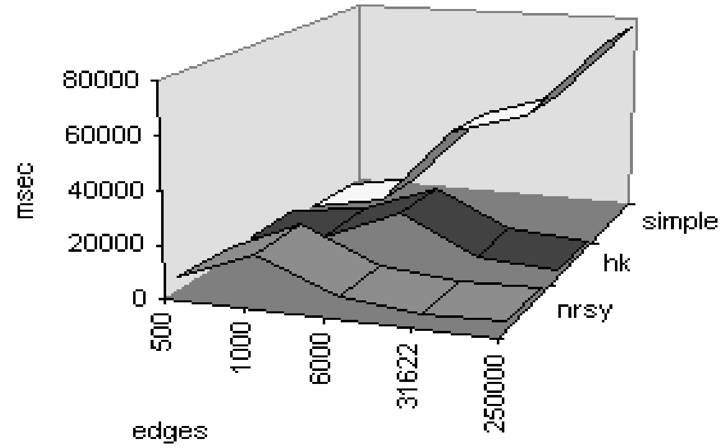

Figure 25: Performance of NRSY, Henzinger-King and Simple for a random graph of 1000 nodes and a sequence of 10000 random operations.

B.5 Short Sequences of Random Updates

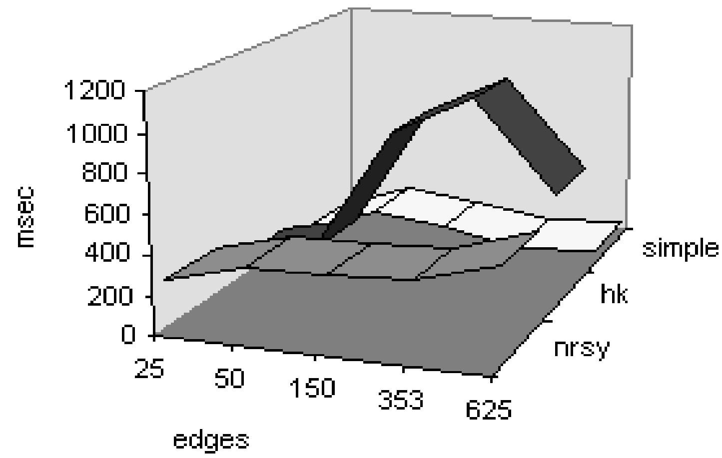

Figure 26: Performance of NRSY, Henzinger-King and Simple for a random graph of 50 nodes and a sequence of 1000 random operations.

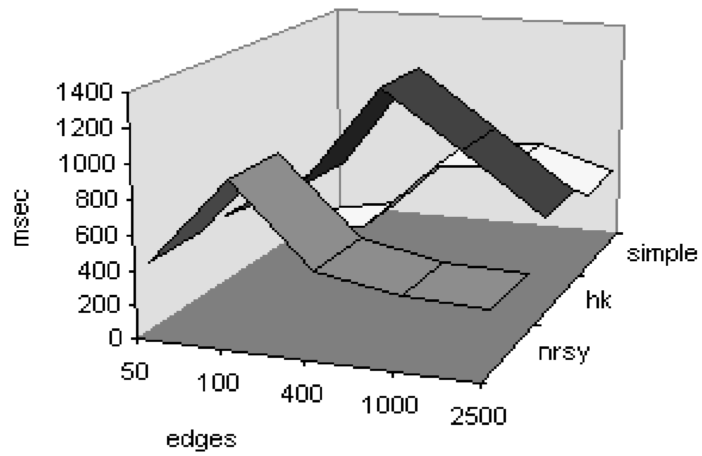

Figure 27: Performance of NRSY, Henzinger-King and Simple for a random graph of 100 nodes and a sequence of 1000 random operations.

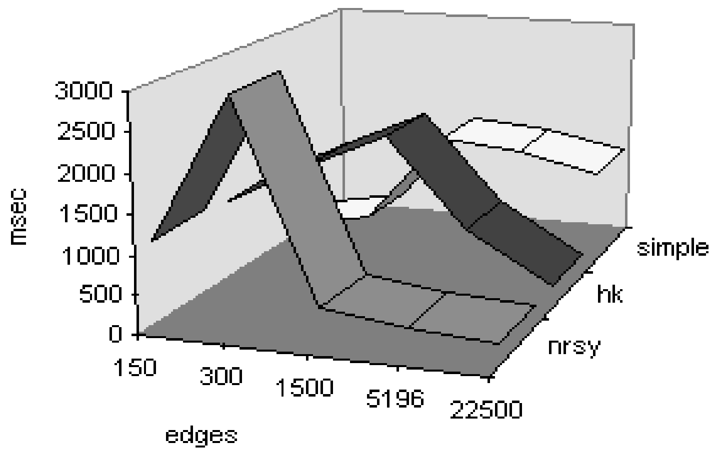

Figure 28: Performance of NRSY, Henzinger-King and Simple for a random graph of 300 nodes and a sequence of 1000 random operations.

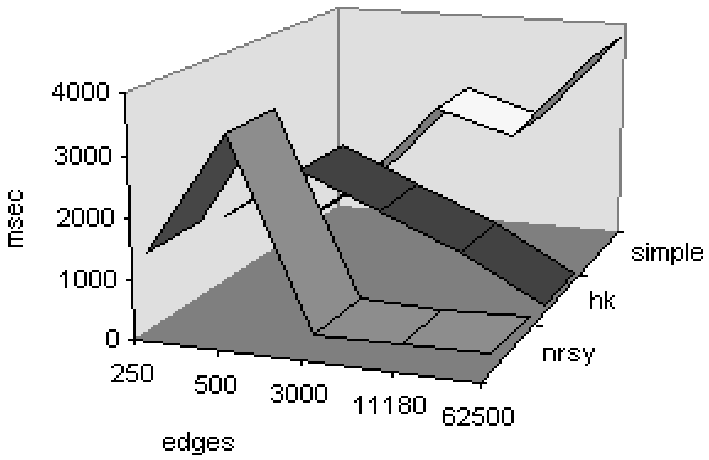

Figure 29: Performance of NRSY, Henzinger-King and Simple for a random graph of 500 nodes and a sequence of 1000 random operations.

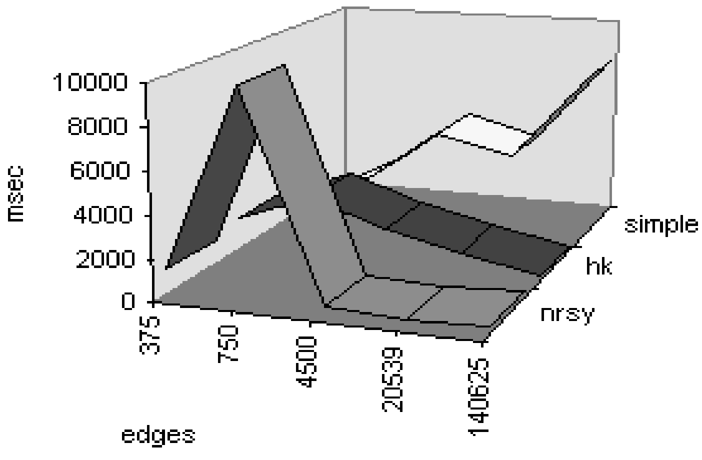

Figure 30: Performance of NRSY, Henzinger-King and Simple for a random graph of 750 nodes and a sequence of 1000 random operations.

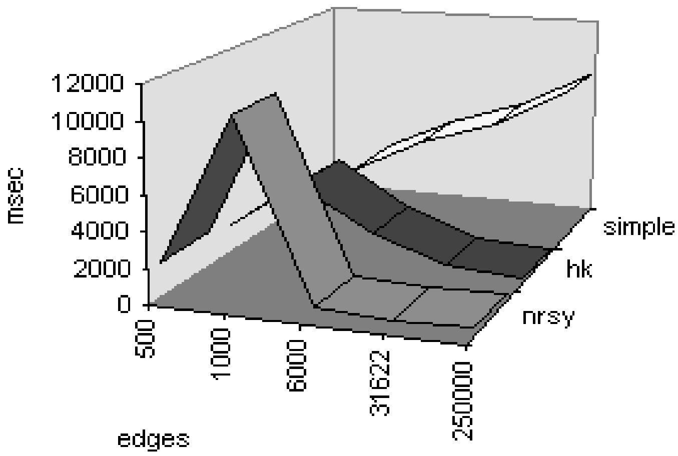

Figure 31: Performance of NRSY, Henzinger-King and Simple for a random graph of 1000 nodes and a sequence of 1000 random operations.