Efficient on-line topological simplification of network-like data

Floris Geerts Peter Revesz* Jan Van den Bussche

Limburgs Universitair Centrum

Abstract

We describe an efficient on-line algorithm to simplify network-like data with the goal of speeding up path queries on these data. Our algorithm is on-line in that it reacts to updates on the data, keeping the simplification up-to-date. The supported updates are insertions of vertices and edges; hence, our algorithm is *semi-dynamic*. We provide both analytical and empirical evaluations of the efficiency of our approach. Specifically, we prove an $O(\log m)$ upper bound on the amortized time complexity of our maintenance algorithm, with m the number of edges. We also show that the overhead due to maintenance is negligible using real data, and illustrate the improved performance on shortest path queries over the same data.

1 Introduction

Many GIS applications involve data in the form of a network, such as road, railway, or river networks. "Path queries," such as shortest paths or transitive closure, are very important in this context. In this paper we will discuss a simpli-

*Work done while on a sabbatical leave from the University of Nebraska-Lincoln. Work supported in part by USA NSF grants IRI-9625055 and IRI-9632871.

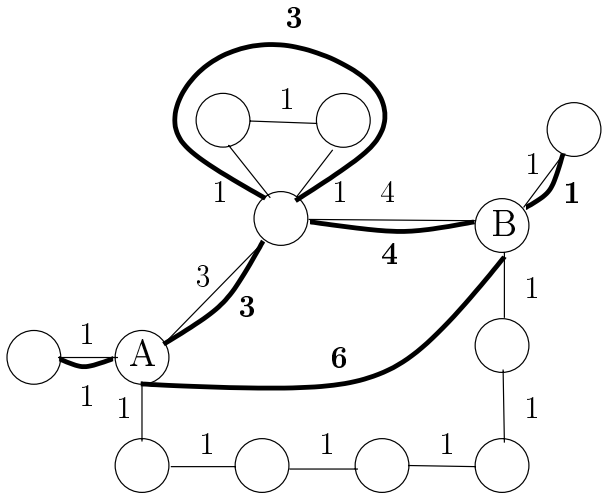


Figure 1: A graph and its simplification (thick edges)

fication method for speeding up these queries on network-like data, represented as undirected graphs with weights on the edges.

The simplification method consists of eliminating all "regular" vertices. These are the vertices that simply lie on a line; in graph-theoretic terms, they are the vertices of degree 2. For example, in Figure 1, a graph together with its simplification are given. It is clear that the shortest path from vertex A to vertex B can be found more quickly in the simplification than in the original graph. Regular vertices occur often abundantly. For example, in a road network database, each bend in the road is represented by a vertex, which will be regular. The same applies more generally to all networks represented on top of a discrete raster [9], where a curved line is approximated by many straight line segments between raster points, which then will be regular.

From a purely topological point of view, the simplification of a graph contains in a compact manner the same information as the original graph. Such "lossless topological representations" (also called "topological invariants") have recently been studied by a number of researchers [3, 6, 7]. For example, initial experiments reported on by Segoufin and Vianu have shown drastic compression

of the size of the data by topological simplification.

Of course, if we want to answer path queries using the simplified graph instead of the original one, we are faced with the problem of on-line maintenance of the simplified graph under updates to the original one. Two of us have recently reported on an initial investigation of this problem [2]. The result was a maintenance algorithm which was *fully-dynamic*, i.e., insertions and deletions of edges and vertices are allowed. This algorithm, however, is (in certain "worst cases") not any better than redoing the simplification from scratch after every update. This is clearly not very practical.

The contribution of the present paper is an algorithm for on-line topological simplification which takes, on the average, only *logarithmic* time per edge insertion to keep the simplified graph up-to-date. Specifically, suppose that the graph representing the network consists of n vertices and m edges. We prove an $O(\log m)$ bound on the "amortized" time complexity of our maintenance algorithm.

Our algorithm does not make any assumptions on the graph, such as planarity and the like. Real-life network data are often *not* planar (e.g., in a road or railway network, bridges occur). Our algorithm is only *semi-dynamic*, in that it can react efficiently to insertions (of vertices and edges), but not to deletions. Whether there exists an algorithm for on-line topological simplification that reacts to deletions as well, and that has an amortized time complexity better than $O(n)$, remains under investigation

We have also performed an empirical validation of our results, using two real life datasets. We maintained the simplification of a railway network of the USA, and the simplification of a hydrographic map of Nebraska. The experiments show that the overhead due to the maintenance of the simplified graph is almost negligible. On both datasets we also conducted a suite of shortest path queries to illustrate the speedups one obtains by querying the simplification instead of the original graph.

This paper is further organized as follows. Basic definitions are given in Section 2. The maintenance algorithm is described in Section 3. Its analytical

performance is described in Section 4, and its empirical performance is described in Section 5.

2 Basic definitions

Consider an undirected graph with weighted edges $G = (V, E, \lambda)$, where $\lambda : E \to \mathbf{R}^+$ is the weight function. We will use the following definitions:

1. A vertex is called *regular* if it is incident on precisely 2 edges.
2. A vertex that is not regular is called *singular*.
3. A path between two singular vertices that passes only through regular vertices is called a *regular path*. The *size* of a regular path is the number of regular vertices on it.

The *simplification* of G is the multigraph[1] $G_s = (V_s, E_s, \lambda_s)$ obtained from G as follows:

1. V_s is the subset of V consisting of all singular vertices.
2. E_s consists of all pairs $\{x, y\}$ such that there is a regular path between x and y in G. Such a pair $\{x, y\}$ may occur multiple times, since there may well be multiple regular paths between x and y in G. If x and y are singular and adjacent in G, then the edge $e = \{x, y\}$ is viewed as a regular path of size 0.

The edges in G_s are called the *topological edges* of G.

3. $\lambda_s(\{x, y\})$ equals the sum of all $\lambda(e)$, where e is an edge on the particular regular path in G, corresponding to the topological edge $\{x, y\}$.

The *degree* of a vertex $x \in V$ is the number of edges in G (or, for a singular vertex, equivalently in G_s) incident with x. Recall that, by definition, the degree of a regular vertex is two. We denote the degree of x by $\deg(x)$.

[1] A multigraph is a graph where multiple edges between two vertices are allowed.

3 On-line simplification

3.1 General description

Updates on the graph G must be translated into updates on its simplification G_s.

We only consider insertions of a new isolated vertex and insertions of edges between existing vertices in the graph G (other more complex insertion operations can be translated into a sequence of these basic insertion operations). The insertion of an isolated vertex is handled trivially: we merely have to insert it in V_s.

We distinguish among six cases that are explained below and depicted in Figures 2–7. The left side of each figure shows the situation before the insertion of the edge $\{x, y\}$, which is the dotted line. The right side of each figure shows the situation after the insertion. As before, the topological edges are drawn in thick lines.

Case 1. Vertices x and y are both singular and $\deg(x) \neq 1$ and $\deg(y) \neq 1$.

Then the edge $\{x, y\}$ is also *inserted* in G_s.

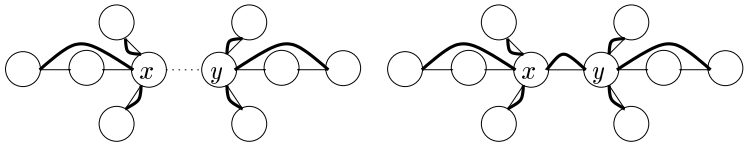


Figure 2: Case 1

Case 2. Vertices x and y are both singular and one of them, say x, has degree one.

Let $\{z, x\}$ be the edge in G_s incident with x. *Extend* this edge to the new edge $\{z, y\}$ in G_s, putting $\lambda_s(\{z, y\}) := \lambda_s(\{z, x\}) + \lambda(\{x, y\})$. Note that x becomes a regular vertex after the insertion.

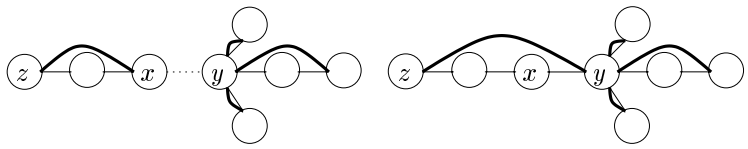


Figure 3: Case 2

Case 3. Vertices x and y are both singular and $\deg(x) = \deg(y) = 1$.

Let $\{z_1, x\}$ (resp. $\{z_2, y\}$) be the edge in G_s adjacent with x (resp. y). Suppose $z_1 \neq y$ (and hence $z_2 \neq x$). Then *merge* the edges $\{z_1, x\}$ and $\{y, z_2\}$ in G_s into a single, new edge $\{z_1, z_2\}$ in G_s, putting $\lambda_s(\{z_1, z_2\}) := \lambda_s(\{z_1, x\}) + \lambda_s(\{y, z_2\}) + \lambda(\{x, y\})$. If $z_1 = y$ (and hence $z_2 = x$) then *extend* the edge $\{x, y\}$ to, a new edge $\{x, x\}$ in G_s, putting $\lambda_s(\{x, x\}) := \lambda_s(\{x, y\}) + \lambda(\{x, y\})$.

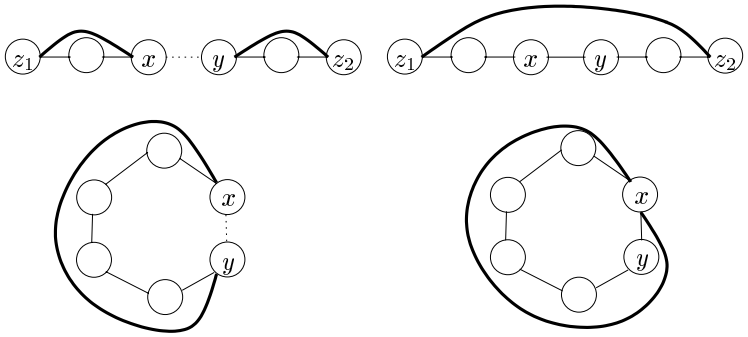


Figure 4: Case 3

Case 4. One of the vertices x and y is regular, say x, and the other vertex, say y, is singular and has degree one.

Two things must be done. First, the edge $\{z_1, z_2\}$ of G_s which corresponds to the regular path between z_1 and z_2 on which x lies, must be *split* into two[2] new edges $\{z_1, x\}$ and $\{x, z_2\}$ of G_s. Here, we put $\lambda_s(\{z_1, x\}) := \sum \lambda(\{u, v\})$, where the summation is over all edges in G on the regular path from z_1 to x. We similarly define $\lambda_s(\{x, z_2\})$. Secondly,

[2]A special case has to be considered when $\deg(z_1) = \deg(z_2) = 2$. In this case, x lies on a regular self–loop with begin– and endpoint $z_1 = z_2$. By splitting this loop, x becomes the new begin and endpoint of the self-loop. We omit this case for simplicity of exposition.

let $\{z_3, y\}$ be the edge in G_s incident with y. Then we *extend* this edge to a new edge $\{z_3, x\}$ in G_s, putting, $\lambda_s(\{x, z_3\}) := \lambda_s(\{y, z_3\}) + \lambda(\{x, y\})$.

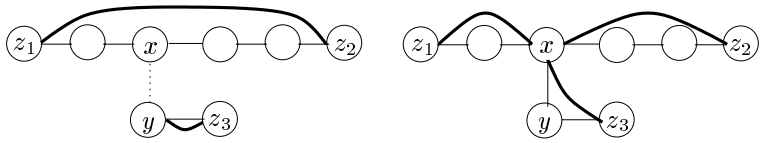


Figure 5: Case 4

Case 5. One of the vertices, say x is regular and the other one, say y is singular with degree other than one.

Then we split exactly as in case 4, and now we also insert $\{x, y\}$ as a new edge in G_s.

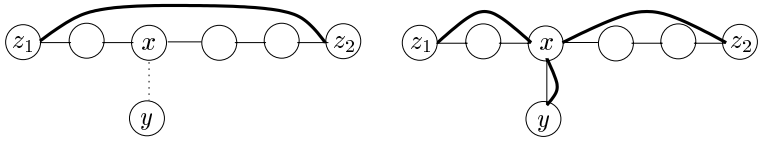


Figure 6: Case 5

Case 6. Both x and y are regular: Then two *splits* must be performed

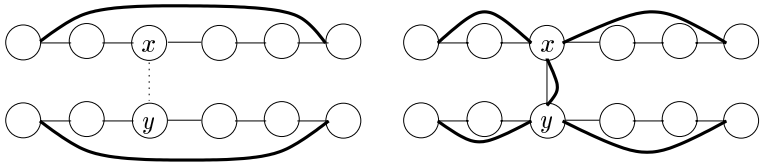


Figure 7: Case 6

As can be seen in the description of cases 1–6, if no regular vertices are involved, then the update on the graph G translates trivially in exactly the same update on the simplification G_s. It is only in cases 4, 5, and 6, that the update on the graph G involves vertices which *have no counterpart* in the simplification G_s. However, we need to know which edge we need to split and what the weights are of the topological edges created by the split. Consequently, the problem of maintaining the simplification G_s of a graph G amounts to two

tasks:

- Maintain a function *find topological edge*, which takes as input a regular vertex, and outputs the topological edge whose regular path in G contains the input vertex.
- Maintain a function *find weights* which outputs the weights of the edges created when a topological edge is split at the input vertex.

In an earlier, naive approach [2], we only discussed the function *find topological edge*. It worked by storing for each regular vertex a pointer to its topological edge. This made the topological edge accessible in constant time, but the maintenance of the pointers under updates can be very inefficient in the worst case. We next describe our new, efficient approach.

3.2 Finding topological edges

Assigning numbers to the regular vertices We *number* the regular vertices consecutively that lie on a regular path. The numbers of the regular vertices on any regular path will always form an interval of the natural numbers. Our algorithm will maintain two properties:

Interval property: the assignment of *consecutive* numbers to *consecutive* regular points;

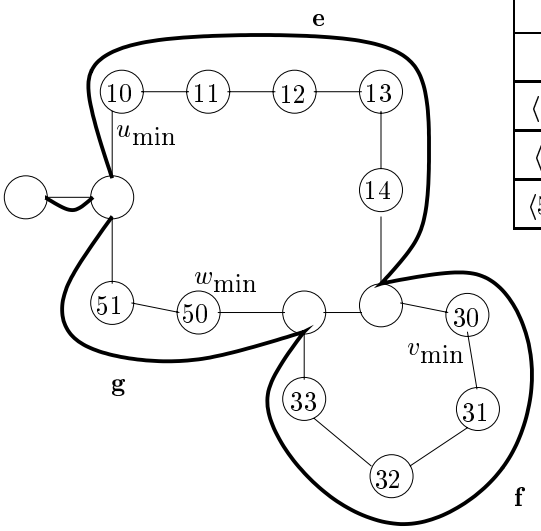
Disjointness property: *different* regular paths have *disjoint* intervals.

We have then a unique interval associated with each regular path, and hence with each topological edge of size > 0. Moreover, we choose the minimum of such an interval as a unique number associated with a topological edge. Specifically, the minimal number serves as a *key* in a dictionary. Recall that in general, a dictionary consists of pairs $\langle \text{key}, \text{item} \rangle$, where the item is unique for each key. Given a number k, the function which returns the item with maximal key smaller than k can be implemented in $O(\log N)$ time, where N is the number of items in the dictionary [1].

We use *items* which contain the following information:

1. An identifier of the topological edge associated with the key.
2. The number of regular vertices on the regular path corresponding to this topological edge.
3. An identifier of the regular vertex on the regular path corresponding to this topological edge, which has the key as number.

In Figure 8 we give an example of a dictionary containing three keys, corresponding to the three topological edges in the simplification G_s of the graph G.



dictionary			
⟨key,	item⟩		
⟨10,	**e**	5	$u_{\min}$⟩
⟨30,	**f**	4	$v_{\min}$⟩
⟨50,	**g**	2	$w_{\min}$⟩

Figure 8: Dictionary example.

Finding the topological edge Consider that we are in one of the cases 4–6 described in Section 3.1, where we have to split a topological edge. We look at the number of x, say k, and find in the dictionary the item associated with the maximal key smaller than k. This key corresponds to the interval to which k belongs, or equivalently, to the regular path to which x belongs. In this way we

find the topological edge which has to be split, since this edge is identified in the returned item.

The numbering thus enables us to find an edge in $O(\log m')$ time, where m' is the number of edges in G_s. Because m' is at most m, the number of edges in G, we obtain:

Proposition 1 *Given a regular vertex and its number, the dictionary returns in $O(\log m)$ time the topological edge corresponding to the regular path on which this regular vertex lies.*

Maintaining the numbers of the regular vertices We must now show how to maintain this numbering under updates, such that the interval and disjointness properties mentioned above remain satisfied.

Actually, only in case 3 we need to do some maintenance work on the numbering. Indeed, by merging two topological edges, the numbering of the regular vertices is no longer necessarily consecutive. We resolve this by *renumbering* the vertices on the shortest of the two regular path. Note that the size of a regular path is stored in the dictionary item for that path.

In order to keep the intervals disjoint, we must assume that the maximal number of edge insertions to which we need to respond is known in advance. Concretely, let us assume that we have to react to at most ℓ update operations.[3] A regular path is "born" with at most two regular vertices on it. Every time a new regular path is created, say the kth time, we assign to one of the two regular vertices on it the number $2k\ell$. Hence, newly created topological edges correspond to numbers which are 2ℓ apart from each other. Since a newly created topological edge can become at most $\ell-1$ vertices longer, no interference is possible.

[3]This assumption is rather harmless: one can set this maximum limit to a large number. If it is eventually reached, we simplify once from scratch and are back to zero.

3.3 Finding weights

We next show how, when a topological edge is split, we can quickly find the weights of the two new edges created by the split.

Assigning weights to the regular vertices We assign a *weight* to each regular vertex.[4] Suppose we have a newly created topological edge $\{x, y\}$. If the corresponding regular path has only one regular vertex, we do nothing. If the regular path has two regular vertices, say u_0 and u_1, let u_0 be the vertex with minimal number. Then we define:

- The weight of u_0 equals the weight of the edge $\{x, u_0\}$ or $\{u_0, y\}$ in G, depending on whether u_0 is adjacent to x or y.
- The weight of u_1 equals the weight of u_0 plus the weight of the edge $\{u_0, u_1\}$ in G.

In order to define the weight of regular vertices on regular paths longer than two, let us define the *kth vertex* of a regular path as the vertex with number s, such that $s - s_{\min} = k$, where $s_{\min}$ is the minimal number of the vertices on the regular path. Hence the vertex with the minimal number has position 0. We now define:

- The weight of the kth regular vertex u_k is the sum of the weight of vertex at position $k-1$, call it u_{k-1}, plus the weight of edge $\{u_k, u_{k-1}\}$ in G.

Maintaining these weights The maintenance of these weights of regular vertices, under edge insertions, is easy. It requires only constant time when a topological edge is extended, and no time at all when a topological edge is split. However, when two topological edges are merged, we need to adjust the weights of the regular vertices on the shortest of the two regular paths, as shown in Figure 9. This adjustment of the weights can clearly be done simultaneously with the renumbering of the vertices, as explained in Section 3.2 on finding topological edges.

[4] The weight of a singular vertex is zero.

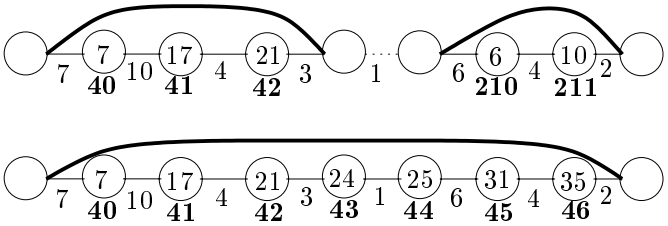


Figure 9: Assigning new numbers and weights of regular vertices simultaneously when two topological edges are merged. The numbers of regular vertices are in bold, the weights are inside the vertices.

Finding the weights The weights of regular vertices, now enable us to find the weights of the two edges created by a split of a topological edge in logarithmic time. Indeed, given the number of the regular vertex where the split occurs, we lookup in the dictionary which topological edge needs to be split; call it $\{z_1, z_2\}$. In the returned item we find the vertex which has the minimal number of the vertices on the regular path corresponding to $\{z_1, z_2\}$. Denote this vertex with u; it is adjacent to either z_1 or z_2. Now the weights of the two new topological edges $\{z_1, x\}$ and $\{x, z_2\}$ can be computed easily:

- The weight of $\{z_1, x\}$ is the weight of vertex x if u is adjacent to z_1; or the weight of $\{z_1, z_2\}$ minus the weight of vertex x if u is adjacent to z_2.
- The weight of $\{x, z_2\}$ equals the weight of $\{z_1, z_2\}$ minus the weight $\{z_1, x\}$.

If only one regular vertex remains on a regular path after a split, or a regular vertex becomes singular, then the weight of this vertex is set to 0. This all takes constant time plus the time for one lookup in the dictionary, which takes logarithmic time. Hence, with m the number of edges of G, we obtain:

Proposition 2 *The weights of the two new edges created by a split can be computed in $O(\log m)$ time.*

4 Complexity analysis

By the *amortized complexity* of an on-line algorithm [5, 8], we mean the total computational complexity of supporting ℓ updates (starting from the empty graph), as a function of ℓ, divided by ℓ to get the average time spent on supporting one single update. We only count edge insertions because the insertion of an isolated vertex is trivial. We will prove here that our algorithm has an $O(\log \ell)$ amortized time complexity.

Theorem 1 *The total time spent on ℓ updates by our maintenance algorithm is $O(\ell \log \ell)$.*

Proof. If we look at the general description of the algorithm in the previous section, we see that in each case only a constant number of steps are performed either elementary operations on the graph, or dictionary lookups. There is however one important exception to this. In cases where we need to merge two topological edges, renumbering of regular vertices (and simultaneously adjustment of their weights) is involved. Since every elementary operation on the graph takes only constant time, and every dictionary lookup takes only $O(\log \ell)$ time, all we have to prove is that the total number of renumberings is $O(\ell \log \ell)$.

A key concept in our proof is the notion of a *super edge* (see Figure 10). Super edges are sets of topological edges which can be defined inductively:

Initially each topological edge (with one or two regular vertices on it) is a member of a separate super edge. If a member **a** of a super edge **A** is merged with a member **b** of another super edge **B**, then the two super edges are unioned together to get a new super edge **C** and **a** and **b** are merged into a new member **c** of the new super edge **C**. If a member **d** of a super edge is split into **e** and **f**, then both **e** and **f** will belong to the same super edge as **d** did. The important property of super edges is that the total number of vertices can only grow. We call this number the *size* of a super edge. Each split operation does not affect the size of super edges, while each merge operation can only increase it.

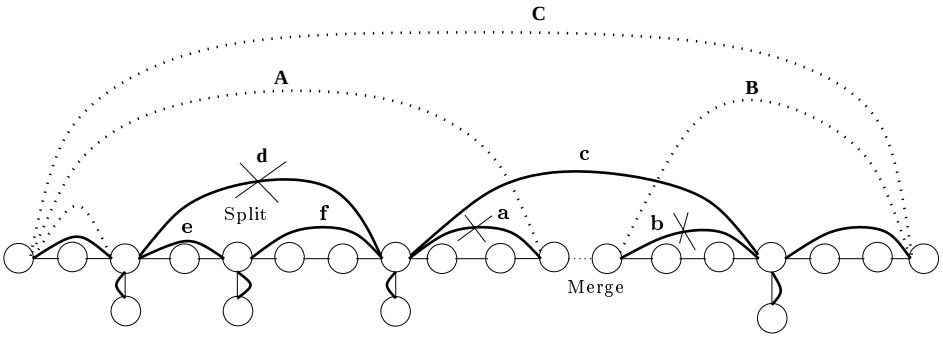


Figure 10: An example of some super edges (dotted lines). Details can be found in the text.

We now prove by induction on ℓ that the total number of renumberings done in a super edge of size m is $O(m \log m)$.

The statement is trivial for $\ell = 0$, so we take $\ell > 0$. We may assume that the ℓth update involves a merge of two topological edges, since this is the only update for which we have to do renumbering. Suppose that the sizes of the two super edges being unioned together are m_1 and m_2. Without loss of generality assume that $m_1 \leq m_2$. So, according to our algorithm which renumbers the shortest of the two, we have to do m_1 renumbering steps: m_1 to assign new numbers, and m_1 to assign new weights. The size of the new super edge will be $m = m_1 + m_2$. By the induction hypothesis, the total numbers of renumberings already done while building the two given super edges are $m_1 \log m_1$ and $m_2 \log m_2$. Therefore, the total number of renumberings is bounded by

$$\begin{aligned} m_1 \log m_1 + m_2 \log m_2 + m_1 &= m_1(1 + \log m_1) + m_2 \log m_2 \\ &= m_1 \log(2m_1) + m_2 \log m_2 \\ &\leq m_1 \log(m_1 + m_2) + m_2 \log(m_1 + m_2) \\ &= m \log m. \end{aligned}$$

We can now finish the proof as follows. After ℓ updates, a number of super edges have been formed, say k, of sizes $m_1, \ldots, m_k$. We have just proved that for each

of these, we performed $m_i \log m_i$ renumberings in total. Hence, the total number of renumberings performed is $\sum_i m_i \log m_i \leq (\sum_i m_i) \log (\sum_i m_i) \leq \ell \log \ell$. □

To conclude this section, we recall from the previous section that the maximal number assigned to a regular vertex is $2\ell^2$. So, all numbers involved in the algorithm take only $O(\log \ell)$ bits in memory.[5]

5 Experimental results

Maintenance of the simplification In order to verify the theoretical running time of our algorithm, we performed some modest experiments with an implementation written in C++ on top of LEDA [4], run on a 266 MHz Pentium II machine with 64 MB RAM.

We present the results on two data sets.

Hydrography graph: The data set represents the hydrography of Nebraska and contains 101 336 edges on 157 972 vertices, of which 96 636 are regular.

Railroad graph: The data set represents a network database of all railway mainlines, railroad yards, and major sidings in the continental U.S. compiled at a scale of 1 : 100 000. It contains 164 380 edges on 133 752 vertices, of which only 14 261 are regular. It is available at the U.S. Bureau of Transportation Statistics (BTS, `http://www.bts.gov/gis`.)

We performed the test as follows. We gradually "grow" the graph, starting from a random vertex, by adding vertices and edges that are adjacent to vertices that we already have. Whenever a connected component is completed, we jump to a new random vertex from another component and continue. At the same time, we maintain the topological simplification.

Figures 11 and 12 show, for one particular run of the experiment, the time elapsed between blocks of 5 000 inserted edges for the hydrography graph, and 10 000 for the railroad graph. Tables 1 and 2 show more details of the run.

[5] Technically, Theorem 1 assumes the standard RAM computation model with unit costs. If logarithmic costs are desired, the complexity is $O(\ell \log^2 \ell)$.

There, m_o is the number of edges in the original graph; m_s is the number of edges in the simplification; t is the elapsed time expressed in seconds; n_o is the number of vertices in the original graph; and n_r is the number of regular vertices. Hence, the number of vertices in the simplified graph is $n_o - n_r$.

These results show an apparently linear, and thus optimal, behavior, thus confirming our theoretical $O(\ell \log \ell)$ time bound. In particular, the logarithmic factor clearly behaves in our experiments as if it were a constant.

Shortest path queries on the simplified graph vs. the original graph We performed on both data sets the following shortest path query: Select at random pairs of vertices and compute a shortest path between them. The shortest path query is posed on both the original graph and it simplification. We use a bidirectional Dijkstra algorithm (which is part of LEDA). This algorithm runs in $O(m + n \log n)$ time. If the shortest path between regular vertices needs to be computed, we lookup the topological edges on which these vertices lie, and split these edges *temporarily* until the shortest path is computed. After that, we restore the simplification in its original form.

We selected a large connected connected component in both datasets and did 1 000 path queries. The results of this experiment are shown in Figures 13 and 14. Quite expectedly, the speedup achieved by working with the simplification is clearly visible. For the railroad data set, the number of regular vertices is much lower and thus the speedup is smaller.

6 Concluding remarks

Especially when a large number of path queries have to be performed, it is advantageous to use the simplification. Indeed, let t_s be the extra time needed to maintain the simplification. Let p_o be the average time for a path query on the original graph, and let p_s be the average time for a path query on the simplification. Then maintaining the simplification becomes worthwhile as soon as we ask at least

$$N = \frac{t_s}{p_o - p_s}$$

path queries.

For the hydrography data set, this number N is around 40, when the number of edges of the graph G is 10 000. For the railroad data set the number N is around 1 000, when the number of edges of the graph G is 100 000.

Acknowledgment We would like to thank Bart Goethals for programming help, and Bill Waltman for providing us with the hydrography data set.

References

[1] T.H. Cormen, C.E. Leierson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[2] F. Geerts, B. Kuijpers, and J. Van den Bussche. Topological canonization of planar spatial data and its incremental maintenance. In T. Polle and T. Ripke, editors, *Fundamentals of Information Systems*, volume 496 of *The Kluwer International Series in Engineering and Computer Science*, pages 55–67. Kluwer, 1999.

[3] B. Kuijpers, J. Paredaens, and J. Van den Bussche. Lossless representation of topological spatial data. In M.J. Egenhofer and J.R. Herring, editors, *Advances in Spatial Databases*, volume 951 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 1995.

[4] Library of Efficient Data Types and Algorithms. A research version is freely avialable at
`http://www.mpi-sb.mpg.de/LEDA/download/research.html`.

[5] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching.* EACTS Monographs on Theoretical Computer Science. Springer-Verlag, 1984.

[6] C.H. Papadimitriou, D. Suciu, and V. Vianu. Topological queries in spatial databases. In *Proceedings of the 15th ACM Symposium on Principles of Database Systems*, pages 81–92. ACM Press, 1996.

[7] Luc Segoufin and Victor Vianu. Querying spatial databases via topological invariants. In *Proceedings of the 17th ACM Symposium on Principles of Database Systems*, pages 89–98. ACM Press, 1998.

[8] R.E. Tarjan. Data structures and network algorithms. In *CBMS-NSF Regional Conference Series in Applied Mathematics*, volume 44. SIAM, 1983.

[9] M.F. Worboys. *GIS: A Computing Perspective*. Taylor and Francis, 1995.

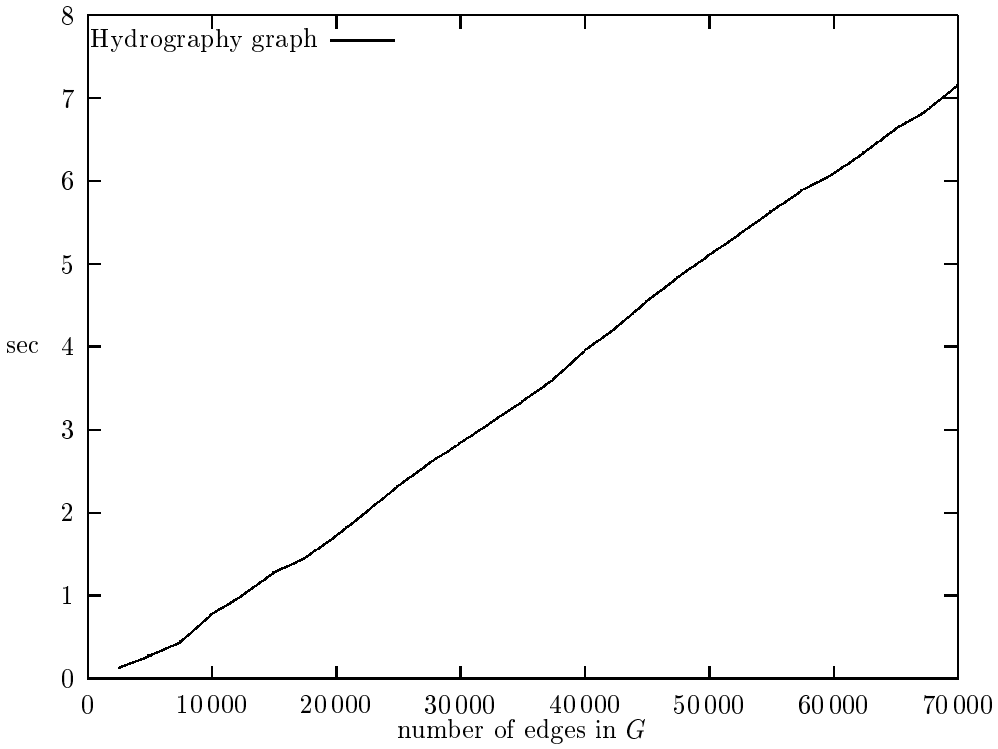


Figure 11: Times to maintain simplification of hydrography graph.

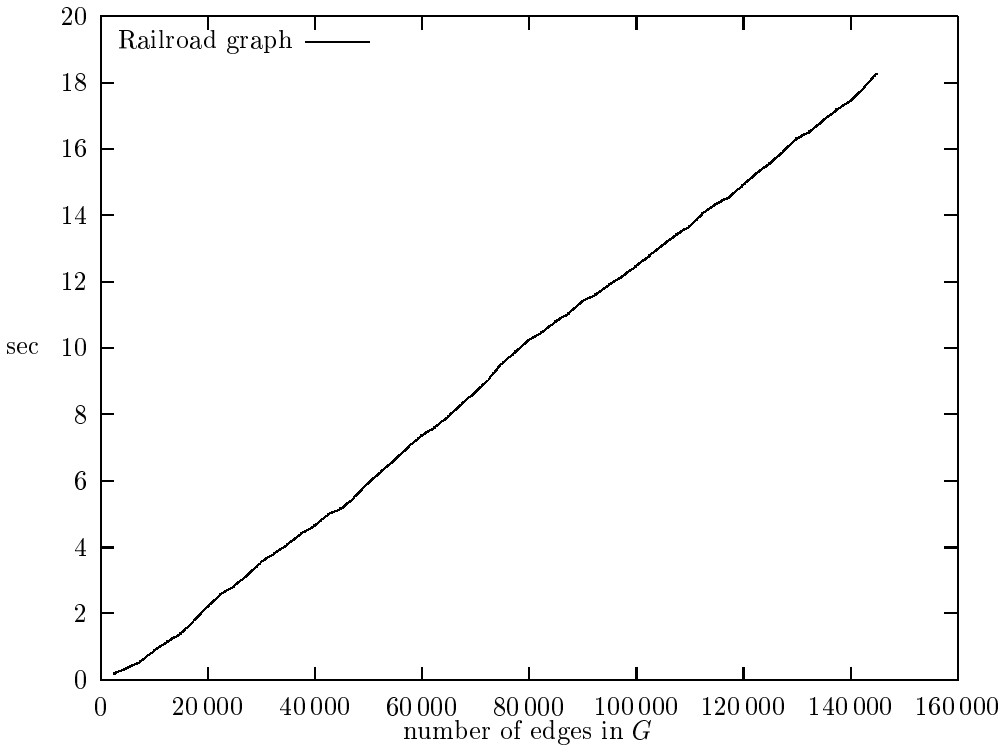


Figure 12: Times to maintain simplification of railroad graph.

m_o	t	m_s	n_o	n_r
5 000	0.28	128	4 998	4 872
10 000	0.78	914	9 763	9 086
15 000	1.28	1 691	14 522	13 309
20 000	1.72	1 771	19 522	18 229
25 000	2.32	1 877	24 514	23 123
30 000	2.84	2 994	29 154	27 006
35 000	3.34	3 367	34 065	31 633
40 000	3.96	3 478	39 058	36 522
45 000	4.55	3 567	44 057	41 433
50 000	5.11	3 661	49 063	46 339
55 000	5.63	3 921	54 007	51 079
60 000	6.09	4 168	58 969	55 832
65 000	6.63	4 249	63 972	60 751
70 000	7.15	4 348	68 997	65 652

Table 1: Maintenance results for the hydrography graph.

m_o	t	m_s	n_o	n_r
10 000	0.89	7 068	8 720	2 932
20 000	2.2	14 376	17 443	5 624
30 000	3.54	22 402	25 979	7 598
40 000	4.64	30 791	34 280	9 209
50 000	5.92	39 345	42 464	10 655
60 000	7.37	47 796	50 683	12 204
70 000	8.67	55 949	59 015	14 051
80 000	10.24	63 994	67 533	16 006
90 000	11.42	72 170	75 857	17 830
100 000	12.48	80 117	84 310	19 883
110 000	13.67	87 881	92 962	22 119
120 000	14.9	95 713	101 552	24 287
130 000	16.32	103 938	110 204	26 062
140 000	17.45	112 021	118 816	27 979
145 000	18.26	115 724	123 189	29 276

Table 2: Maintenance results for the railroad graph.

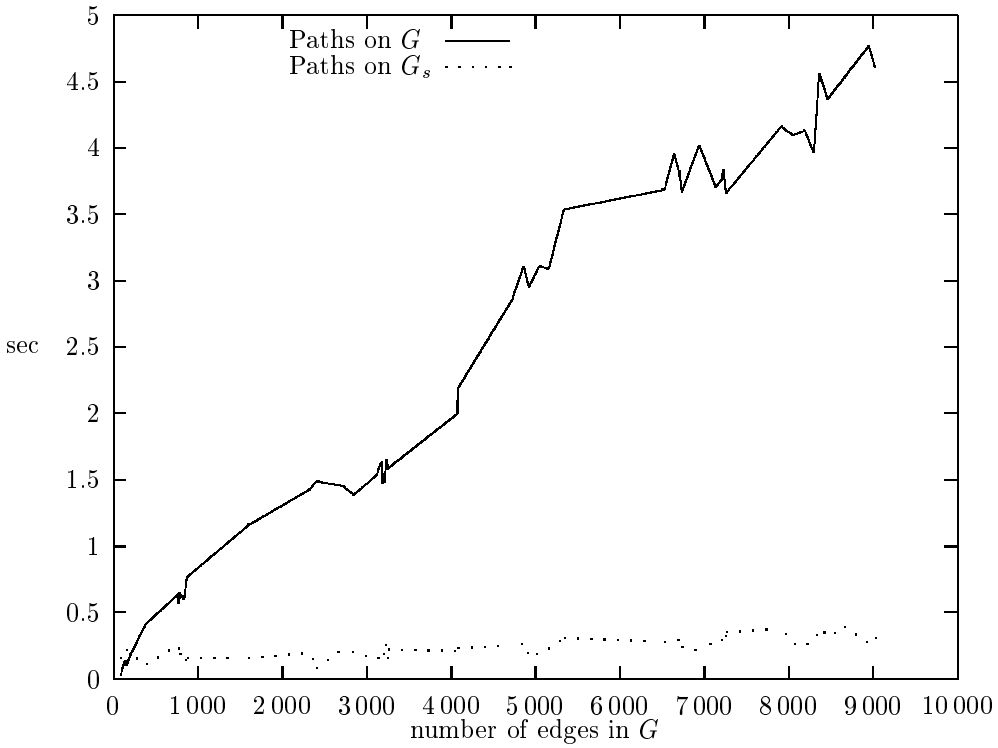


Figure 13: Shortest path queries on the largest connected component of the hydrography graph, and on its simplification.

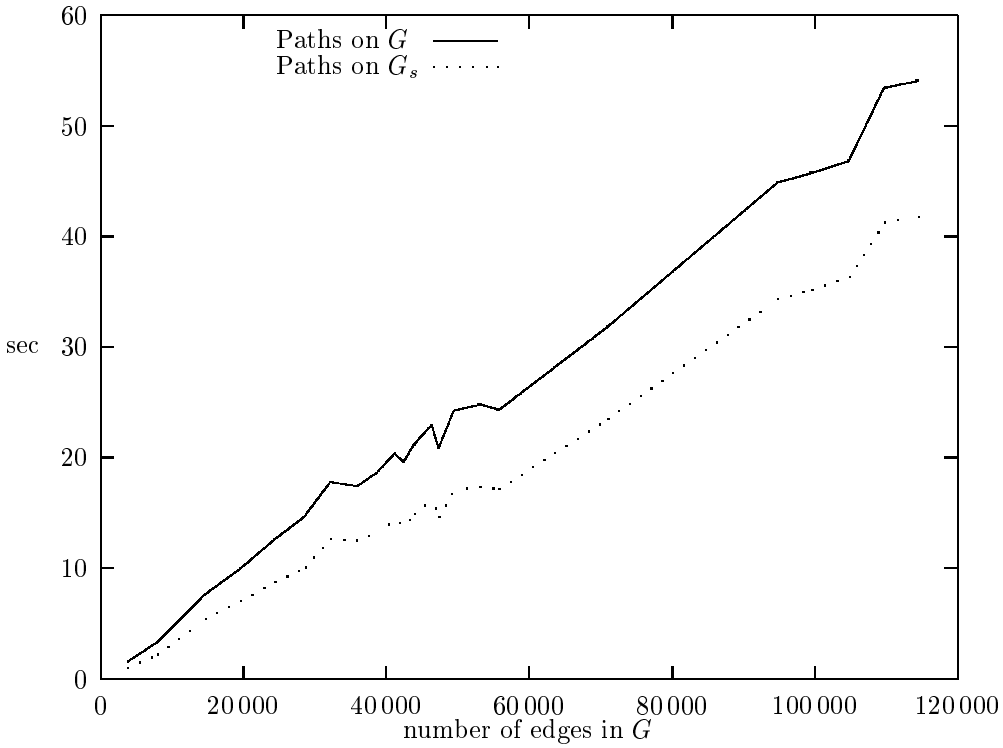


Figure 14: Shortest path queries on the largest connected component of the railroad graph, and on its simplification.