Alcom-FT Technical Report Series
ALCOMFT-TR-03-9

Experimental Analysis of Dynamic All Pairs Shortest Path Algorithms *

Camil Demetrescu † *Stefano Emiliozzi* ‡ *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 [18] and of Demetrescu and Italiano [7], and compare them to the dynamic algorithm of Ramalingam and Reps [25] 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., [24]), 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 [10, 22, 23].

This problem was first studied in 1967 [20, 21, 26], 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 [8, 11], where m is the number of edges and n is the number of nodes. Since m can be as high as $O(n^2)$, this is $O(n^3)$ in the worst case. In 1999 King [18] 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 [6] showed how to maintain all pairs shortest paths on directed graphs with real-value edge weights that can assume at most S different values: their bound is $O(n^{2.5}\sqrt{S \log^3 n}\,)$ per update and $O(1)$ per query. Very recently, the same authors [7] presented a fully dynamic shortest paths algorithm that requires $\widetilde{O}(n^2)$[1] per update and constant time per query.

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 [18] and of Demetrescu and Italiano [7] with a thorough empirical study. In particular, we present and experiment with efficient implementations of dynamic shortest path algorithms. Our empirical analysis shows that

*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"). The generous hospitality of CASPUR, which let us run experiments on some of their machines, is deeply acknowledged.

†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 e Sistemistica, Università di Roma "La Sapienza", Roma, Italy. Email: `thepomy@tin.it`.

§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`.

[1] Throughout the paper, we use $\widetilde{O}(f(n))$ to denote $O(f(n)\mathrm{polylog}(n))$.

some of the dynamic algorithms and their algorithmic 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 the 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., [4, 14, 28]), many researchers have been complementing the wealth of theoretical results on dynamic graphs with thorough empirical studies, in the effort of bridging the gap between the design and theoretical analysis and the actual implementation, experimental tuning and practical performance evaluation of dynamic graph algorithms. In particular, Frigioni *et al.* [13] proposed efficient implementations of dynamic transitive closure and shortest path algorithms, while Frigioni *et al.* [12] and later Demetrescu *et al.* [5] 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. Other dynamic graph implementations include the work of Alberts *et al.* [1] and Iyer *et al.* [16], who implemented and evaluated experimentally algorithms for fully dynamic connectivity problems, and the work of Amato *et al.* [2] and Cattaneo *et al.* [3], who proposed and analyzed efficient implementations of dynamic MST algorithms.

2 Experimental Setup

2.1 Test Sets

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

Random inputs. We considered random graphs with n nodes, m edges, and random integer edge weights. 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.

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, and represent physical distances. 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 in real-world graphs were random weight updates on their edges.

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, $i = 1, 2$. 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 $x \in X_2$. In this way, most shortest paths in the graph go through edge (u, v). 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 so that the algorithms have to rescan the whole graph at each update.

2.2 Computing Platforms

To assess the experimental performance of our implementations on different architectures, we experimented on a variety of computing platforms:

- AMD Athlon, 1.5 GHz, 256KB L2 cache, 512MB RAM.
- Intel Xeon, 500 MHz, 512KB L2 cache, 512MB RAM.
- Intel Pentium 4, 2 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 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 was significant, we will point it out explicitly.

3 Algorithms Implemented

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

Algorithm	Reference	Update Time	Query Time	Space
`S-DIJ`	[8, 11]	$O(mn + n^2 \log n)$	$O(1)$	$O(n^2)$
`S-UP`	This paper	$O(\|UP\| + n^2 \log n)$, $\|UP\| \leq mn$	$O(1)$	$O(n^2)$
`D-RRL`	[25]	$O(mn + n^2 \log n)$	$O(1)$	$O(n^2)$
`D-KIN`	[18]	$\widetilde{O}(n^{2.5}\sqrt{C})$, C =max weight	$O(1)$	$\widetilde{O}(n^{2.5}\sqrt{C})$
`D-PUP`	[7]	$\widetilde{O}(n^2)$	$O(1)$	$\widetilde{O}(mn)$

The algorithms were implemented in C following a uniform programming style and using exactly the same data structures 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 heaps). 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/`.

We used Dijkstra's single-source algorithm [8, 11] repeated from every node as a static reference code (`S-DIJ`) to evaluate the performances of the other algorithms in a dynamic setting. Algorithms `D-RRL`, `D-KIN`, `D-PUP`, and `S-UP` are described below.

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

The algorithm by Ramalingam and Reps [25] 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. Note that 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., [5, 10, 12]). In our implementation, we considered a further simplified version of the original algorithm, which was previously described in [5]. 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 [5]) than the original algorithm in [25].

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

The dynamic shortest paths algorithm by King [18] works on directed graphs with small integer edge 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 $\leq d$.

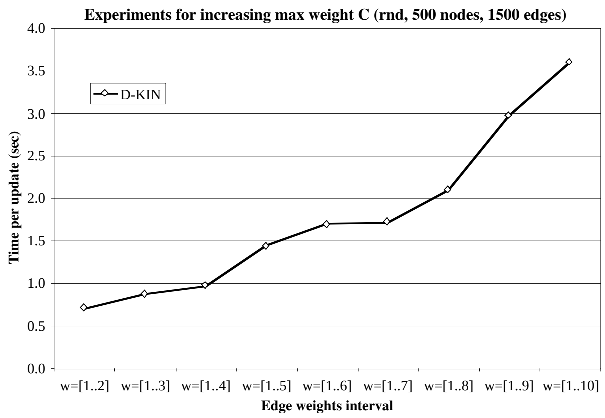
To maintain shortest paths up to distance d, the algorithm keeps a pair of in/out shortest paths trees $IN(v)$ and $OUT(v)$ of depth $\leq d$ rooted at each node v. Trees $IN(v)$ and $OUT(v)$ are maintained with a variant of the decremental data structure by Even and Shiloach [9]. It is easy to prove 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 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., [15]): 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 [18] requires $O(n^3)$ space, in [19] 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 [18, 19].

Our implementation. Following the techniques for space reduction [19], 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 1 showed that going from a random graph with 500 nodes, 1500 edges, and maximum weight $C = 2$ to a graph with the same size and $C = 10$ can degradate performances by a factor of 7, while requiring twice as much space. 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 inputs.



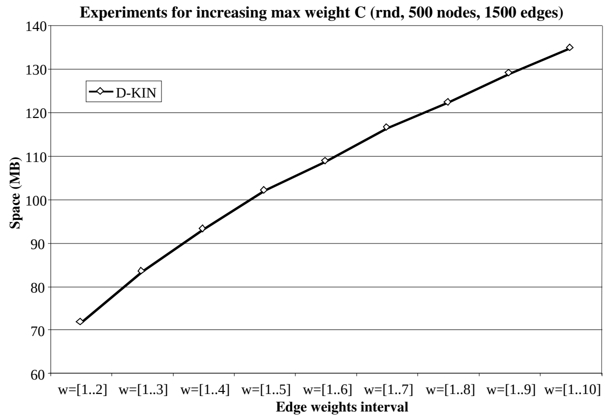


Figure 1: Algorithm D-KIN: dependence of the time per update and space on the maximum edge weight C. The experiment was done on an AMD Athlon, 1.5 GHz, 256KB L2 cache, 512MB RAM. The update sequence contained 100 evenly mixed operations on a random graph with 500 nodes and 1500 edges.

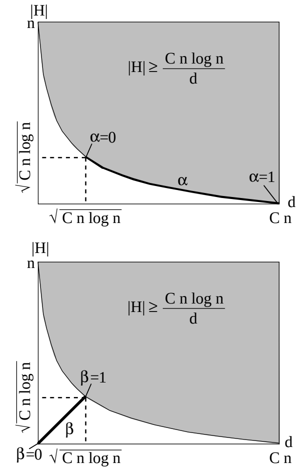
Profiling information revealed that with the standard setting $|H| = d = \sqrt{C \log n}$, typically more than 75% of the update time is required by the path stitching procedure. We thus expected a reduction of $|H|$ to yield substantial benefits in the running time, and this was fully confirmed by our experiments. As an example, Figure 2 (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 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 [15], with this set of parameters we found no errors in the maintained distances.

Another interesting question was how far we can decrease both $|H|$ and d without incurring in 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 in distance errors, with substantial time and space improvements (see Figure 2 (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-PUP)

The dynamic shortest path algorithm in [7] works on directed graphs with nonnegative real-valued edge weights and hinges on the notion of *uniform paths* (UP): we say that a path π is uniform 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 *potentially uniform* if every proper subpath of π is a historical shortest path. The main idea behind the algorithm is to maintain dynamically the set of potentially uniform paths (PUP), which include uniform paths and shortest paths as special cases. The following theorem from [7] bounds the number of paths that become potentially uniform 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 potentially uniform at each update is $O(zn^2)$.*



(a)

alpha	time (sec)	space (MB)	avg. error
0.00	2.59	135	0
0.01	1.41	181	0
0.02	1.11	227	0
0.03	1.01	274	0
0.04	0.89	320	0
0.05	0.85	367	0

(b)

beta	time (sec)	space (MB)	avg. error
1.0	2.59	135	0
0.9	2.06	123	0
0.8	1.58	112	0
0.7	1.51	101	0
0.6	1.15	90	0
0.5	0.98	79	3.431
0.4	0.94	67	4.468
0.3	0.57	54	6.904

Figure 2: Algorithm `D-KIN`: parameter tuning with α and β. The experiment was done on an AMD Athlon, 1.5 GHz, 256KB L2 cache, 512MB RAM. The update sequence contained 100 evenly mixed operations.

To keep changes in potentially uniform 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 [7]). This technique implies that only $O(n^2 \log k)$ paths become potentially uniform 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 [8] 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 potentially uniform paths. The update algorithm spends $O(\log n)$ time for each of the $O(zn^2)$ new potentially uniform 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 $O(mn \log n)$ potentially uniform 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 [7] for the low-level details of the method.

Our implementation. Our implementation (`D-PUP`) followed closely the theoretical algorithm in [7], with one additional heuristic: we stopped performing smoothing whenever the ratio between the number of created PUPs and deleted PUPs 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. This was confirmed by our experiments: indeed smoothing did not appear to be of any benefit in random and real-world inputs, as they seemed to contain no pathological instances.

To assess the effects of smoothing, we performed another experiment by changing the way the data structure initialization was done. Namely, rather than starting from an initial data structure having only one (historical) shortest path per pair, we built the graph via edge insertions, in order to force artificially the data structure to contain initially a very high number of 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, Figure 3 illustrates the effects of smoothing on the Utah road network (269 nodes and

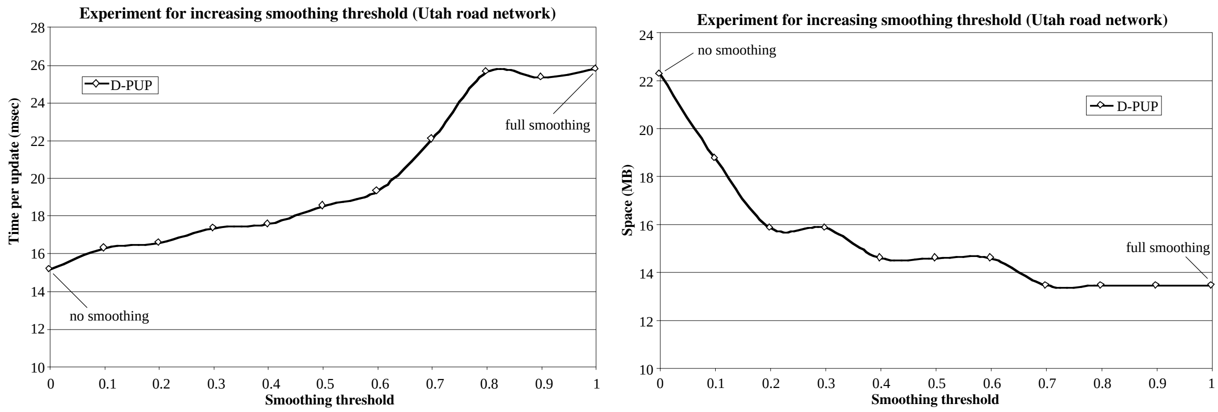


Figure 3: Algorithm D-PUP: effects of varying the degree of smoothing on the Utah road network initialized from an empty graph via edge insertions in order to start with many historical shortest paths. The experiment was done on an Intel Xeon 500MHz, 512KB L2 cache, 512MB RAM. The update sequence contained $1,000$ evenly mixed operations (after the initialization).

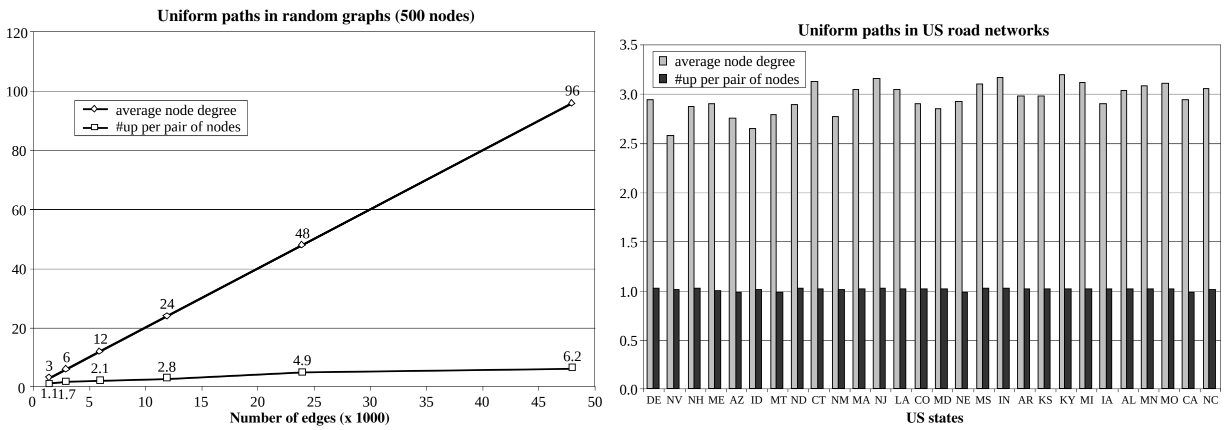


Figure 4: Counting uniform paths in random and real-world graphs.

742 edges) initialized from an empty graph via edge insertions. This suggests that a certain degree of smoothing can be useful only when there can be a high number of historical shortest paths in the data structure. Since this possibility did not appear frequently in our experiments, we used a D-PUP code with smoothing threshold set to 0 (i.e., no smoothing).

3.4 A New Static Algorithm Based on Uniform Paths (S-UP)

Another interesting issue to investigate experimentally was related to the number of uniform 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 both random and real-world graphs they tend to be very close to n^2, i.e., there is typically only one uniform path between any pair of nodes, which is also a shortest path (see Figure 4).

This suggested that uniform paths could be exploited also for static all pairs shortest path algorithms. In particular, we investigated how to deploy uniform 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 dense graphs. We designed a static algorithm, which we refer to as S-UP, that essentially runs Dijkstra's algorithm in parallel from all nodes and scans only edges in uniform paths to reduce the overall work. The running time of S-UP

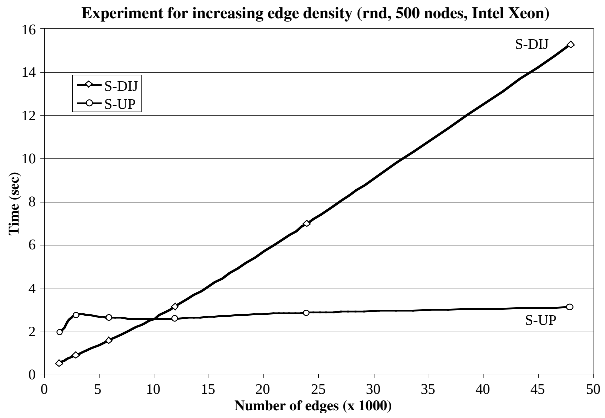


Figure 5: `S-UP` versus `S-DIJ` on a random graph with 500 nodes, increasing densities, and edge weights in the range $[1, 1000]$. The experiment was done on an Intel Xeon 500MHz, 512KB L2 cache, 512MB RAM. The update sequence contained $1,000$ evenly mixed operations.

is $O(|UP| + n^2 \log n)$, where $|UP|$ is the number of uniform paths in the graph. This can yield substantial time savings whenever $|UP| << mn$. For instance, in Figure 5 we report the results of an experiment comparing `S-UP` 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-UP` can be substantially faster than `S-DIJ`.

Note that `S-UP` follows a similar approach to the Hidden Paths Algorithm by Karger *et al.* [17]: their algorithm runs in $O(m^* n + n^2 \log n)$ time, where m^* is the number of edges that participate in shortest paths. By the optimal-substructure property of uniform paths, it is easy to see that $|UP| \leq m^* n \leq mn$. We remark that, in case of unweighted graphs, $m^* = m$, while $|UP|$ might be much smaller than mn.

4 Discussion

Random inputs. Our experiments with random graphs pointed out that `D-PUP` and `D-RRL` are the fastest implementations: in this scenario they can be faster than static algorithms (`S-DIJ` and `S-UP`) 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-PUP` are very close, and the underlying computing platform plays a role in deciding which one is to be used (see later for a discussion on this). As the graph density increases, and consequently the computational savings of uniform paths become more significant (as illustrated also in Figure 4), `D-PUP` becomes the most efficient choice on all the platforms we considered. This can be seen in Figure 6, 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 done with integer weights in the smaller range $[1, 5]$, to avoid the performance degradation of this algorithm for large values of C. Figure 6 reports also 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-PUP`. 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 substantially upon the graph density, while the space requirement of `D-PUP` ($O(|PUP| + n^2)$) depends on the number of potentially uniform paths and thus varies with the graph and with its density.

Real-world inputs. The same general trend can be observed also in our experiments with real-word graphs: in particular, `D-RRL` and `D-PUP` are much faster than the other implementations,

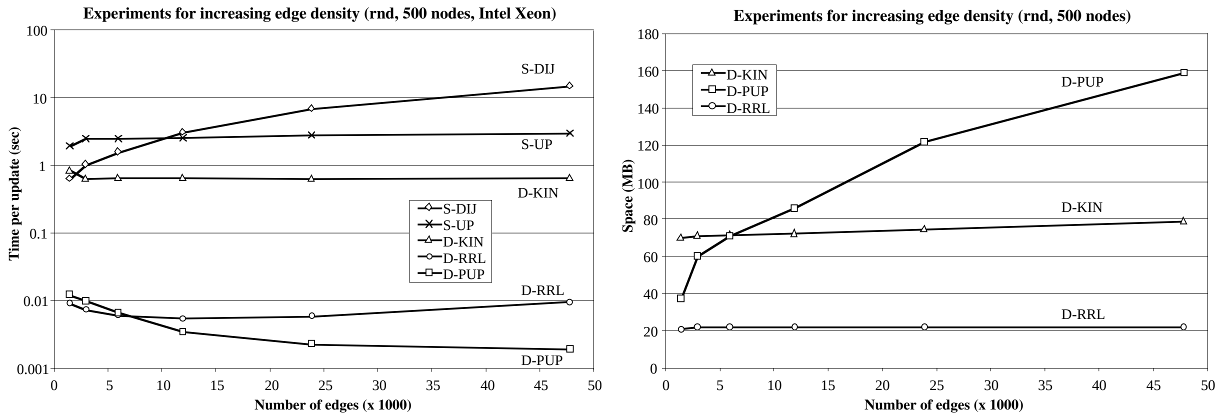


Figure 6: Overall comparison on random graphs with 500 nodes, increasing densities, and edge weights in the range $[1, 1000]$. D-KIN was run with $\beta = 0.6$, and integer weights in the smaller range $[1, 5]$. The experiment was done on an Intel Xeon 500MHz, 512KB L2 cache, 512MB RAM. The update sequence contained $1,000$ evenly mixed operations and running times are reported on a logarithmic scale.

and their running times may get very close, so that their relative performance depends on the underlying computing platform. In particular, Figure 7 (a) shows the results of an experiment on road networks run on an IBM Power 4 (1.4MB L2 cache, 32MB L3 cache, 64GB RAM), while Figure 7 (b) shows the results of the same experiment run on an UltraSPARC IIi (2MB L2 cache, 512MB RAM). In both cases, we have that the larger is the graph, the more D-RRL is likely to become faster than D-PUP. However, with a better memory system (e.g., on the IBM Power 4) D-PUP tends to remain competitive on larger instances. Our experiments on Internet graphs (see, e.g., Figure 7 (d)) confirmed this trend. Figure 7 (c) shows the space usage of the two implementations on the road networks.

Figure 8 (a) plots the relative time performance of the two implementations (i.e., running time of D-RRL over running time of D-PUP) on the US road networks measured on platforms with different cache sizes: as it can be clearly seen from this figure, the speed-up of D-PUP over D-RRL tends to increase with increasing cache sizes. To investigate more this phenomenon, 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 8 (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-PUP on different machines depends on their different cache usage. As a possible explanation for this, we observe that D-RRL, besides requiring less space than D-PUP, tends to have a more regular pattern of access in its data, since it recomputes single-source shortest paths starting repeatedly from every node in the graph, as opposed to D-PUP, which updates all shortest paths in parallel, and thus needs to access more global data structures. Consequently, D-RRL seems to suffer less from reduced cache sizes.

Bottleneck inputs. Bottleneck inputs force a pathological sequence of updates, changing the weight of many shortest paths and causing the algorithm to rescan a constant fraction of the graph during each update. Figure 9 shows the result of an experiment with bottleneck graphs of different densities. As it can be seen from this figure, D-RRL is hit pretty badly by those inputs, and becomes 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 these inputs D-PUP is faster than either D-RRL or S-DIJ, but becomes comparable to its static version S-UP. This can be explained by noting that for those inputs most of the computation savings of D-PUP come from uniform paths, which are $O(n^2)$ in this case. Since bottleneck inputs force scanning of a great portion of the graph, D-PUP and S-UP end up performing similar tasks on those test sets.

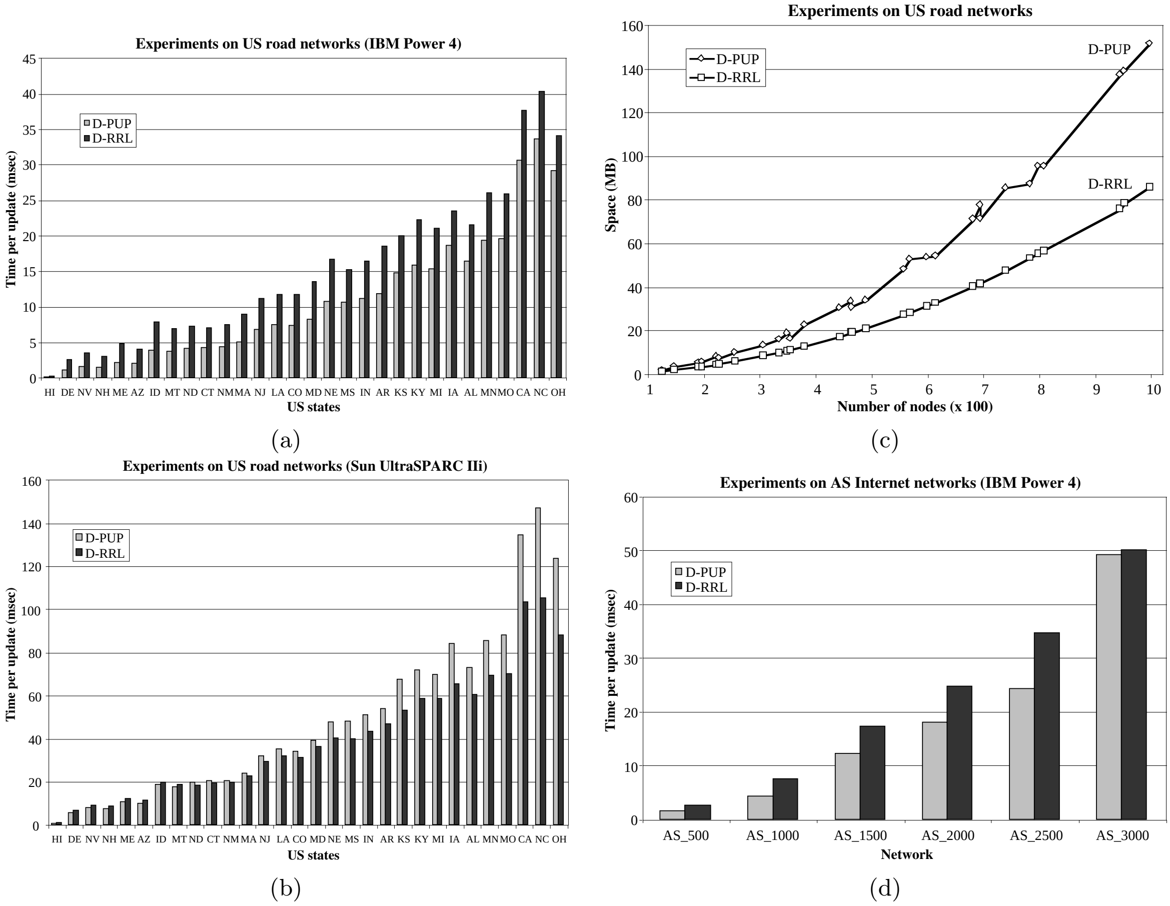


Figure 7: D-PUP versus D-RRL in real-world graphs under sequences of 1,000 evenly mixed updates. (a), (b) Update times of D-PUP and D-RRL on the US road networks measured on an AMD Athlon, 1.5GHz, 256KB L2 cache, 512MB RAM, and on an IBM Power 4, 1.1GHz, 1.4MB L2 cache, 32MB L3 cache, 65GB RAM. (c) Space usage of D-PUP and D-RRL on the US road networks. (d) Update times of D-PUP and D-RRL on the Internet networks measured on an IBM Power 4.

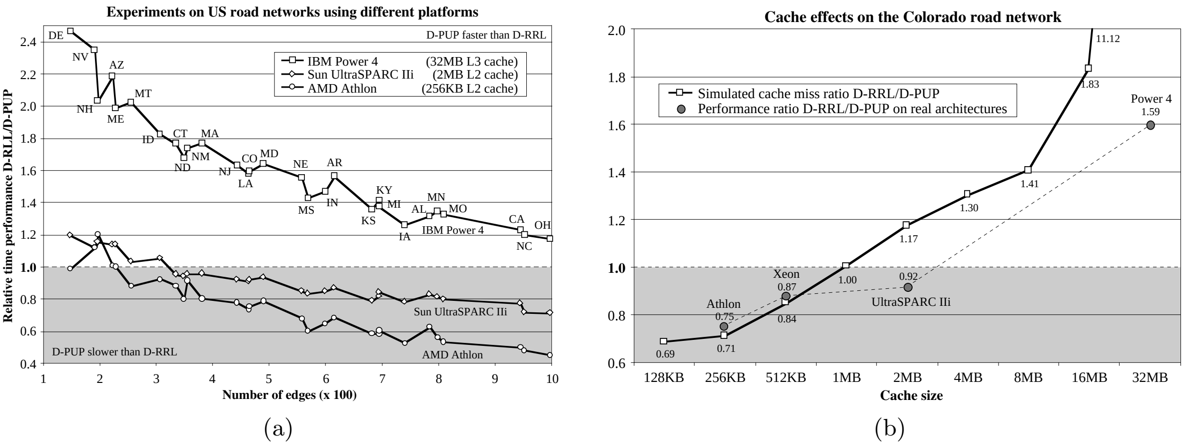


Figure 8: Studying cache effects. (a) Performance ratio D-RRL/D-PUP on the US road networks using architectures with different cache sizes. (b) Simulated cache miss ratio D-RRL/D-PUP on the Colorado road network.

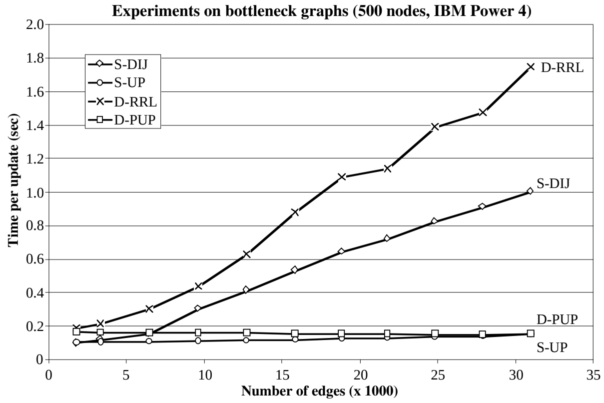


Figure 9: Experiment on bottleneck graphs of different sizes. This experiment was done on an IBM Power 4, 1.4MB L2 cache, 32MB L3 cache, 64GB RAM. The update sequence contained $1,000$ evenly mixed operations.

5 Conclusions

In this paper we have implemented, engineered and evaluated experimentally three different dynamic shortest path algorithms: D-RRL [25], D-KIN [18] and D-PUP [7]. In our experiments, implementing a dynamic algorithm seemed really worth the effort, as all the dynamic implementations can be much faster than recomputing a solution from scratch with a static algorithm: D-RRL and D-PUP 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 of 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-PUP) 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 uniform 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, (1) D-RRL is likely to become faster than D-PUP as the number of nodes increases; and (2) in our experiments, D-PUP 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-PUP revealed to be the most robust implementation on different inputs among the ones we tested. As a side effect, we derived from D-PUP a new static algorithm, 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 the memory usage of dynamic all pairs shortest paths algorithms. To answer queries fast, all these algorithms maintain explicitly the all pairs shortest paths matrix. This makes the algorithms 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-PUP is roughly 80 bytes per potentially uniform 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 in memory swap problems. These were indeed the larger graphs that we were able to consider in our computational study.

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. 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'97)*, pages 314–323, 1997.

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

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

[5] 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 4-st Workshop on Algorithm Engineering (WAE'00), Saarbrücken, Germany, September 5-8*, 2000.

[6] C. Demetrescu and G.F. Italiano. Fully dynamic all pairs shortest paths with real edge weights. In *Proc. of the 42nd IEEE Annual Symposium on Foundations of Computer Science (FOCS'01), Las Vegas, Nevada*, pages 260–267, 2001.

[7] C. Demetrescu and G.F. Italiano. A new approach to dynamic all pairs shortest paths. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing (STOC'03), San Diego, CA*, pages 159–166, 2003.

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

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

[10] B. Fortz and M. Thorup. Internet traffic engineering by optimizing OSPF weights. In *Proceedings of the 19th IEEE INFOCOM - The Conference on Computer Communications*, pages 519–528, 2000.

[11] 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.

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

[13] D. Frigioni, T. Miller, U. Nanni, G. Pasqualone, G. Shaefer, and C.D. Zaroliagis. An experimental study of dynamic algorithms for directed graphs. In *Proc. European Symposium on Algorithms (ESA'98), LNCS 1461*, pages 320–331, 1998.

[14] A.V. Goldberg. Shortest path algorithms: Engineering aspects. In *Proc. 12th International Symposium on Algorithms and Computation (ISAAC'01), LNCS 2223*, 2001.

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

[16] R. Iyer, D. R. Karger, H. S. Rahul, and M. Thorup. An experimental study of poly-logarithmic fully-dynamic connectivity algorithms. In B.E. Moret and A.V. Goldberg, editors, *Proc. of the 2nd International Workshop on Algorithm Engineering and Experiments (ALENEX'00)*, 2000.

[17] 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.

[18] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proc. 40th IEEE Symposium on Foundations of Computer Science (FOCS'99)*, pages 81–99, 1999.

[19] V. King and M. Thorup. A space saving trick for directed dynamic transitive closure and shortest path algorithms. In *Proceedings of the 7th Annual International Computing and Combinatorics Conference (COCOON), LNCS 2108*, pages 268–277, 2001.

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

[21] 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, London, UK, 1967.

[22] 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.

[23] 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.

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

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

[26] V. Rodionov. The parametric problem of shortest distances. *U.S.S.R. Computational Math. and Math. Phys.*, 8(5):336–343, 1968.

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

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