Experimental Analysis of Dynamic All Pairs Shortest Path Algorithms *

Camil Demetrescu † *Giuseppe F. Italiano* ‡

Abstract

We present the results of an extensive computational study on dynamic algorithms for all pairs shortest path problems. We describe our implementations of the recent dynamic algorithms of King and of Demetrescu and Italiano, and compare them to the dynamic algorithm of Ramalingam and Reps and to static algorithms on random, real-world and hard instances. Our experimental data suggest that some of the dynamic algorithms and their algorithmic techniques can be really of practical value in many situations.

1 Introduction

In this paper we consider fully dynamic algorithms for all pairs shortest path problems. Namely, we would like to maintain information about shortest paths in a weighted directed graph subject to edge insertions, edge deletions and edge weight updates. This seems an important problem on its own, and it finds applications in many areas (see, e.g., [25]), including transportation networks, where weights are associated with traffic/distance; database systems, where one is often interested in maintaining distance relationships between objects; data flow analysis and compilers; document formatting; and network routing [11, 23, 24].

This problem was first studied in 1967 [21, 22], but the first fully dynamic algorithms that in the worst case were provably faster than recomputing the solution from scratch were proposed only twenty years later. We recall here that the all pairs shortest path problem can be solved in $O(mn + n^2 \log n)$ worst-case time with Dijkstra's algorithm and Fibonacci heaps [9, 12], where m is the number of edges and n is the number of nodes. Since m

*This work has been partially supported by the IST Programme of the EU under contract n. IST-1999-14.186 (ALCOM-FT), by the Italian Ministry of University and Research (Project "ALINWEB: Algorithmics for Internet and the Web"). A preliminary version of this paper was presented at the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'04), New Orleans, LA. January 2004.

†Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Roma, Italy. Email: demetres@dis.uniroma1.it. URL: http://www.dis.uniroma1.it/~demetres.

‡Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", Roma, Italy. Email: italiano@disp.uniroma2.it. URL: http://www.info.uniroma2.it/~italiano.

can be as high as $\Theta(n^2)$, this is $\Theta(n^3)$ in the worst case. In 1999 King [19] presented a fully dynamic algorithm for maintaining all pairs shortest paths in directed graphs with positive integer weights less than C: the running time of her algorithm is $O(n^{2.5}\sqrt{C \log n})$ per update and $O(1)$ per query. Demetrescu and Italiano [7] showed how to maintain all pairs shortest paths on directed graphs with real-value edge weights that can take at most S different values: their time bound is $O(n^{2.5}\sqrt{S \log^3 n})$ per update and $O(1)$ per query. In 2003, the same authors [8] presented a fully dynamic shortest paths algorithm that requires $O(n^2 \cdot \log^3 n)$ time per update and constant time per query. This bound has been recently improved to $O(n^2(\log n + \log^2(m/n)))$ amortized time per update by Thorup [28].

Our Results.

The objective of this paper is to advance our knowledge on dynamic shortest path algorithms by following up the recent theoretical progress of King [19] and of Demetrescu and Italiano [8] with a thorough empirical study. In particular, we present and experiment with efficient implementations of the dynamic shortest path algorithms described in those papers. We do not consider here the algorithm by Thorup [28], since preliminary investigations suggest that this improvement is mainly of theoretical interest. Our empirical analysis shows that some of the dynamic algorithms and their techniques can be really of practical value in many situations. Indeed, we observed that in practice their speed up is much higher than the one predicted by theoretical bounds: they can be even two to four orders of magnitude faster than repeatedly computing a solution from scratch with a static algorithm. Furthermore, our work may shed light on the relative behavior of some implementations on different test sets and on different computing platforms: this might be helpful in identifying the most suitable dynamic shortest path code for different application scenarios. As a side result of our experimental work, we propose also a new static algorithm, which in practice reduces substantially the total number of edges scanned and thus can run faster than Dijkstra's algorithm on dense graphs.

Related Work.

Besides the extensive computational studies on static shortest path algorithms (see, e.g., [5, 15, 29]), many researchers have been complementing the wealth of theoretical results on dynamic graphs with thorough empirical studies. In particular, Frigioni *et al.* [14] proposed efficient implementations of dynamic transitive closure and shortest path algorithms, while Frigioni *et al.* [13] and later Demetrescu *et al.* [6] conducted an empirical study of dynamic single-source shortest path algorithms. Many of these shortest path implementations refer either to partially dynamic algorithms or to special classes of graphs. More recently, Buriol *et al.* [3] have conducted an extensive computational analysis studying the effects of heap size reduction techniques in dynamic single-source shortest path algorithms. Other dynamic graph implementations include the work of Alberts *et al.* [1] and Iyer *et al.* [17] (dynamic connectivity), and the work of Amato *et al.* [2] and Cattaneo *et al.* [4] (dynamic minimum spanning tree).

2 Experimental Setup

2.1 Test Sets

In our experiments we considered three kinds of test sets: random inputs, synthetic inputs, and real-world inputs.

Random inputs.

We considered random graphs with n nodes, m edges, under the $G_{n,m}$ model. Edge weights are integers chosen uniformly at random in a certain range. To generate the update sequence, we select at random one operation among edge insertion, edge deletion, and edge weight update. If the operation is an edge insertion, we select at random a pair of nodes x, y such that edge (x, y) is not in the graph, and we insert edge (x, y) with random weight. If the operation is a deletion, we pick at random an edge in the graph, and delete it. If the operation is an edge weight update, we randomly pick an edge in the graph, and change its weight to a new random value.

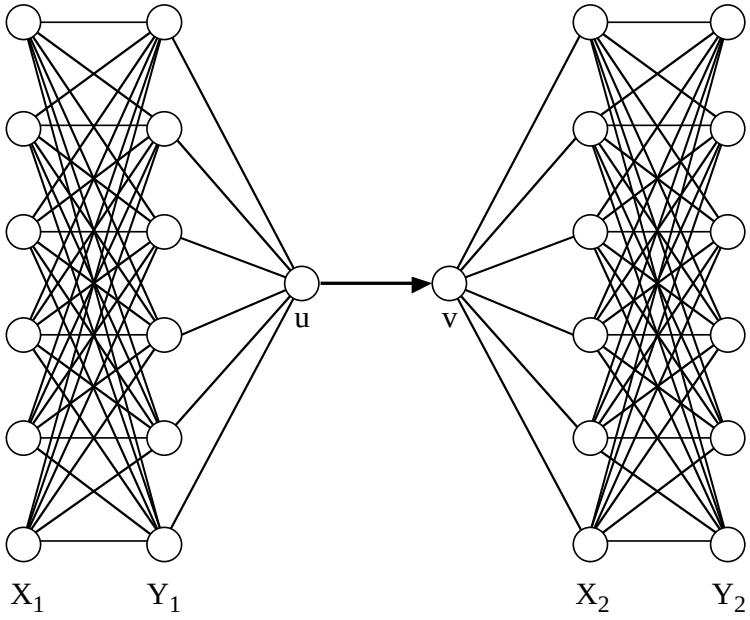


Figure 1: Example of bottleneck graph with complete bipartite components. The sequence of updates is performed on edge (u, v) and affects all node pairs (a, b) with $a \in X_1 \cup Y_1$ and $b \in X_2 \cup Y_2$.

Synthetic inputs.

We considered bottleneck graphs formed by two bipartite components (X_1, Y_1) and (X_2, Y_2), with $|X_1| = |Y_1| = |X_2| = |Y_2|$ and with edges directed from X_i to Y_i, $i = 1, 2$ (see Figure 1).

The density of the bottleneck graph depends on the density of the bipartite components. Nodes in Y_1 reach nodes in X_2 through a single edge (u, v), i.e., there is an edge (y, u) for each $y \in Y_1$ and there is an edge (v, x) for each $x \in X_2$. This way, most shortest paths in the graph go through edge (u, v). All edge weights are chosen uniformly at random within a certain range. Bottleneck inputs are obtained by applying a sequence of random weight updates on the edge (u, v). This is done to force hard instances, which cause many changes in the solution.

Real-world inputs.

We considered two kinds of real-world inputs: US road networks and Internet networks. The US road networks were obtained from `ftp://edcftp.cr.usgs.gov`, and consist of graphs having 148 to 952 nodes and 434 to 2,896 edges. The edge weights can be as high as 200,000. As Internet networks, we considered snapshots of the connections between Autonomous Systems (AS) taken from the University of Oregon Route Views Archive Project (`http://www.routeviews.org`). The resulting graphs (AS_500, . . . ,AS_3000) have 500 to 3,000 nodes and 2,406 to 13,734 edges, with edge weights as high as 20,000. The update sequences we considered for real-world graphs were of two different kinds: random edge weight updates (within the same weight range as the original graph) and random edge weight updates within 5% of their original value. We note that the latter update sequence might be closer to real applications (for which no update traces were available)

2.2 Computing Platforms

To assess the experimental performance of our implementations, we experimented on a variety of computing platforms, ranging from a low-end cache system (256KB L2 cache) to a high-end cache system (1.4MB L2 cache, 32MB L3 cache):

- AMD Athlon - 1.6 GHz, 256KB L2 cache, 1GB RAM.
- Intel Xeon - 500 MHz, 512KB L2 cache, 512MB RAM.
- Intel Pentium 4 - 2.0 GHz, 512KB L2 cache, 2GB RAM.
- Sun UltraSPARC IIi - 440 MHz, 2MB L2 cache, 256MB RAM.
- IBM Power 4 - 1.1 GHz, 1.4MB L2 cache, 32MB L3 cache, 64GB RAM.

On these platforms, we experimented on a variety of different operating systems (Linux kernel 2.2.18, Solaris 8, Windows XP Professional, AIX 5.2) and compilers (GNU gcc 2.95, IBM xlc 6.0, Microsoft Visual C++ 7, Metrowerks CodeWarrior 6). We also used different systems for monitoring memory accesses and for simulating cache effects (Valgrind, Cachegrind [27]).

In our experiments, we noticed that most of the relative behaviors of the implementations did not depend heavily on the architecture/operating system/compiler. Whenever

this dependence seems to be significant, we will point it out explicitly. In particular, in Section 3, we will start presenting our experimental results homogeneously on the low-end cache platform (AMD Athlon, 1.6 GHz, 256KB L2 cache, 1GB RAM). We will then show in Section 4 how more efficient memory systems can make a difference for some implementations.

3 Algorithm Implementation and Tuning

In this section we briefly describe the algorithms considered in our experimental analysis, addressing implementation and performance tuning issues. Table 1 lists the algorithms under investigation and their theoretical asymptotic bounds.

Algorithm	**Reference**	**Update Time**	**Query Time**	**Space**
`S-DIJ`	[9, 12]	$O(mn + n^2 \log n)$	$O(1)$	$O(n^2)$
`S-LSP`	This paper	$O(\|LSP\| + n^2 \log n)$	$O(1)$	$O(n^2)$
`D-RRL`	[26]	$O(mn + n^2 \log n)$	$O(1)$	$O(n^2)$
`D-KIN`	[19]	$\widetilde{O}(n^{2.5}\sqrt{C})$	$O(1)$	$\widetilde{O}(n^{2.5}\sqrt{C})$
`D-LHP`	[8]	$\widetilde{O}(n^2)$	$O(1)$	$\widetilde{O}(mn)$

Table 1: Algorithms under investigation. We use $\widetilde{O}(f(n))$ to denote $O(f(n)\text{polylog}(n))$. For `S-LSP`, the quantity $|LSP|$ denotes the total number of edge scans performed by the algorithm, which can be much smaller than mn, as we will see in Section 3.3. For `D-KIN`, C denotes the maximum edge weight in the graph.

The first implementation that we considered is Dijkstra's static single-source algorithm [9, 12] repeated from every node (`S-DIJ`). `S-LSP` is another all-pairs shortest paths static algorithm that will be described in Section 3.4. The other implementations are dynamic shortest paths algorithms: in particular, `D-RRL` is an implementation of the algorithm by Ramalingam and Reps [26], `D-KIN` is an implementation of the algorithm by King [19], and `D-LHP` is an implementation of the algorithm by Demetrescu and Italiano [8]. More details about these implementations will be given later.

All the algorithms were implemented in `C` following a uniform programming style and using exactly the same data structure implementations as basic building blocks (heaps, dynamic arrays, hash tables, graphs). Our goal was more to provide a unified experimental framework for comparing the relative efficiency of different algorithmic techniques, rather than to produced heavily engineered codes. We remark that the absolute performances of our codes might be further improved by reducing the library overhead due to, e.g., runtime checking and exception handling, and by using more engineered implementations of data structures (e.g., better heap implementations). In the code development process, we used the `gprof` profiling tool to identify hot spots in code and the Cachegrind system to analyze cache effects, tuning the code accordingly. We also used the experimental results to identify

a good setting of the relevant parameters of each implementation. The source code of our implementations is distributed under the terms of the GNU Lesser General Public License and is available at the URL `http://www.dis.uniroma1.it/~demetres/experim/dsp/` along with the full experimental package, including test sets, generators, and scripts to reproduce the experiments described in this paper.

3.1 The Algorithm by Ramalingam and Reps (`D-RRL`)

The algorithm by Ramalingam and Reps [26] works on a directed graph G with strictly positive real-valued edge weights and maintains information about the shortest paths from a given node s. In particular, it maintains a directed acyclic subgraph of G containing all the edges that belong to at least one shortest path from s. To support an edge weight update, the algorithm first computes the subset of nodes whose distance from s is affected by the update. Distances are then updated by running a Dijkstra-like procedure on those nodes. Maintaining single-source shortest paths from s requires $O(m_a + n_a \log n_a)$ per update, where n_a is the number of nodes affected by the update and m_a is the number of edges having at least one affected endpoint. This yields a $O(mn + n^2 \log n)$ time bound in the worst case for dynamic all pairs shortest paths.

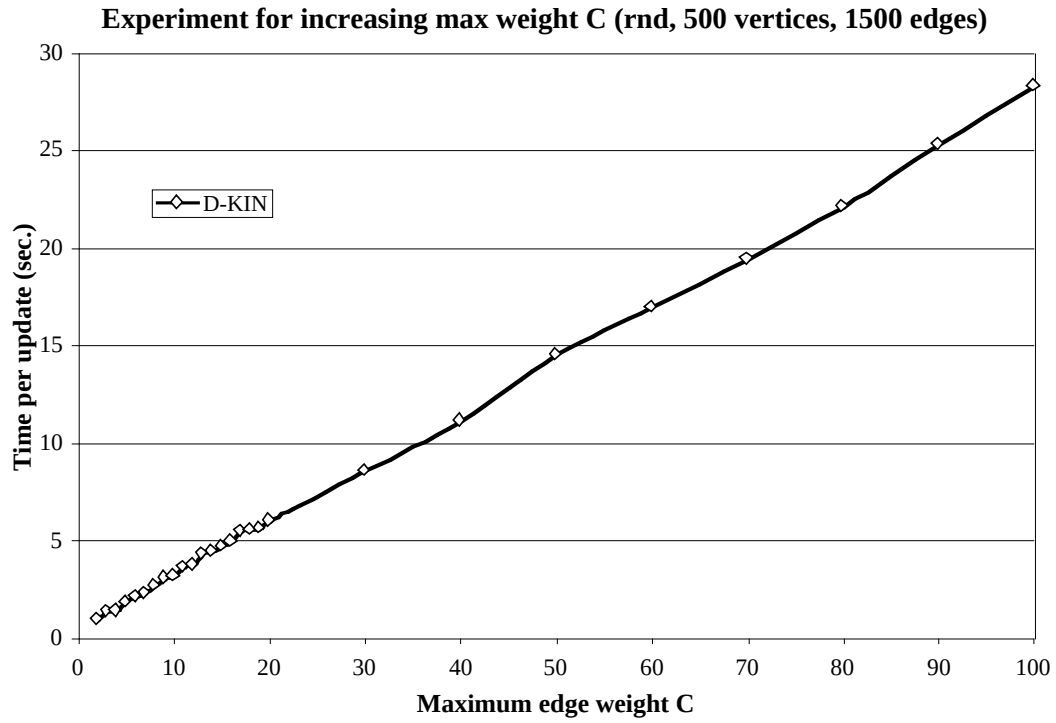
Our implementation.

The algorithm by Ramalingam and Reps is known to be very fast in practice (see, e.g., [6, 11, 13]). In our implementation, we considered a further simplified version of the original algorithm, which was previously described in [6]. This lighter version, which we refer to as `D-RRL`, maintains a shortest paths tree instead of a directed acyclic subgraph, and does not spend time in identifying nodes that do not change distance from the source after an update. If the graph has only a few different paths having the same weight (as in the real-world inputs we considered), this variant can be much faster (see [6]) than the original algorithm in [26].

3.2 The Algorithm by King (`D-KIN`)

The dynamic shortest paths algorithm by King [19] works on directed graphs with small integer weights. The main idea behind the algorithm is to maintain dynamically all pairs shortest paths up to a distance d, and to recompute longer shortest paths from scratch at each update by stitching together shortest paths of length $\leq d$.

To maintain shortest paths up to distance d, the algorithm keeps a pair of in/out shortest paths trees $IN(v)$ and $OUT(v)$ of depth $\leq d$ rooted at each node v. Trees $IN(v)$ and $OUT(v)$ are maintained with a variant of the decremental data structure by Even and Shiloach [10]. It is easy to prove that, if the distance d_{xy} between any pair of nodes x and y is at most d, then d_{xy} is equal to the minimum of $d_{xv} + d_{vy}$ over all nodes v such that $x \in IN(v)$ and $y \in OUT(v)$. To support updates, insertions/decreases of edges around a node v are handled by rebuilding only $IN(v)$ and $OUT(v)$, while edge deletions/increases



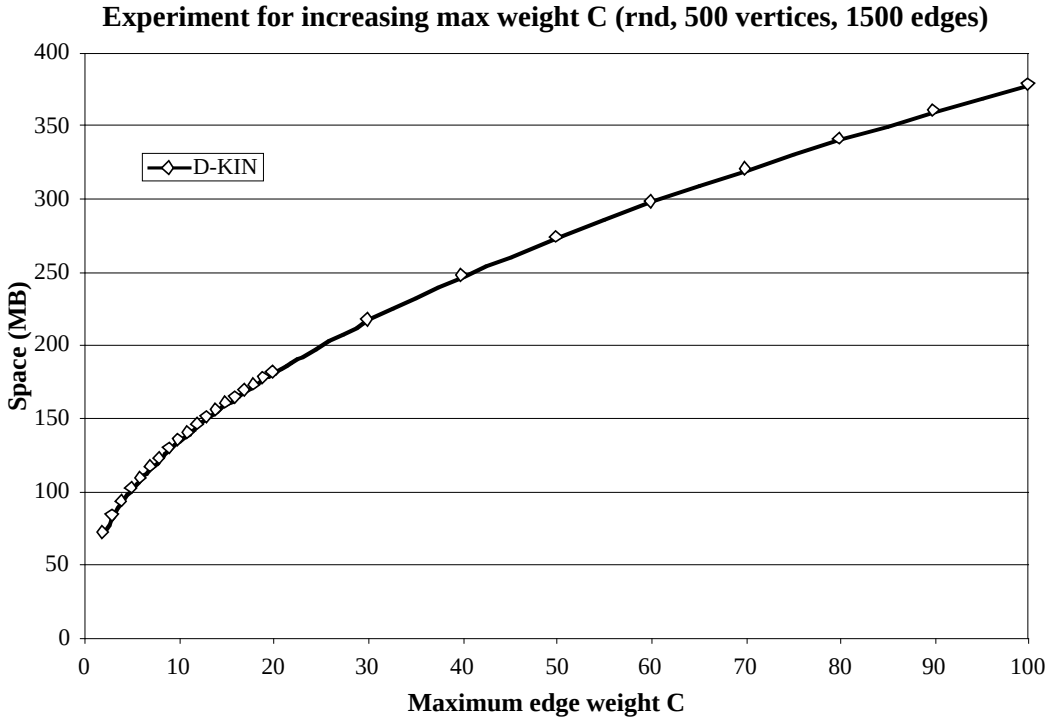


Figure 2: Algorithm `D-KIN`: dependence of the time per update and space on the maximum edge weight C. The experiment was done on an AMD Athlon, 1.6 GHz, 256KB L2 cache, 1GB RAM. The update sequence contained 100 evenly mixed operations on a random graph with 500 nodes and 1500 edges.

are performed via operations on any trees that contain them. The amortized cost of such updates is $O(n^2 d)$ per operation.

To maintain shortest paths longer than d, the algorithm exploits the following property (see, e.g., [16]): if H is a random subset of $\Theta((Cn \log n)/d)$ nodes in the graph, then the probability of finding more than d consecutive nodes in a path, none of which are from H, is very small. Thus, if we look at nodes in H as "hubs", then any shortest path from x to y of length $\geq d$ can be obtained by stitching together shortest subpaths of length $\leq d$ that first go from x to a node in H, then jump between nodes in H, and eventually reach y from a node in H. This can be done by first computing shortest paths only between nodes in H using any static all-pairs shortest paths algorithm, and then by extending them at both endpoints with shortest paths of length $\leq d$ to reach all other nodes. This stitching operations requires $O(n^2|H|) = O((Cn^3 \log n)/d)$ time.

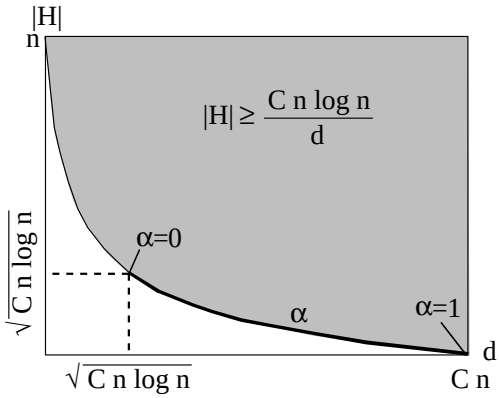
Choosing $d = \sqrt{Cn \log n}$ yields an $O(n^{2.5}\sqrt{C \log n}\,)$ amortized update time. Since H can be also computed deterministically, the algorithm can be derandomized. While the original algorithm in [19] requires $O(n^3)$ space, in [20] King and Thorup showed how to reduce space to $\widetilde{O}(n^{2.5}\sqrt{C}\,)$. The interested reader can find the low-level details of the algorithm in [19, 20].

Our implementation.

Following the techniques for space reduction [20], we implemented a simpler randomized version of the algorithm by King, which we refer to as `D-KIN`. The code is divided into three modular building blocks: (i) an increase-only data structure for maintaining single-source shortest paths up to distance d; (ii) a forest of in/out trees for maintaining all pairs shortest paths up to distance d under fully dynamic update sequences; (iii) a data structure that maintains a forest of in/out trees and performs stitching to rebuild shortest paths longer than d.

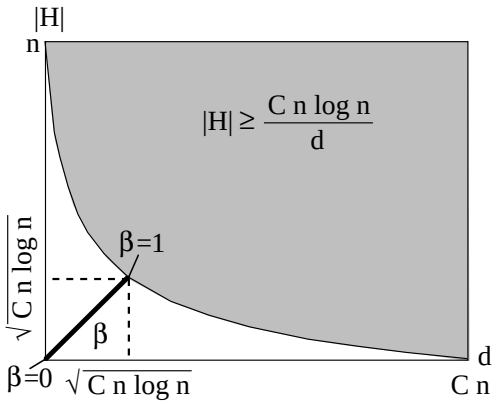
Since the maximum edge weight C is involved in the theoretical bounds, we conducted some experiments aimed at evaluating the effect of increasing C on the update time and space. For instance, the experiment considered in Figure 2 shows that while going from a random graph with 500 nodes, 1500 edges, and maximum weight $C = 10$ to a graph with the same size and $C = 100$, the running time can degrade by about a factor of 10, and the space usage can degrade by about a factor of 3. Note that, while the space is growing as predicted by the theoretical analysis, the time bound appears to grow linearly in the given range of C: this is most probably due to the $O(n^{2.5}\sqrt{C \log n}\,)$ bound not being tight for small values of C. Since C can be as high as $20,000$ in Internet graphs and as high as $200,000$ in US road networks, we could not run `D-KIN` on these real-world inputs.

Profiling information revealed that with the standard setting $|H| = d = \sqrt{C \log n}$, typically more than 75% of the update time is required by the path stitching procedure. We thus expected a reduction of $|H|$ to yield substantial benefits in the running time, and this was fully confirmed by our experiments. As an example, Figure 3 (a) shows the result of an experiment on a random graph with 500 nodes, 1500 edges, and weights in $[0, 5]$, where we set $|H| = \frac{Cn \log n}{d}$ and $d = (1 - \alpha)\sqrt{Cn \log n} + \alpha \cdot Cn$, for $\alpha \in [0, 0.05]$. The



(a)

alpha	time (sec)	space (MB)	avg. error
0.00	1.58	133	0
0.01	0.81	180	0
0.02	0.63	225	0
0.03	0.50	272	0
0.04	0.41	319	0
0.05	0.36	365	0



(b)

beta	time (sec)	space (MB)	avg. error
1.0	1.58	133	0
0.9	1.29	122	0
0.8	1.00	110	0
0.7	0.79	100	0
0.6	0.65	88	0
0.5	0.56	78	3.431
0.4	0.44	66	4.468
0.3	0.36	53	6.904

Figure 3: Algorithm `D-KIN`: parameter tuning with α and β. The experiment was done on an AMD Athlon, 1.6 GHz, 256KB L2 cache, 1GB RAM. The update sequence contained 100 evenly mixed operations.

experiment showed that going from $\alpha = 0$ to $\alpha = 0.05$, which corresponds to increasing d and decreasing $|H|$ by 2.5×, can yield a speedup of 3× at the price of a space increase of 2.7×. For larger values of α the time improvements were negligible, while space kept on increasing substantially. As predicted by the long paths property of [16], with this set of parameters we found no errors in the maintained distances throughout our experiments.

Another interesting question was how far we can decrease both $|H|$ and d without incurring errors. To this aim, we ran an experiment with $|H| = d = \beta \cdot \sqrt{Cn \log n}$, for $\beta \in (0, 1]$ on a random graph with 500 nodes, 1500 edges, and weights in $[0, 5]$. We found out that we can nearly halve both $|H|$ and d (i.e., $\beta = 0.6$) without incurring distance errors, with substantial time and space improvements (see Figure 3 (b)). For smaller values of β, we started getting errors. In general, we could see that the larger the number of nodes or edges in the graph, the smaller were the values of β for which errors started to appear. Since space was crucial to experiment with reasonably large graphs, we preferred to tune β rather than α in the `D-KIN` code used in our experiments.

3.3 The Algorithm by Demetrescu and Italiano (D-LHP)

The dynamic shortest path algorithm in [8] works on directed graphs with nonnegative real-valued edge weights and hinges on the notion of *locally shortest paths* (LSP): we say that a path π is locally shortest if every proper subpath of π is a shortest path (note that π is not necessarily a shortest path). A *historical shortest path* is a path that has been a shortest path at some point during the sequence of updates, and none of its edges has been updated since then. We further say that a path π in a graph is *locally historical* if every proper subpath of π is a historical shortest path. The main idea behind the algorithm is to maintain dynamically the set of locally historical paths (LHP), which include locally shortest paths and shortest paths as special cases. The following theorem from [8] bounds the number of paths that become locally historical after each update:

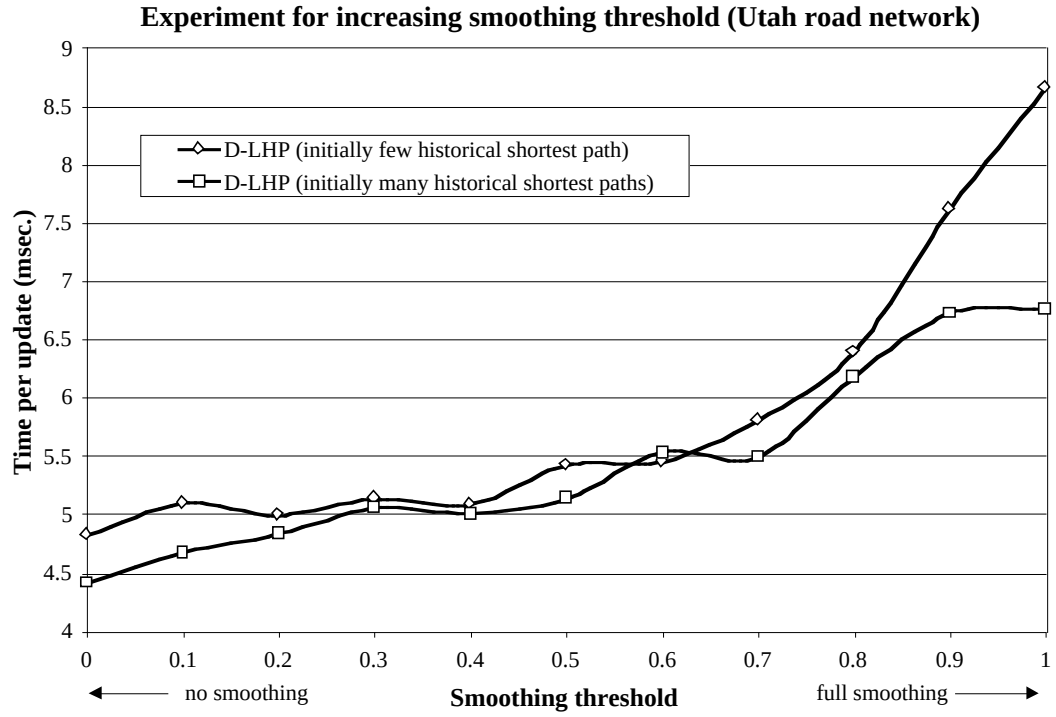
Theorem 1 *Let G be a graph subject to a sequence of update operations. If at any time throughout the sequence of updates there are at most z historical shortest paths between each pair of nodes, then the amortized number of paths that become locally historical at each update is $O(zn^2)$.*

To keep changes in locally historical paths small, it is then desirable to have as few historical shortest paths as possible throughout the sequence of updates. To do so, the algorithm transforms on the fly the input update sequence into a slightly longer equivalent sequence that generates only a few historical shortest paths. In particular, it uses a simple *smoothing* strategy that, given any update sequence Σ of length k, produces an operationally equivalent sequence $F(\Sigma)$ of length $O(k \log k)$ that yields only $O(\log k)$ historical shortest paths between each pair of nodes in the graph (see [8]). By Theorem 1, this technique implies that only $O(n^2 \log k)$ paths become locally historical at each update in the smoothed sequence $F(\Sigma)$.

To support an edge weight update operation, the algorithm works in two phases. It first removes all maintained paths that contain the updated edge. Then it runs a dynamic modification of Dijkstra's algorithm [9] in parallel from all nodes: at each step a shortest path with minimum weight is extracted from a priority queue and it is combined with existing historical shortest paths to form new locally historical paths. The update algorithm spends $O(\log n)$ time for each of the $O(zn^2)$ new locally historical paths. Since the smoothing strategy lets $z = O(\log n)$ and increases the length of the sequence of updates by an additional $O(\log n)$ factor, this yields $O(n^2 \log^3 n)$ amortized time per update. Even with smoothing, there can be as many as $O(mn \log n)$ locally historical paths in a graph: this implies that the space required by the algorithm is $O(mn \log n)$ in the worst case. We refer the interested reader to [8] for the low-level details of the method.

Our implementation.

Our implementation (D-LHP) followed closely the theoretical algorithm in [8], with two main differences. First, in our code, after an update we only process node pairs that are affected by the change. Second, we implemented one additional heuristic: namely, we stopped



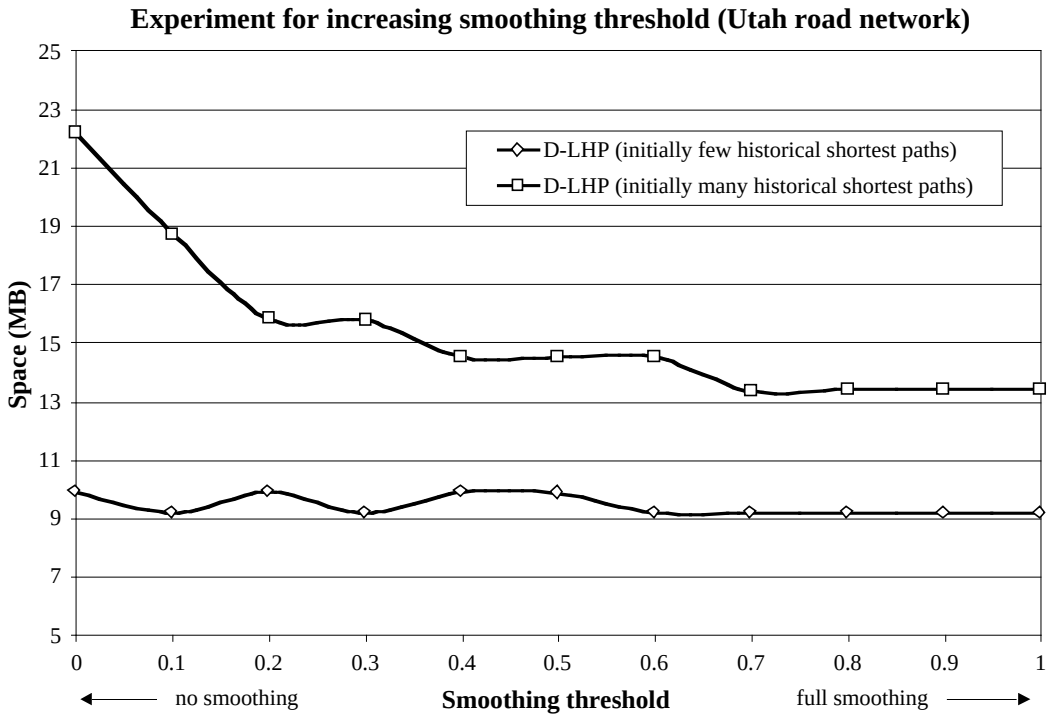


Figure 4: Algorithm `D-LHP`: effects of varying the degree of smoothing on the Utah road network initialized from the initial graph (few historical shortest paths), or from the empty graph via edge insertions (many historical shortest paths). The experiment was done on an AMD Athlon, 1.6 GHz, 256KB L2 cache, 1GB RAM. The update sequence contained $1,000$ evenly mixed operations (after the initialization).

performing smoothing whenever the ratio between the number of created LHPs and deleted LHPs exceeded a certain *smoothing threshold*. In particular, when the smoothing threshold is 0, no smoothing is in place, while a smoothing threshold equal to 1 corresponds to full smoothing. We expected that the main objective of smoothing was to ensure the theoretical worst-case bounds, but we were not fully convinced of its practical significance. To assess the effects of smoothing, we performed different experiments on `D-LHP` for different values of the smoothing threshold and for two different data structure initialization methods:

1. *Initially at most one historical shortest path per node pair*: We started a random update sequence from a data structure having at most one (historical) shortest path per pair. In this scenario, smoothing did not appear to be of any benefit.

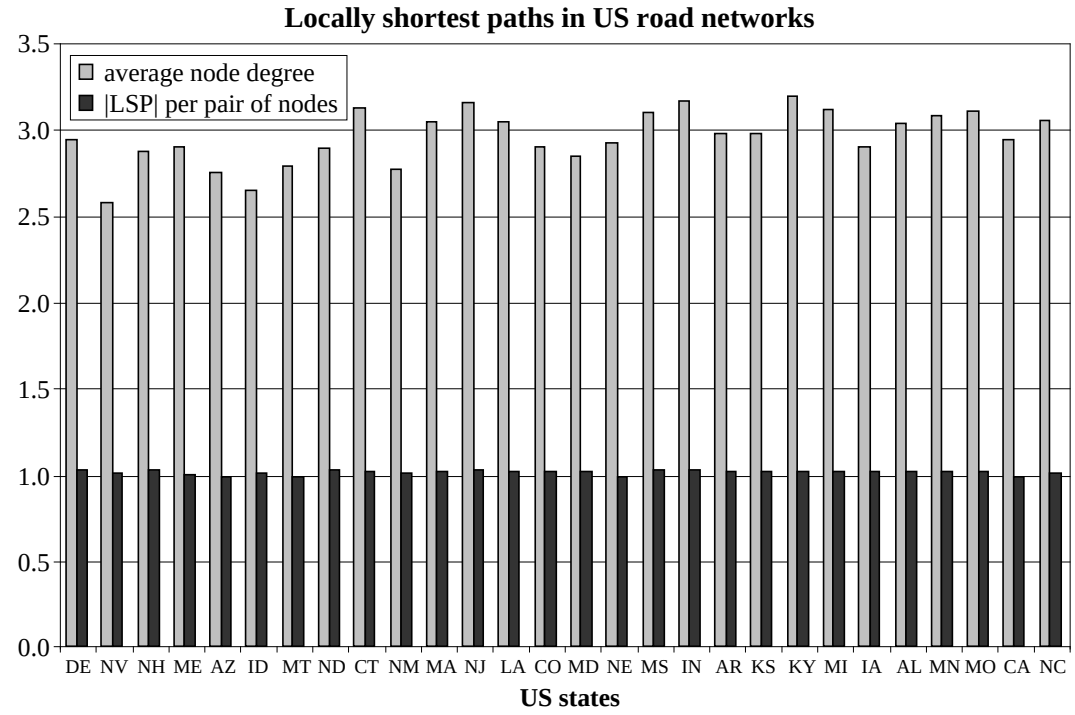
2. *Initially many historical shortest paths*: We built the data structure via edge insertions starting from an empty graph, in order to force artificially the data structure to contain at the beginning of the random update sequence a very large number of historical shortest paths. We remark that only edge insertions or edge weight decreases can create new historical shortest paths without destroying existing historical shortest paths. In this scenario, our experiments showed that the main positive effect of smoothing was to reduce the space usage, but this always came at the price of a running time overhead.

For example, in Figure 4 we report the results of such an experiment on the Utah road network (269 nodes and 742 edges). These results suggest that a certain degree of smoothing can be useful only when there can be a large number of historical shortest paths in the data structure. This may happen, e.g., if we have long bursts of edge weight decreases or edge insertions. Since even in this case the payoff of smoothing seemed of limited extent, we decided to use in all our further experiments a `D-LHP` code with smoothing threshold set to 0 (i.e., no smoothing).

3.4 A New Static Algorithm Based on Locally Shortest Paths (`S-LSP`)

Another interesting issue to investigate experimentally was related to the number of locally shortest paths in a graph. In the worst case, we know that they can be as many as $O(mn)$. But how many of them can we have in "typical" instances? Our experiments showed that in real-world graphs they tend to be very close to n^2, i.e., there is typically only one locally shortest path between any pair of nodes, which is also a shortest path (see upper chart in Figure 5). For random graphs, this number appears to grow very slowly with the graph density (see lower chart in Figure 5).

This suggested that locally shortest paths could be exploited also for static all pairs shortest path algorithms. In particular, we investigated how to deploy locally shortest paths in Dijkstra-like algorithms in order to reduce substantially the number of total edge scans, which is known to be the performance bottleneck for shortest path implementations on



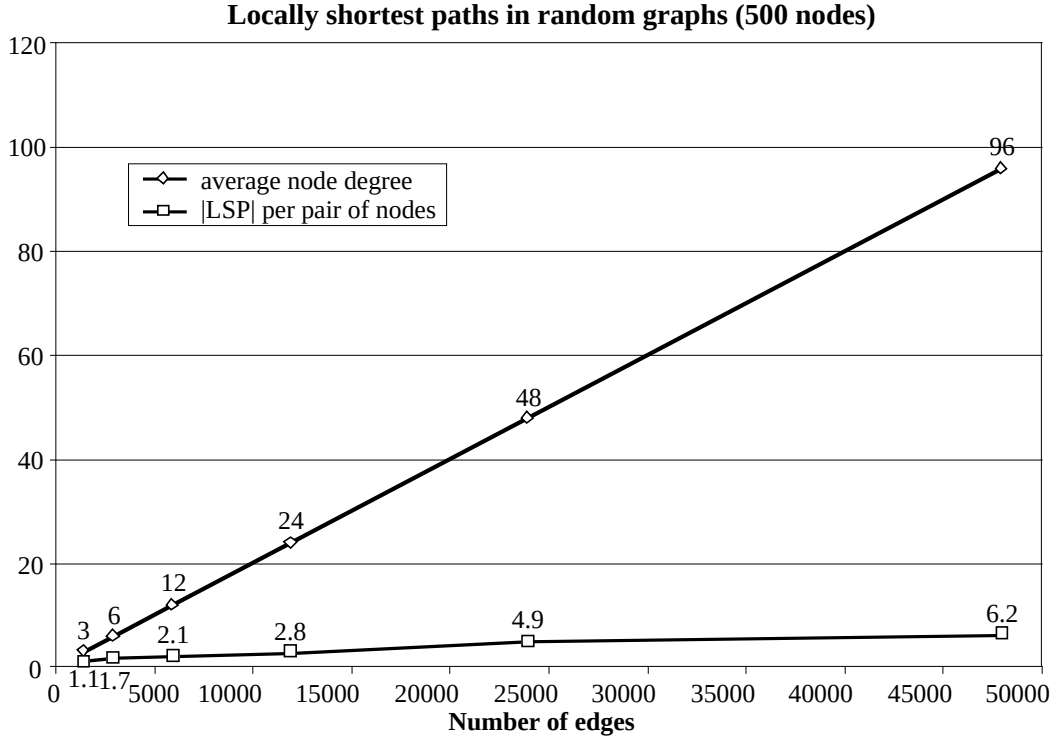


Figure 5: Number of locally shortest paths in random graphs and in US road networks.

dense graphs. We designed a static algorithm, which we refer to as `S-LSP`, that essentially runs Dijkstra's algorithm in parallel from all nodes and scans only edges in locally shortest paths to reduce the overall work. The running time of `S-LSP` is $O(|LSP| + n^2 \log n)$, where $|LSP|$ is the number of locally shortest paths in the graph. This can yield substantial time savings whenever $|LSP| << mn$. For instance, in Figure 6 we report the results of an experiment comparing `S-LSP` and Dijkstra's algorithm (`S-DIJ`) on random graphs with 500 nodes and increasing edge density: as it can be seen, while on sparse graphs the data structure overhead in the algorithms is significant, as the graph becomes denser and edge scanning becomes more relevant in the running times of the algorithms, `S-LSP` can be substantially faster than `S-DIJ`.

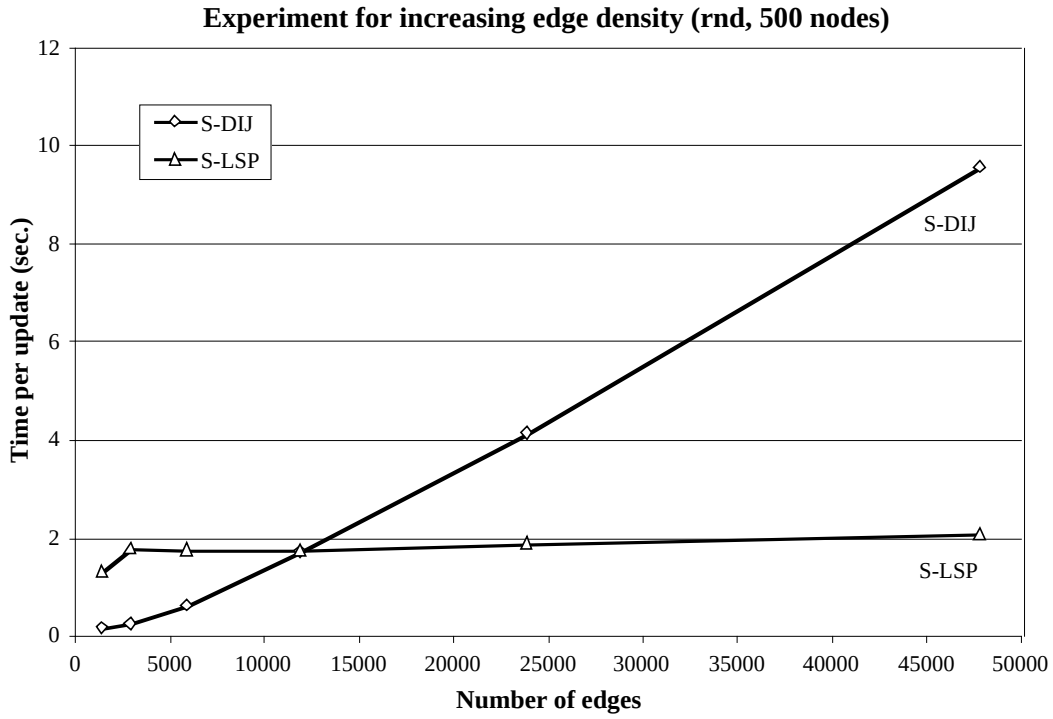


Figure 6: `S-LSP` versus `S-DIJ` on a random graph with 500 nodes, increasing densities, and edge weights in the range $[1, 1000]$. The experiment was done on an AMD Athlon, 1.6 GHz, 256KB L2 cache, 1GB RAM. The update sequence contained $1,000$ evenly mixed operations.

Note that `S-LSP` has a similar flavor as the hidden paths algorithm by Karger *et al.* [18], which runs in $O(m^*n + n^2 \log n)$ time, where m^* is the number of edges that participate in shortest paths. We remark that the two approaches are quite different. To see this, consider the bottleneck graph shown in Figure 1 with edge weights equal to 1: for the hidden paths algorithm we get $m^* = m$, since every edge is trivially a shortest path, while $|LSP| = O(n^2)$. Thus, the hidden paths algorithm runs in $O(mn + n^2 \log n)$ time, while `S-LSP` runs in $O(n^2 \log n)$ time. By the optimal-substructure property of locally shortest paths, it is easy to see that $|LSP| \leq m^*n \leq mn$: thus, `S-LSP` is never slower than the hidden paths algorithm.

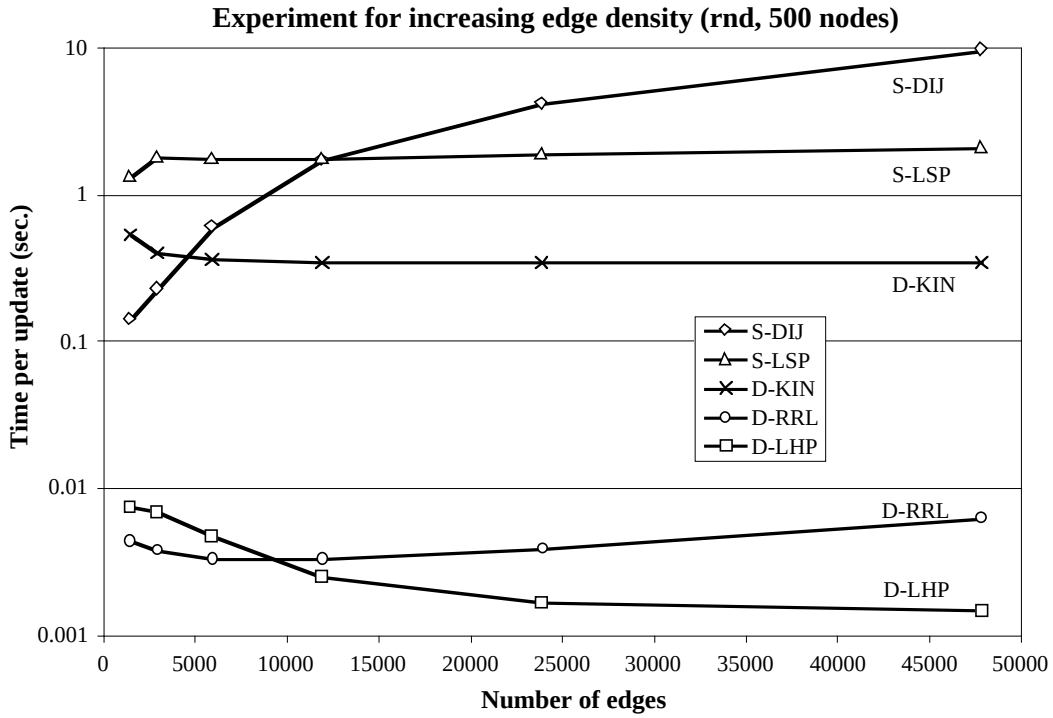


Figure 7: Running times of the different algorithms on random graphs with 500 nodes, increasing densities, and edge weights in the range $[1, 1000]$. `D-KIN` was run with $\beta = 0.6$, and integer weights in the smaller range $[1, 5]$. The experiment was done on an AMD Athlon, 1.6 GHz, 256KB L2 cache, 1GB RAM. The update sequence contained $1,000$ evenly mixed operations; the running times are reported on a logarithmic scale.

4 Overall Discussion

Random inputs.

Our experiments with random graphs pointed out that `D-LHP` and `D-RRL` are the fastest implementations: in this scenario they can be faster than static algorithms (`S-DIJ` and `S-LSP`) even by two to four orders of magnitude, depending on the inputs. Algorithm `D-KIN` was only at most one order of magnitude faster than the static algorithms: this seems to be mainly due to the high overhead caused by maintaining the forest of in/out trees and by stitching paths in the data structure. For sparse random graphs, the running times of `D-RRL` and `D-LHP` are very close, and the underlying computing platform plays a role in deciding which one performs better (see later for a discussion on this). As the graph density increases, and consequently the computational savings of locally shortest paths become more significant (as illustrated also in Figure 5), `D-LHP` becomes the most efficient choice on all the platforms we considered. This can be clearly seen in Figure 7, which depicts the result of an experiment on random graphs with 500 nodes, increasing edge densities, and edge weights in the range $[1, 1000]$. Only the experiment on `D-KIN` was

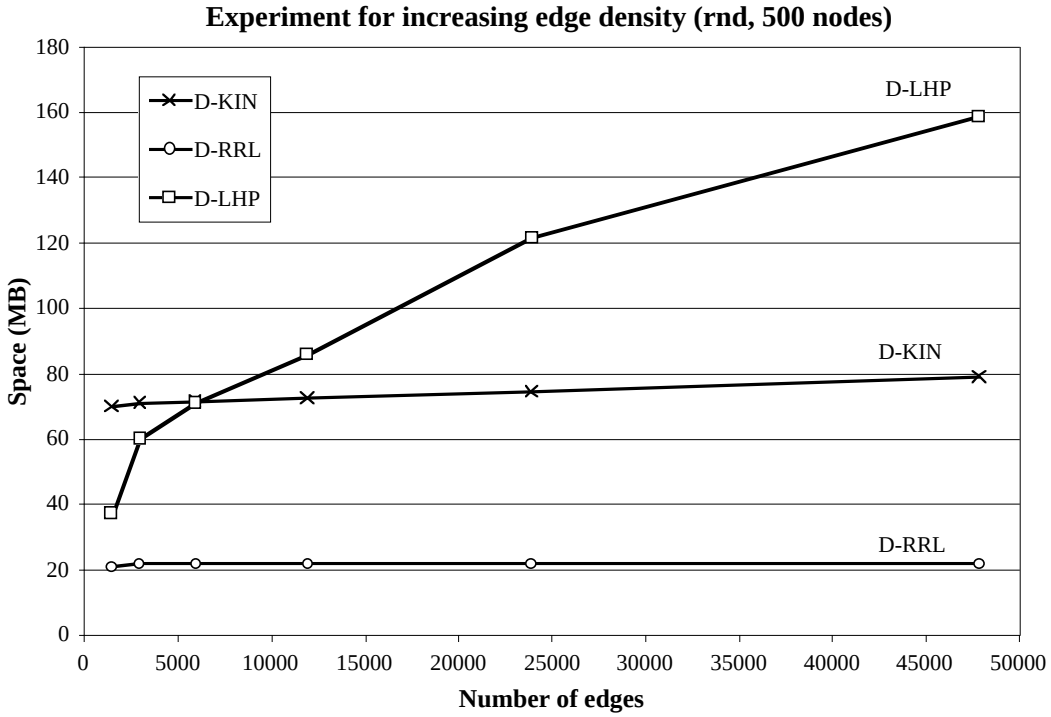


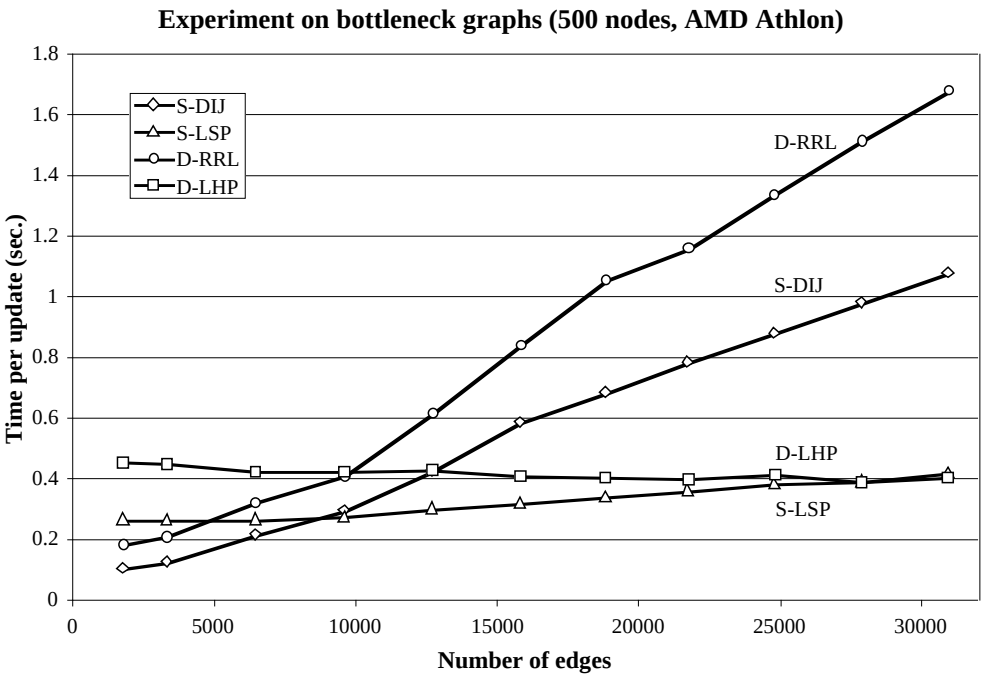
Figure 8: Space usage of the different dynamic algorithms on random graphs with 500 nodes, increasing densities, and edge weights in the range $[1, 1000]$. `D-KIN` was run with $\beta = 0.6$, and integer weights in the smaller range $[1, 5]$.

done with integer weights in the smaller range $[1, 5]$, to avoid the performance degradation of this algorithm for large values of C. Figure 8 reports the space usage of the dynamic implementations: `D-RRL` uses simple data structures, retains little information throughout the sequence of updates, and thus can save space with respect to `D-KIN` and `D-LHP`. Note that, as correctly predicted by theory, the amount of memory used by `D-RRL` ($O(n^2)$) and by `D-KIN` ($O(n^{2.5}\sqrt{C})$) does not depend upon the graph density, while the space requirement of `D-LHP` ($O(|LHP| + n^2)$) depends on the number of locally historical paths and thus varies substantially with the graph and with its density.

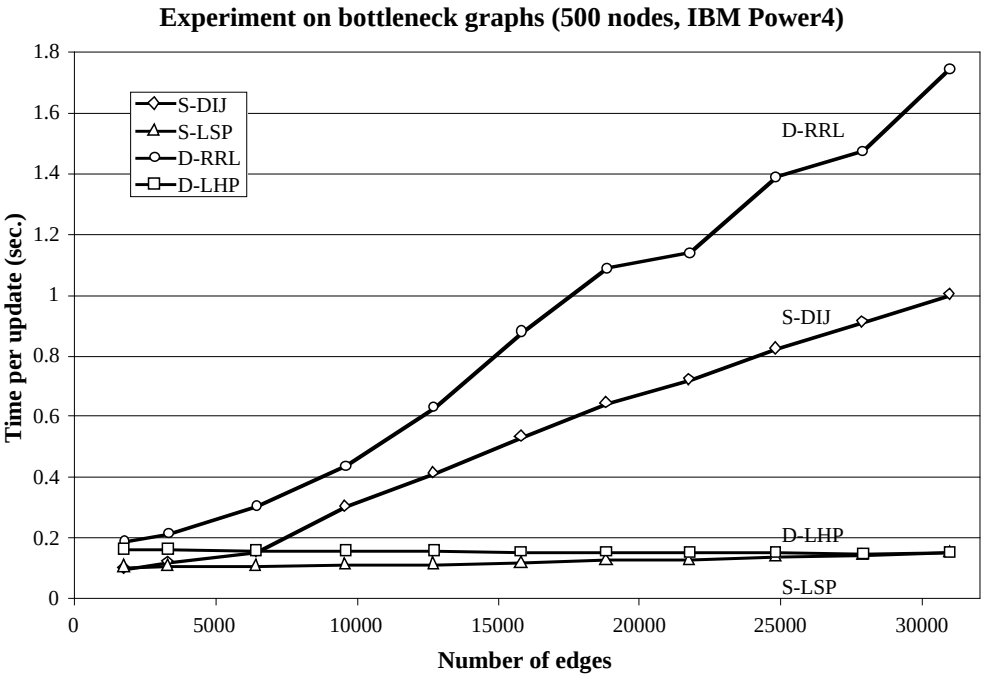
Bottleneck inputs.

Bottleneck inputs (see Figure 1) force a pathological sequence of updates, changing the weight of many shortest paths and causing the algorithm to rescan a large portion of the graph during each update. Figure 9 shows the result of an experiment with bottleneck graphs of different densities on two different platforms: our low-end cache system (256KB L2 cache) and our high-end cache system (1.4MB L2 cache, 32MB L3 cache). We first discuss some features which appear to be architecture-independent, and then analyze the effects of the underlying platforms on the implementations.

As it can be seen from Figure 9, `D-RRL` is hit pretty badly by those inputs, and becomes



(a) Experiment on an AMD Athlon, 1.6 GHz, 256KB L2 cache, 1GB RAM.



(b) Experiment on an IBM Power 4, 1.1 GHz, 1.4MB L2 cache, 32MB L3 cache, 64GB RAM.

Figure 9: Experiment on bottleneck graphs of different sizes. The update sequence contained $1,000$ evenly mixed operations.

even slower than the static implementation of Dijkstra (S-DIJ). This reflects the fact that the worst-case update time of the algorithm by Ramalingam and Reps is asymptotically the same as the static algorithm, and thus the performance of D-RRL can be quite bad on hard instances. On the other side, in those inputs D-LHP becomes comparable to its static version S-LSP. This can be explained by noting that for those inputs most of the computation savings of D-LHP come from locally shortest paths, which are $O(n^2)$ in this case. Since bottleneck inputs force scanning of a large portion of the graph, D-LHP and S-LSP end up performing similar tasks on those test sets.

In all our experiments with bottleneck inputs, D-LHP was substantially faster than either D-RRL or S-DIJ on dense graphs or on platforms with a good cache system. The first case can be easily explained by considering the computational savings of locally shortest paths on dense graphs. To explain the second case, we consider more closely Figure 9 (a) and Figure 9 (b). In the two experiments, the running times of D-RRL and S-DIJ are very similar, while the implementations based on locally shortest paths (D-LHP and S-LSP) have a speedup of more than 2 on the high-end cache system. Since the CPU clocks of the two platforms are comparable, this may suggest that algorithms based on locally shortest paths, which retain more information throughout the updates, seem to benefit more from better memory systems. Moreover, we observe that D-RRL and S-DIJ, besides requiring less space than D-LHP, tend to have a more regular pattern of data access, since they recompute single-source shortest paths starting repeatedly from every node in the graph, as opposed to D-LHP and S-LSP, which compute all shortest paths in parallel, and thus need to access more global data structures. Consequently, D-RRL and S-DIJ seem to suffer less from reduced cache sizes. We will investigate more closely this issue next.

Real-world inputs.

The same general trend observed for random graphs can be seen also in our experiments with real-word graphs: in particular, D-RRL and D-LHP are much faster than the other implementations (1–3 orders of magnitude). This is illustrated in Figure 10, which reports the results of our experiments on US road networks and AS Internet networks with completely random edge updates. The chart shows the running time of all algorithms except for D-KIN, which was omitted from our experiments on real-world inputs due to its prohibitive performance degradation in case of large edge weights. Once again, D-LHP requires more space than D-RRL as illustrated in Figure 11.

The results of our experiments on real-world inputs with edge weight updates within 5% of their original value are illustrated in Figure 12. In this scenario, there seems to be consistent performance gain of D-LHP over D-RRL, which can be explained as follows. When the update range becomes smaller, also the portion of the graph affected by the change is likely to be smaller. D-LHP, which is based on locally shortest paths, seems to be able to exploit more effectively the locality of this change. This is confirmed by the reduced space usage of D-LHP in this framework, as reported in Figure 13.

Similarly to the experiments on bottleneck inputs, the running times of D-LHP and D-RRL can be very close and their relative performance seems to vary on different platforms.

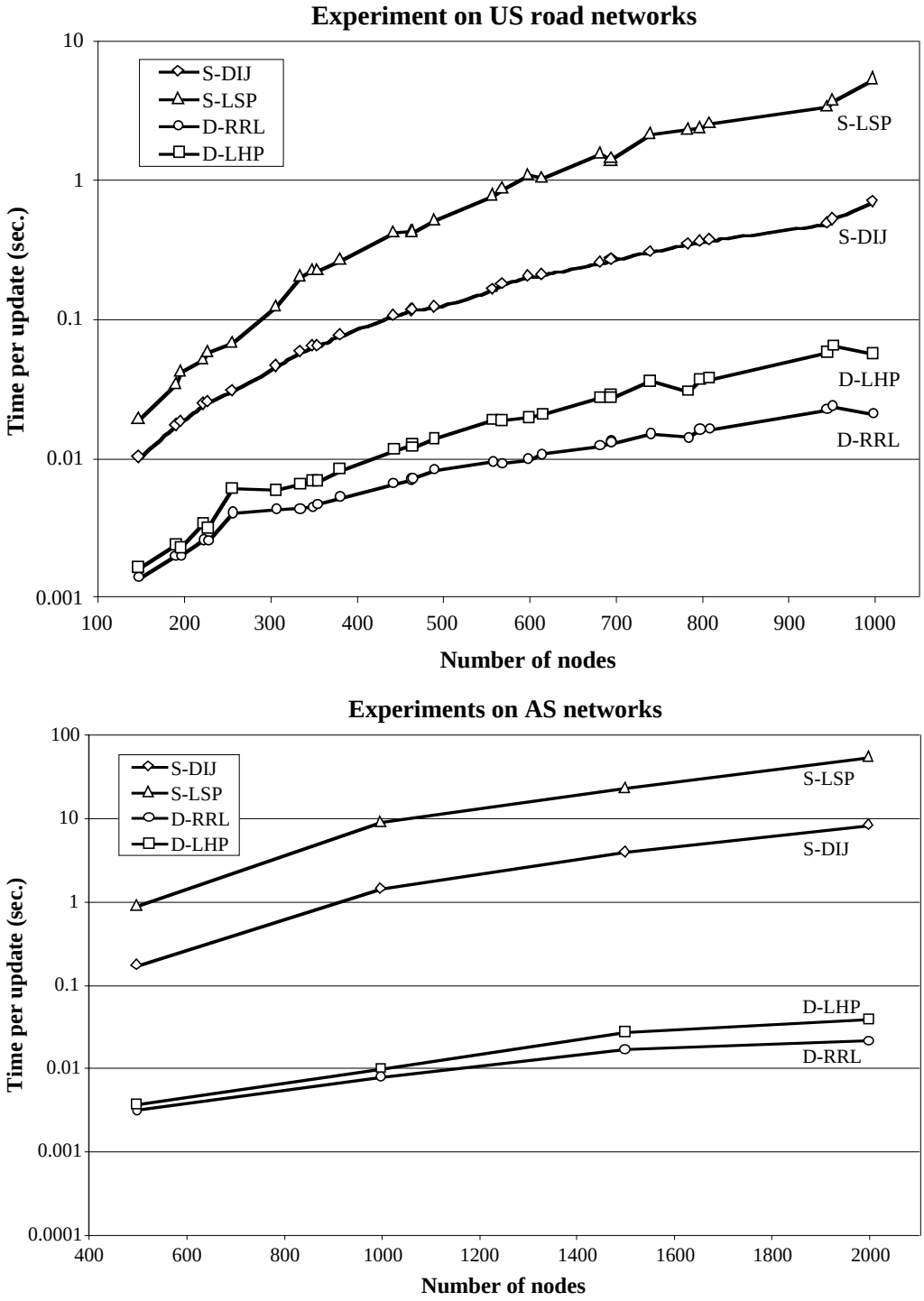


Figure 10: Running times of the different algorithms on US road networks and AS Internet networks under sequences of $1,000$ evenly mixed random updates. `D-KIN` was not applicable since edge weights are too large in these graphs. The experiment was done on an AMD Athlon, 1.6 GHz, 256KB L2 cache, 1GB RAM.

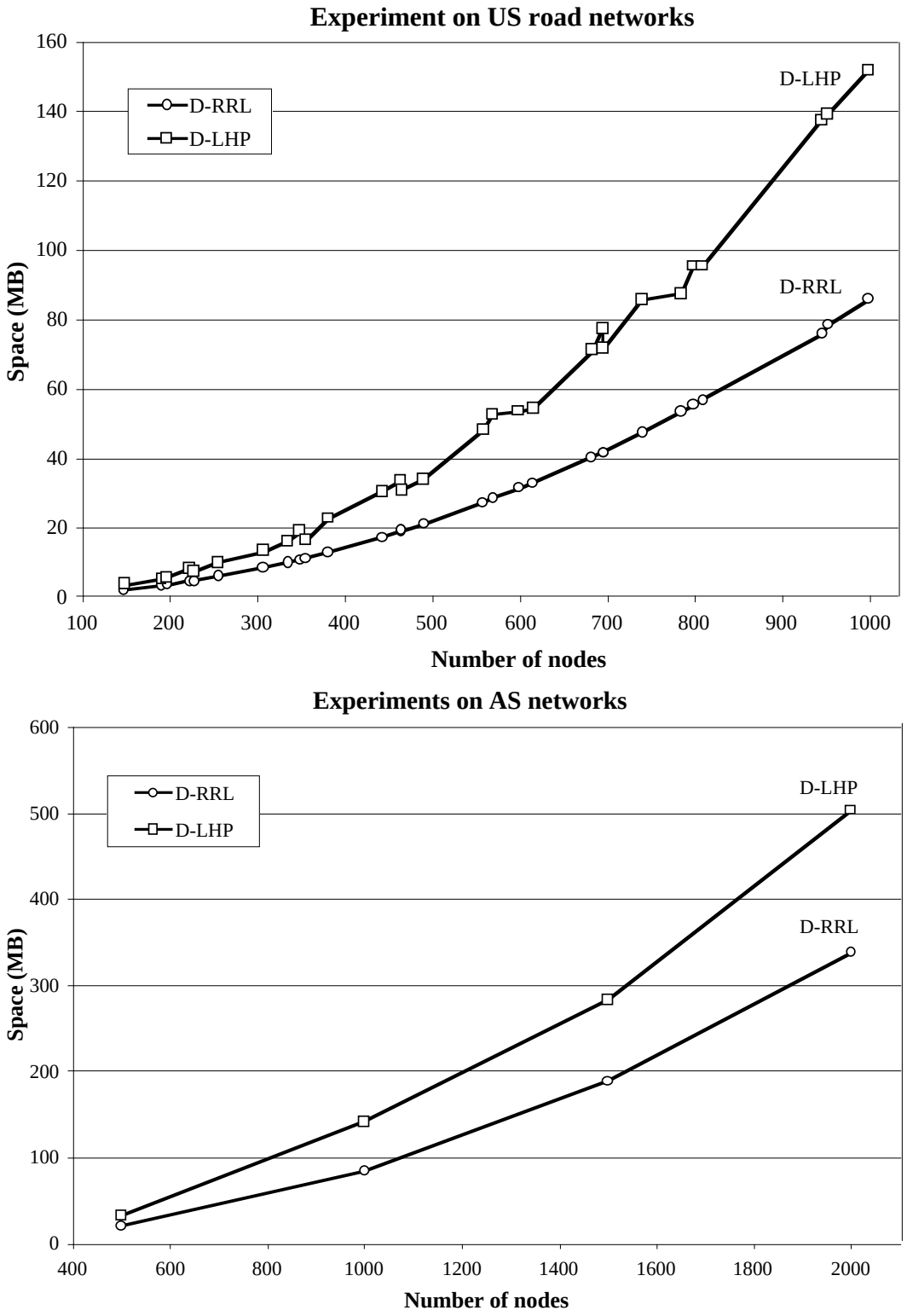


Figure 11: Space usage of `D-LHP` and `D-RLL` on US road networks and AS Internet networks under sequences of $1,000$ evenly mixed random updates.

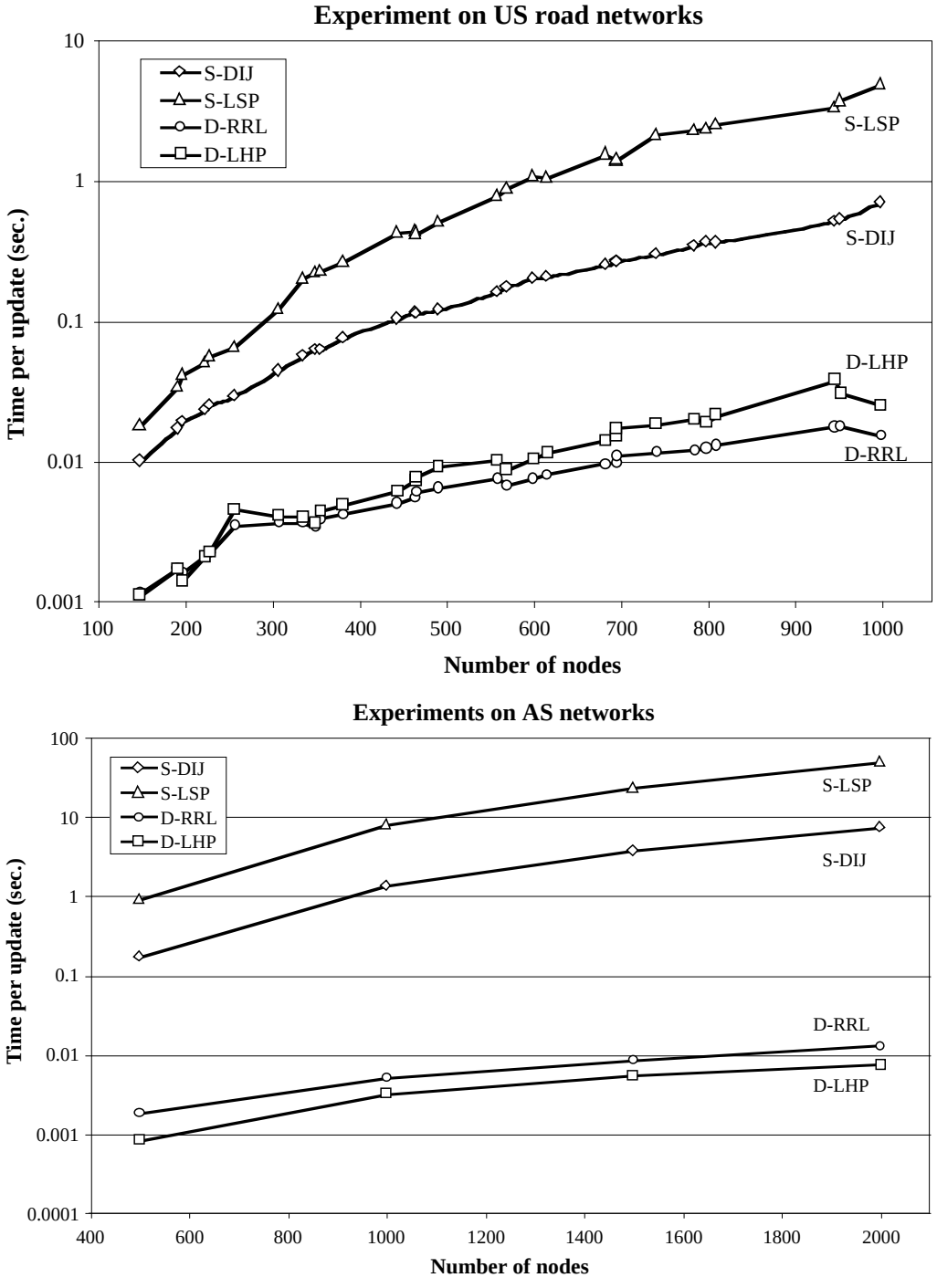


Figure 12: Running times of the different algorithms on US road networks and AS Internet networks under sequences of $1,000$ evenly mixed random updates within 5% of their original value. `D-KIN` was not applicable since edge weights are too large in these graphs. The experiment was done on an AMD Athlon, 1.6 GHz, 256KB L2 cache, 1GB RAM.

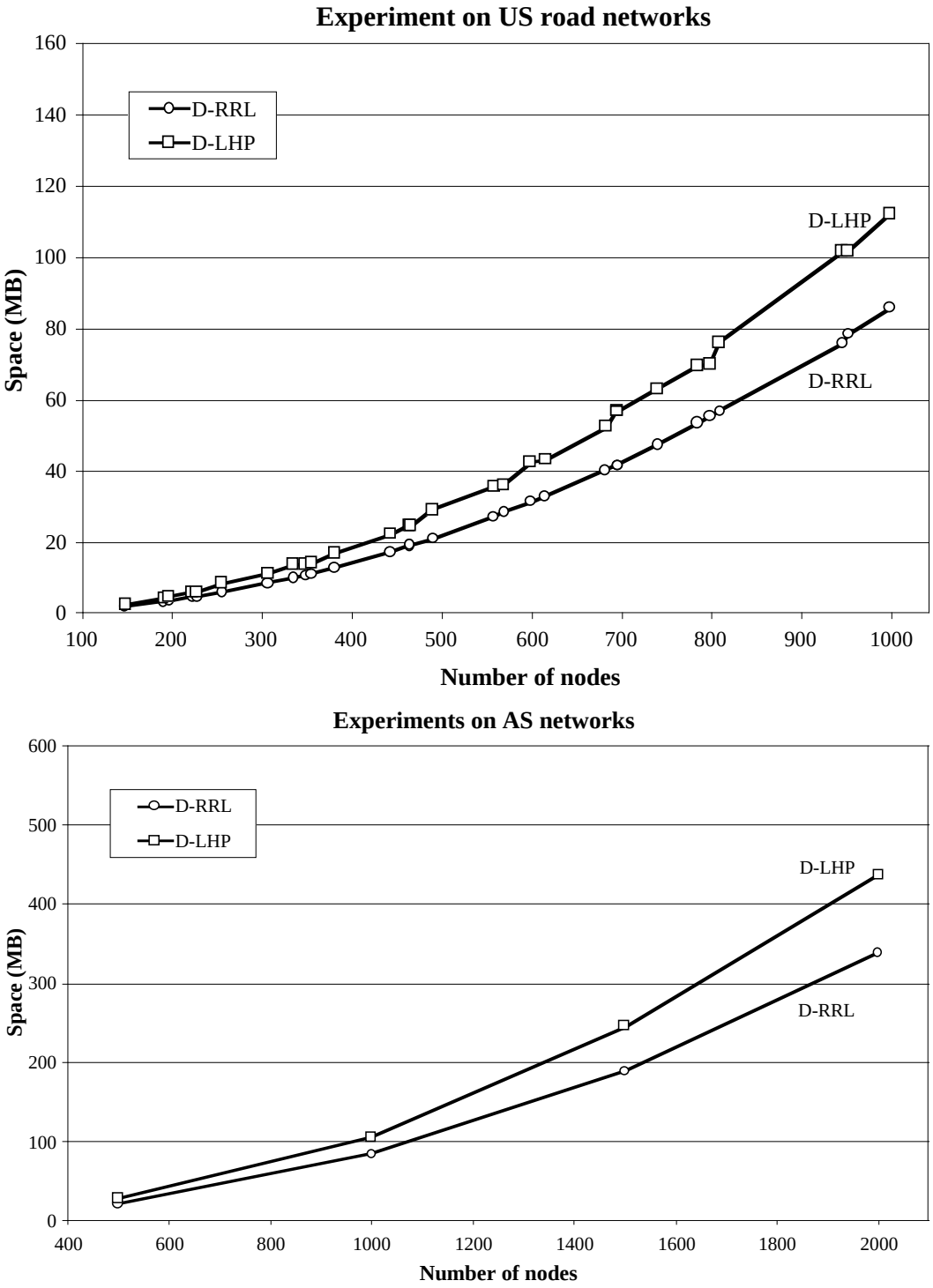


Figure 13: Space usage of `D-LHP` and `D-RLL` on US road networks and AS Internet networks under sequences of $1,000$ evenly mixed random updates within 5% of their original value.

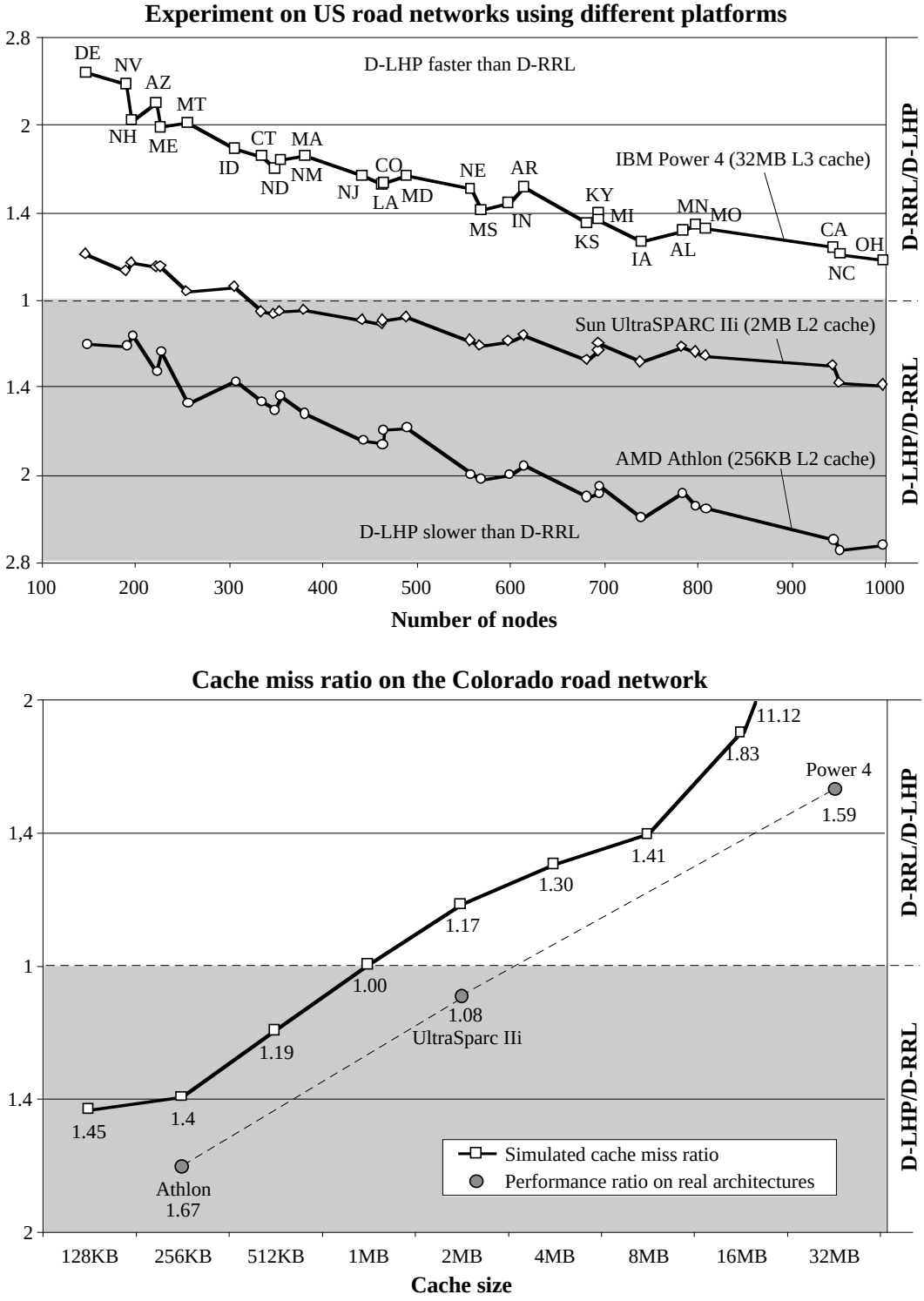


Figure 14: Studying cache effects. (a) Performance ratio max{D-RRL/D-LHP, D-LHP/D-RRL} on the US road networks using architectures with different cache sizes. (b) Simulated cache miss ratio max{D-RRL/D-LHP, D-LHP/D-RRL} on the Colorado road network.

To illustrate this, we refer to Figure 14 (a), which plots the running time of D-RRL versus the running time of D-LHP on the US road networks with completely random edge weight updates measured on platforms with different cache sizes. There are two main phenomena that can be seen from this figure: first, the relative performance of D-LHP and D-RRL tends to change with different cache sizes: i.e., the better the cache system, the faster is D-LHP over D-RRL. Second, on any given platform, D-RRL is likely to become faster than D-LHP as the number of nodes increases. We believe that both phenomena are due to cache effects: D-LHP requires more memory and uses more global data structures, while D-RRL requires less space and exhibits a better locality in the memory access pattern. To investigate more these aspects, we performed an extensive simulation of cache miss ratios with the Cachegrind tool [27], and the simulation results on the Colorado road network are reported in Figure 14 (b). The time ratio measured on different platforms seems to follow the same general trend as the simulated cache miss ratio, which indeed suggests that the relative performance of D-RRL and D-LHP on different machines depends on their different cache usage. The jump from 1.83 to 11.2 in the simulated cache miss ratio for 32MB could be explained by observing that, in our experiments on the Colorado road network, D-LHP requires 30.59MB of main memory; this is close to the simulated cache size, and may cause threshold phenomena.

5 Concluding Remarks

In this paper we have implemented, engineered and evaluated experimentally three different dynamic shortest path algorithms: D-RRL [26], D-KIN [19] and D-LHP [8]. In our experiments, implementing a dynamic algorithm seemed really worth the effort, as all the dynamic implementations are typically much faster than recomputing a solution from scratch with a static algorithm: D-RRL and D-LHP can be up to 10,000 times faster than a repeated application of a static algorithm, while D-KIN can be around 10 times faster than a static algorithm.

The algorithm by Ramalingam and Reps (D-RRL) is basically a variant of Dijkstra's algorithm, and works only on the portion of the graph that is changing throughout updates. Since it uses simple data structures, it seems very hard to beat in situations where the updates produce a very small change in the solution. However, its worst-case running time is asymptotically the same as Dijkstra's algorithm, and thus it can be quite bad in pathological worst-case instances. The algorithm of Demetrescu and Italiano (D-LHP) uses more sophisticated data structures than D-RRL, but it still works only on the portion of the graph that is modified by the updates. It can be as fast as D-RRL on sparse graphs, and it becomes substantially faster than D-RRL on dense graphs and on worst-case inputs, where locally shortest paths seem to gain better payoffs. The main difference in performance with D-RRL on sparse graphs seems related to memory issues, i.e., to the algorithms' space usage and pattern access on data: in particular, in our experiments D-LHP ran faster on platforms with a good memory hierarchy system (i.e., cache and/or memory bandwidth), while D-RRL seemed preferable on platforms with small cache and/or small memory bandwidth. Overall,

D-LHP revealed to be the most robust implementation on different inputs among the ones we tested. As a side effect, we derived from D-LHP a new static algorithm, S-LSP, which can run faster than Dijkstra's algorithm on dense graphs, since in practice it reduces substantially the total number of edges scanned.

One issue that seems to deserve further theoretical and empirical study is memory usage. To answer queries fast, all the algorithms maintain explicitly the all pairs shortest paths matrix. This makes them hit very soon a "memory wall" in today's computing platforms, thus limiting substantially the maximum problem size that can be solved in practice. Just to make an example, if an implementation requires around 100 bytes per each pair of nodes in the graph (for instance the space usage of D-LHP is roughly 80 bytes per locally historical path, plus 24 bytes per pair of nodes) on a memory system with 10 GB of RAM we can only solve instances of up to 10,000 nodes without incurring memory swap problems. These were indeed the larger graphs that we were able to consider in our computational study.

Acknowledgments

We are indebted to the anonymous reviewers for many useful comments. We wish to thank Stefano Emiliozzi for his valuable help in the implementation and experimental analysis of some of the algorithms discussed in this paper. We deeply acknowledge the generous hospitality of Federico Massaioli and CASPUR (Consorzio interuniversitario per le Applicazioni di Supercalcolo Per Università e Ricerca), which let us run experiments on their IBM Power 4.

References

[1] D. Alberts, G. Cattaneo, and G.F. Italiano. An empirical study of dynamic graph algorithms. *ACM Journal on Experimental Algorithmics*, 2(5), 1997.

[2] G. Amato, G. Cattaneo, and G.F. Italiano. Experimental analysis of dynamic minimum spanning tree algorithms. In *Proc. SODA'97*, pages 314–323, 1997.

[3] L. Buriol, M. Resende, and M. Thorup. Speeding up dynamic shortest path algorithms. In *AT&T Labs Research Report TD–5RJ8B*, September 2003.

[4] Cattaneo, G. and Faruolo, P. and Ferraro-Petrillo, U. and Italiano, G. F. Maintaining dynamic minimum spanning trees: An experimental study. In *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments* (ALENEX'02), pages 111-125, 2002.

[5] B.V. Cherkassky, A.V. Goldberg, and T. Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Math. Programming*, 73:129–174, 1996.

[6] C. Demetrescu, D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Maintaining shortest paths in digraphs with arbitrary arc weights: An experimental study. In *Proceedings of the 4th Workshop on Experimental Algorithmics* (WAE'00), pages 218–229, 2000.

[7] C. Demetrescu and G.F. Italiano. Fully dynamic all pairs shortest paths with real edge weights. In *Proc. FOCS'01*, pages 260–267, 2001.

[8] C. Demetrescu and G.F. Italiano. A new approach to dynamic all pairs shortest paths. In *Proceedings of STOC'03*, pages 159–166, 2003.

[9] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Math.*, 1:269–271, 1959.

[10] S. Even and Y. Shiloach. An on-line edge-deletion problem. *Journal of the ACM*, 28:1–4, 1981.

[11] B. Fortz and M. Thorup. Internet traffic engineering by optimizing OSPF weights. In *Proceedings INFOCOM'00*, pages 519–528, 2000.

[12] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their use in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

[13] D. Frigioni, M. Ioffreda, U. Nanni, and G. Pasqualone. Analysis of dynamic algorithms for the single source shortest path problem. *ACM Journal of Experimental Algorithmics* (JEA), 3, 1998.

[14] D. Frigioni, T. Miller, U. Nanni, G. Pasqualone, G. Schäfer, and C.D. Zaroliagis. An experimental study of dynamic algorithms for directed graphs. In *Proc. ESA'98*, pages 320–331, 1998.

[15] A.V. Goldberg. Shortest path algorithms: Engineering aspects. In *Proc. ISAAC'01*, pages 502–513, 2001.

[16] D. H. Greene and D.E. Knuth. *Mathematics for the analysis of algorithms*. Birkhäuser, 1982.

[17] R. Iyer, D. R. Karger, H. S. Rahul, and M. Thorup. An Experimental Study of Polylogarithmic, Fully Dynamic, Connectivity Algorithms. *ACM Journal of Experimental Algorithmics* (JEA), 6, 2001.

[18] D. Karger, D. Koller, and S.J. Phillips. Finding the hidden path: Time bounds for all-pairs shortest paths. *SIAM Journal on Computing*, 22(6):1199–1217, 1993.

[19] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proc. FOCS'99*, pages 81–99, 1999.

[20] V. King and M. Thorup. A space saving trick for directed dynamic transitive closure and shortest path algorithms. In *Proc. COCOON'01*, pp. 268–277, 2001.

[21] P. Loubal. A network evaluation procedure. *Highway Research Record 205*, pages 96–109, 1967.

[22] J. Murchland. The effect of increasing or decreasing the length of a single arc on all shortest distances in a graph. Technical report, LBS-TNT-26, London Business School, Transport Network Theory Unit, 1967.

[23] Paolo Narvaez, Kai-Yeung Siu, and Hong-Yi Tzeng. New dynamic algorithms for shortest path tree computation. *IEEE/ACM Transactions on Networking*, 8:734–746, 2000.

[24] Paolo Narvaez, Kai-Yeung Siu, and Hong-Yi Tzeng. New dynamic SPT algorithm based on a ball-and-string model. *IEEE/ACM Transactions on Networking*, 9:706–718, 2001.

[25] G. Ramalingam. Bounded incremental computation. *Lecture Notes in Computer Science 1089*, 1996.

[26] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest path problem. *Journal of Algorithms*, 21:267–305, 1996.

[27] J. Seward and N. Nethercote. Valgrind, an open-source memory debugger for x86-gnu/linux. URL: `http://developer.kde.org/~sewardj/`.

[28] M. Thorup. Fully-Dynamic All-Pairs Shortest Paths: Faster and Allowing Negative Cycles. Proceedings of the 9th Scandinavian Workshop on Algorithm Theory (SWAT'04), Humlebæk, Denmark, July 8–10, 2004.

[29] F. Zhan and C. Noon. Shortest path algorithms: An evaluation using real road networks. *Transportation Science*, 32(1):65–73, 1998.