Maintenance of Multi-level Overlay Graphs for Timetable Queries*

Francesco Bruera, Serafino Cicerone, Gianlorenzo D'Angelo,
Gabriele Di Stefano and Daniele Frigioni

Dipartimento di Ingegneria Elettrica e dell'Informazione,
Università degli Studi dell'Aquila, I-67040 Monteluco di Roio, L'Aquila - Italy.
E-mail: francesco.bruera@gmail.com;
{cicerone, gdangelo, gabriele, frigioni}@ing.univaq.it

Abstract. In railways systems the timetable is typically represented as a weighted digraph on which itinerary queries are answered by shortest path algorithms, usually running Dijkstra's algorithm. Due to the continuously growing size of real-world graphs, there is a constant need for faster algorithms and many techniques have been devised to heuristically speed up Dijkstra's algorithm. One of these techniques is the *multi-level overlay graph*, that has been recently introduced and shown to be experimentally efficient, especially when applied to timetable information. In many practical application major disruptions to the normal operation cannot be completely avoided because of the complexity of the underlying systems. Timetable information update after disruptions is considered one of the weakest points in current railway systems. This determines the need for an effective online redesign and update of the shortest paths information as a consequence of disruptions. In this paper, we make a step forward toward this direction by showing some theoretical properties of multi-level overlay graphs that lead us to the definition of a new data structure for the dynamic maintenance of a multi-level overlay graph of a given graph G while *weight decrease* or *weight increase* operations are performed on G. Our solution is theoretically faster than the recomputation from scratch and allows fast queries.

Keywords. Timetable Queries, Speed-up techniques for shortest paths

1 Introduction

The computation of shortest paths is a central requirement for many applications, such as route planning or search in huge networks. In a railways system, timetables are typically represented as weighted directed graphs and itinerary queries are answered by shortest path algorithms, usually running Dijkstra's algorithm. Due to the continuously growing size of real-world graphs, there is a

* This work was partially supported by the Future and Emerging Technologies Unit of EC (IST priority - 6th FP), under contract no. FP6-021235-2 (project ARRIVAL).

ATMOS 2007 (p.226-242)
7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems
http://drops.dagstuhl.de/opus/volltexte/2007/1171

constant need for faster algorithms and in the course of the years a large number of techniques have been devised to heuristically speed up Dijkstra's algorithm.

In most of the above mentioned practical application major disruptions to the normal operation cannot be avoided because of the complexity of the underlying systems. This determines the need for an effective online redesign and update of the shortest paths information as a consequence of these disruptions. Timetable information update after disruptions is considered one of the weakest points in current railway systems, and it has received little attention in the scientific literature. Hence, there is a constant need of dynamic algorithms that are faster than the recomputation from scratch of shortest paths, especially when applied to huge graphs as those resulted from many practical applications.

Previous works There are numerous approaches to speed-up single-pair shortest path computations when the graph is static [1–10]. On the one hand, there are speed-up techniques that are based on pruning strategies of the search space of Dijkstra's algorithm (see, e.g., [3, 6, 8]). On the other hand, there are speed-up techniques that require to preprocess the graph at an off-line step so that subsequent on-line queries take only a fraction of the time used by Dijkstra's algorithm. The known preprocessing techniques are based on different approaches: geometric information [10], hierarchical decomposition [1, 4, 9, 11–13], landmark distances [2, 3], and arc-labelling [14]. For a survey of speed-up techniques for shortest paths computation see [15].

Despite the great job done in the last years in this area, very few solutions have been proposed that are suitable to be used in a dynamic environment, where modifications can happen to the underlying graph and preprocessed information on shortest paths have to be recomputed. Up to now only dynamic approaches based on geometric information and landmark distances are known as that in [16, 17]. Unfortunately, the known theoretical approaches for dealing with dynamic shortest path problems are based on a matrix representation of shortest path information, whose size is at least quadratic (see, e.g., [18]) to the number of nodes of the graph. For instance, for graphs representing timetable information, with typically millions of nodes and edges, such an approach cannot be applied.

Results of the paper One of the speed-up techniques for shortest paths requiring preprocessing is known as *multi-level overlay graph* and it has been introduced in [4]. Given a weighted directed graph G and a sequence $S_1, S_2, \ldots, S_l$ of subsets of V such that $V \supset S_1 \supset S_2 \supset \ldots \supset S_l$, a *multi-level overlay graph* is defined as $\mathcal{M}(G; S_1, ..., S_l) = (V, E \cup E_1 \cup E_2 \cup \ldots \cup E_l)$, where E_i, $1 \leq i \leq l$, is a set containing the so called *i-level edges*, which are additional edges determined by the nodes in S_i that represent pre-computed shortest paths in G. When a s-t distance query is asked, this hierarchical decomposition allows to build a graph $\mathcal{M}_{st}(V_{st}, E_{st})$ whose size is much smaller than the size of the original graph G, and such that the distance from s to t is the same in $\mathcal{M}_{st}$ and in G. Thus, an s-t distance query can be answered faster in $\mathcal{M}_{st}$ than in G.

In [4], multi-level overlay graphs have been shown to be experimentally efficient when applied to timetable information, as it has been done with other

multi-level approaches (see, e.g., [9]). In [19] a dynamic approach has been proposed to update a variation of the multi-level overlay graphs. Experiments on the Western European road network, show that this technique is potentially suitable for practical application. However, there is no theoretical and experimental study about the efficient dynamic maintenance of this data structure after disruptions.

In this paper, we make a first step forward toward this direction by proposing a theoretical study that leads us to the definition of a new data structure for the dynamization of a multi-level overlay graph, while *weight decrease* or *weight increase* operations are performed on the original graph. In particular, let be given a multi-level overlay graph $\mathcal{M}(G; S_1, ..., S_l)$ of a given weighted directed graph $G = (V, E)$, with n nodes and m edges. We show theoretical properties of $\mathcal{M}(G; S_1, ..., S_l)$ that allow us to: (i) store the information on $\mathcal{M}$ in a data structure requiring $O(n + m + |\bigcup_{i=1}^{l} E_i|)$ optimal space; (ii) compute $\mathcal{M}$ in $O(|S_1|(m + n \log n))$ worst case time; (iii) answer s-t distance queries as in [4], in $O(m + |S_1|^2 + |V_{st}| \log |V_{st}|)$ worst case time, $|V_{st}| < n$; (iv) dynamize the newly introduced data structure with the additional storage of $|S_1|$ shortest paths trees. In fact, we show that, if a *modification* (either a *weight decrease* or a *weight increase* operation on an edge) occurs on G, to update $\mathcal{M}(G; S_1, ..., S_l)$, it is sufficient to update the stored $|S_1|$ shortest paths trees. We propose a dynamic algorithm that requires $O(|S_1|(m+n))$ space, $O(|S_1|(m+n) \log n)$ preprocessing time, and $O(|S_1|n+m+\Delta\sqrt{m} \log n)$ worst case time to deal with a modification, by using the fully dynamic algorithm in [20]. Here, Δ is the number of pairs in $S_1 \times V$ that change the distance as a consequence of a modification, and hence $\Delta = O(|S_1|n)$.

We show that the proposed dynamic solution is asymptotically better than the recomputation from scratch in the case of sparse graphs; while, in the case of random graphs (that are connected with high probability) and dense graphs, the dynamic algorithm is better than the recomputation from scratch when $\Delta = o(|S_1|n/\log n)$, that is a $\log n$ factor far from its maximum value. However, since the graphs representing timetables are usually huge in size, it is important to keep the space occupancy of the dynamic algorithm within the optimal space of the static algorithm. To this aim we fix $|S_1| = O(1)$, thus reducing the query time to $O(m + |V_{st}| \log |V_{st}|)$.

2 Multi-Level Overlay Graphs

Let us consider a weighted directed graph $G = (V, E, w)$, where V is a finite set of nodes, E is a finite set of edges and w is a weight function $w : E \to \mathbb{R}^+$. The number of nodes and the number of edges of G are denoted by n and m, respectively. Given a node $v \in V$, we denote as $N(v)$ the *neighbors* of v, that is the nodes in the adjacency list of v. A path in G between nodes u and v is denoted as $P = (u, \ldots, v)$. The *weight* of P is the sum of the weights of the edges in P and we denote it by *weight*(P). A *shortest path* between nodes u and v is a path from u to v with the minimum weight. The *distance* between u and

v is the weight of a shortest path from u to v and is denoted as $d(u, v)$. In the remainder of the paper, we will assume that graphs are connected.

Multi-level overlay graphs have been introduced in [4] and represent a speed-up technique to improve the computation of single-pair shortest paths. Informally, a multi-level overlay graph $\mathcal{M}$ of G is a graph obtained by adding edges to G which represent precomputed shortest paths in G. Once $\mathcal{M}$ has been computed, for each pair of nodes $s, t \in V$ it is possible to compute a subgraph $\mathcal{M}_{st}$ of $\mathcal{M}$, such that the distance from s to t in $\mathcal{M}_{st}$ is equal to the distance from s to t in G, and $\mathcal{M}_{st}$ is smaller than G. In what follows we give a brief description of multi-level overlay graphs. For more details on multi-level overlay graphs, refer to [4].

Given G and a sequence $S_0, S_1, \ldots, S_l$ of subsets of V such that $V \equiv S_0 \supset S_1 \supset S_2 \supset \ldots \supset S_l$, a *multi-level overlay graph* is defined as $\mathcal{M}(G; S_1, ..., S_l) = (V, E \cup E_1 \cup E_2 \cup \ldots \cup E_l)$, where E_i, $1 \leq i \leq l$, is a set containing the so called *i-level edges*, which are additional edges determined by shortest paths among nodes in S_i. In particular, for each $(u, v) \in S_i \times S_i$, the pair (u, v) belongs to E_i if and only if there exists a path from u to v in G and for each shortest path P from u to v in G no internal node of P belongs to S_i. The weight of a level edge (u, v) is $d(u, v)$.

In [4] the authors show that, to build level i of an overlay graph $\mathcal{M}$, $|S_i|$ single source shortest paths trees, each rooted in a node x in S_i, have to be computed on a graph G_x^i obtained from G by assigning to each edge (u, v) of G a new weight $w_x^i(u, v) = (w(u, v), t_x^i(u, v))$, where $t_x^i(u, v)$ is defined as follows:

$$t_x^i(u, v) = \begin{cases} -1 \text{ if } u \text{ belongs to } S_i \setminus \{x\} \\ 0 \quad \text{otherwise} \end{cases}$$

Then, the results of the execution of a simple variation of Dijkstra's algorithm on G_x^i are the pairs $(d(x, z), s_x^i(z))$, for each node $z \in V$. Here $d(x, z)$ is the distance from x to z in G and $s_x^i(z)$ is the sum of $t_x^i(u, v)$ for each (u, v) belonging to the computed shortest path from x to z in G_x^i. At this point, it remains only to select which pairs $(x, z) \in S_i \times S_i$ are i-level edges. This can be easily checked because (x, z) is an i-level edge if and only if $s_x^i(z) = 0$ and $d(x, z) \neq \infty$.

Graph $\mathcal{M}(G; S_1, ..., S_l)$ can be used to speed-up single-pair distance queries. Based on the source node s and the target node t, a subgraph $\mathcal{M}_{st}$ of $\mathcal{M}$ is determined; in a real world graph G, the size of $\mathcal{M}_{st}$ is smaller than that of the original graph. In [4], the authors show that the distance from s to t is the same in G and in $\mathcal{M}_{st}$. Hence, the shortest path from s to t is computed in $\mathcal{M}_{st}$.

The computation of $\mathcal{M}_{st}$ uses the *tree of connected components* of $\mathcal{M}$ (also called *component tree*), which is denoted as $T_{\mathcal{M}}$. Formally, $T_{\mathcal{M}}$ is defined in what follows. For each level i, let us consider the subgraph of G that is induced by the nodes in $V \setminus S_i$. The set of connected components of this subgraph is denoted by $\mathcal{C}_i$. For a node $v \in V \setminus S_i$, let C_i^v denote the component in $\mathcal{C}_i$ that contains v. The nodes of $T_{\mathcal{M}}$ are the connected components in $\mathcal{C}_1 \cup \mathcal{C}_2 \cup \ldots \cup \mathcal{C}_l$. Additionally, there is a root C_{l+1} and, for each node $v \in V$, a leaf C_0^v in the tree. The parent of a leaf C_0^v is determined as follows. Let i be the largest level with $v \in S_i$. If

$i = l$, the parent is the root C_{l+1}. Otherwise, the level with smallest index where v is contained in a connected component is level $i+1$, and the parent of C_0^v is the component C_{i+1}^v. The parent of the components in $\mathcal{C}_l$ is the root C_{l+1}. For the remaining components $C_i \in \mathcal{C}_i$, the parent is the component C_{i+1}^u, $u \in C_i$.

The subgraph $\mathcal{M}_{st}$ of $\mathcal{M}$ is computed as follows. Let L be the level such that $C_L^s = C_L^t$ is the lowest common ancestor of C_0^s and C_0^t in $T_{\mathcal{M}}$. Then, the path $(C_0^s, C_k^s, C_{k+1}^s, \ldots, C_L^s = C_L^t, \ldots, C_{k'+1}^t, C_{k'}^t, C_0^t)$, from C_0^s to C_0^t in $T_{\mathcal{M}}$ induces a subgraph $\mathcal{M}_{st} = (V_{st}, E_{st})$ of the multi-level overlay graph $\mathcal{M}$ as follows. For each component $C \in \{C_0^s, C_k^s, C_{k+1}^s, \ldots, C_{L-1}^s\} \cup \{C_0^t, C_{k'}^t, C_{k'+1}^t, \ldots, C_{L-1}^t\}$, all edges of level i incident to a node in component C belong to E_{st}. Further, all edges of level L belong to E_{st}. V_{st} contains the nodes induced in G by edges in E_{st}. Once $\mathcal{M}_{st}$ has been computed, a s-t-distance query is answered by running Dijkstra's algorithm on $\mathcal{M}_{st}$. In [4], it has been experimentally shown that it is better to build $\mathcal{M}_{st}$ and run Dijkstra's algorithm on $\mathcal{M}_{st}$, rather than running Dijkstra's algorithm on G.

3 Computation of multi-level overlay graphs

In this Section we first give some theoretical properties of multi-level overlay graphs (that are proved in [21]), then we show how to use these properties to build a new algorithm for the computation of $\mathcal{M}$.

3.1 Characterization of level edges

Given a digraph G and the sets $S_1, \ldots, S_l$, the computation of $\mathcal{M}$ consists of calculating the level edges E_i, for each $i = 1, 2, \ldots, l$. For each $(u, v) \in S_i \times S_i$, (u, v) is an i-level edge if and only if for each shortest path P from u to v in G no internal node of P belongs to S_i. That is, if there exists a shortest path from u to v that contains a node in S_i different from u and v, then the pair (u, v) is not an i-level edge. For a fixed source u, and for each $v \in V$, let us denote as $P_u(v)$ the set of nodes x such that x is different from u and v, and x belongs to at least one shortest path from u to v in G. Furthermore, given $x \in V$, let us denote as $maxlevel(x)$ the maximum level containing x, that is $maxlevel(x) = \max\{j \mid x \in S_j\}$.

Definition 1. *Given $u, v \in V$, the* barrier level $s_u(v)$ *of pair (u, v) is:*

$$s_u(v) = \begin{cases} \max\{maxlevel(x) \mid x \in P_u(v)\} & \text{if } P_u(v) \not\equiv \emptyset \\ 0 & \text{if } P_u(v) \equiv \emptyset \end{cases}$$

Informally, the *barrier level* $s_u(v)$ of pair (u, v) is the maximum level containing a node in $P_u(v)$. Next lemma gives a property of level edges and barrier levels.

Lemma 1. *Let $j \in \{1, 2, \ldots, l\}$ and $u, v \in S_j$. The pair (u, v) is a j-level edge if and only if there exists a path from u to v in G and $s_u(v) < j$.*

For each $i = 1, 2, \ldots, l$, in order to test whether a pair $(u, v) \in S_i \times S_i$ is a i-level edge it is sufficient to compute $s_u(v)$. Since $s_u(v)$ does not depend on a specific level i and $S_1 \supset S_2 \supset \ldots \supset S_l$, then, we only need to compute $s_u(v)$, for each $(u, v) \in S_1 \times S_1$. It is clear that an edge (u, v) can belong to more than one level of $\mathcal{M}$, thus implying the necessity of multiple storing of each level edge. The next lemma gives a property that allows us to store a level edge only once.

Lemma 2. *If $e = (u, v) \in \bigcup_{i=1}^{l} E_i$, then there exist $j, k \in \mathbb{N}$, $1 \leq j \leq k \leq l$, such that $e \in E_i$, $\forall i \in \{j, j+1, \ldots, k\}$, and $e \notin E_i$, $\forall i \notin \{j, j+1, \ldots, k\}$.*

Lemma 2 allows us to store the multi-level overlay graph as follows. For each edge (u, v) belonging to $\bigcup_{i=0}^{l} E_i$, with $E \equiv E_0$, we store a triple

$$w_{\mathcal{M}}(u, v) = (\bar{d}(u, v), f(u, v), \ell(u, v)).$$

If (u, v) is a level edge, $\bar{d}(u, v)$, $f(u, v)$ and $\ell(u, v)$ are defined as follows:

- $\bar{d}(u, v)$ is equal $d(u, v)$;
- $f(u, v)$ is the smallest level j, with $1 \leq j \leq l$, such that $(u, v) \in E_j$. Since, by Lemma 1, (u, v) is a j-level edge only if $s_u(v) < j$, then $f(u, v) = s_u(v) + 1$;
- $\ell(u, v)$ is the largest level k, with $f(u, v) \leq k \leq l$, such that $(u, v) \in E_k$. Let $k' = maxlevel(u)$ and $k'' = maxlevel(v)$, then $\ell(u, v) = \min\{k', k''\}$.

If (u, v) is not a level edge, then $(\bar{d}(u, v), f(u, v), \ell(u, v)) = (w(u, v), 0, 0)$. By these definitions, to assign $w_{\mathcal{M}}(u, v)$, we need to know whether (u, v) is a level edge or not. The following lemma gives us a condition to recognize a level edge.

Lemma 3. *The pair $(u, v) \in S_1 \times S_1$ is a level edge if and only if there exists a path from u to v in G and $s_u(v) < \min\{maxlevel(u), maxlevel(v)\}$.*

In conclusion, in order to build $\mathcal{M}$, we need to compute $s_u(v)$ for each $u, v \in S_1$.

3.2 Computation of barrier levels

Given $G = (V, E, w)$, the sets $S_1, \ldots, S_l$ and $u, v \in S_1$, then $s_u(v)$ can be computed by running Dijkstra's shortest paths algorithm on a graph G_u obtained by suitably labelling the edges of G. Formally, for each $u \in S_1$, G_u is defined as follows: $G_u = (V, E, w_u)$, where $w_u(x, y) = (w(x, y), m_u(x))$ for each $(x, y) \in E$. Here, $w(x, y)$ is the weight of (x, y) in G, and

$$m_u(x) = \begin{cases} maxlevel(x) & \text{if } x \not\equiv u \\ 0 & \text{otherwise} \end{cases}$$

As shown in [22], Dijkstra's algorithm finds the single source shortest paths in a weighted graph when the edge weights are elements of a closed semiring. In what follows, we define an algebraic structure that is a closed semiring in such a way that, if weights w_u of edges in G_u are elements of this algebraic structure, then $(d(u, v), s_u(v))$ is the distance between u and v in G_u. Here, $d(u, v)$ is the distance from u to v in G.

Definition 2. $(\mathcal{K}, \min_{\mathcal{K}}, \oplus_{\mathcal{K}})$ *is an algebraic structure where:*

- $\mathcal{K} = \{(w, i) \mid w \in \mathbb{R}^+, i \in \mathbb{N}\} \cup \{(\infty, 0)\}$.
- *Given* $a_1 = (w_1, i_1)$ *and* $a_2 = (w_2, i_2)$ *in* $\mathcal{K}$*, the relation* $\leq_{\mathcal{K}}$ *is defined by*

$$a_1 \leq_{\mathcal{K}} a_2 \Leftrightarrow w_1 < w_2 \ \vee \ (w_1 = w_2 \ \wedge \ i_1 \geq i_2)$$

- *Given* $a_1, a_2 \in \mathcal{K}$*,*

$$\min_{\mathcal{K}}\{a_1, a_2\} = \begin{cases} a_1 & \text{if } a_1 \leq_{\mathcal{K}} a_2 \\ a_2 & \text{otherwise} \end{cases}$$

- *Given* $a_1 = (w_1, i_1)$ *and* $a_2 = (w_2, i_2)$ *in* $\mathcal{K}$*,*

$$a_1 \oplus_{\mathcal{K}} a_2 = \begin{cases} (w_1 + w_2, \ \max\{i_1, i_2\}) & \text{if } a_1 \neq (\infty, 0) \wedge a_2 \neq (\infty, 0) \\ (\infty, 0) & \text{if } a_1 = (\infty, 0) \vee a_2 = (\infty, 0) \end{cases}$$

The properties of $(\mathcal{K}, \min_{\mathcal{K}}, \oplus_{\mathcal{K}})$ are shown in the next theorem.

Theorem 1. $(\mathcal{K}, \min_{\mathcal{K}}, \oplus_{\mathcal{K}}, (\infty, 0), (0, 0))$ *is a closed semiring.*

Theorem 1 allows us to define the weight of a path and the distance from u to v in G_u as in the next definition.

Definition 3. *Let* $u \in S_1$ *and* $v \in V$*,*

- *let* $P = (u \equiv x_1, x_2, \ldots, x_k \equiv v)$ *be a path from* u *to* v *in* G_u*, the weight of* P *in* G_u *is defined as* $weight_{\mathcal{K}}(P) = w_u(x_1, x_2) \oplus_{\mathcal{K}} w_u(x_2, x_3) \oplus_{\mathcal{K}} \ldots \oplus_{\mathcal{K}} w_u(x_{k-1}, x_k)$
- *the distance from* u *to* v *in* G_u *is defined as* $d_u(v) = \min_{\mathcal{K}}\{weight_{\mathcal{K}}(P) \mid P \text{ is a path from } u \text{ to } v \text{ in } G_u\}$ *if there exists a path from* u *to* v *in* G_u*, while* $d_u(v) = (\infty, 0)$ *otherwise.*

Theorem 2. *Let* $G = (V, E, w)$ *be a weighted directed graph and* $u \in V$*. If* $G_u = (V, E, w_u)$ *is a graph where* $w_u : E \to \mathcal{K}$*, such that* $w_u(x, y) = (w(x, y), m_u(x))$ *for each* $(x, y) \in E$*, then* $d_u(v) = (d(u, v), s_u(v))$*, for each* $v \in V$*.*

Theorems 1 and 2 allows us to run Dijkstra's algorithm to compute $d(u, v)$ and $s_u(v)$. Hence, in order to compute all level edges of $\mathcal{M}$, we run Dijkstra's algorithm on G_u, for each node $u \in S_1$. As a result, we obtain a shortest paths tree T_u rooted in u such that, each node $v \in T_u$ is labeled with the distance from u to v in G_u that is, the pair $(d(u, v), s_u(v))$.

3.3 Computation of $\mathcal{M}$ and $T_{\mathcal{M}}$

First of all we have to compute the graphs G_u, for each $u \in S_1$. We assume that the sets $S_1, \ldots, S_l$ are given in input as a linked list $\mathtt{L}_{S_1}$ of the nodes in S_1 and an array $\mathtt{S}$ of size n such that, for each node $v \in V$, $\mathtt{S}[v] = maxlevel(v)$. The array $\mathtt{S}$ allows us to check in constant time whether a node belongs to a given

Input a graph $G = (V, E, w)$, a node $u \in V$, the array S

Output the graph $G_u = (V, E, w_u)$

Procedure LABEL

```
1.  for each (x, y) ∈ E do
2.    if x ≢ u then
3.      w_u(x, y) := (w(x, y), S[x])
4.    else
5.      w_u(x, y) := (w(x, y), 0)
```

Fig. 1.

level. As a consequence, for each $u \in S_1$, we can build graph G_u in linear time using Procedure LABEL in Figure 1.

Now, we show how to compute a multi-level overlay graph $\mathcal{M}$ as an adjacency list in $O(n + m + |\bigcup_{i=1}^{l} E_i|)$ optimal space. The solution we propose is given in Figure 2. Lines 1 and 2 initialize $w_{\mathcal{M}}(u, v)$ for each $(u, v) \in E$. The block at Lines 4–19 is performed for each node u in S_1. Line 5 computes G_u, while Line 6 computes $d(u, v)$ and $s_u(v)$, for each $v \in V$ (see Theorems 1 and 2). Lines 7–17 use $d(u, v)$ and $s_u(v)$ to compute $w_{\mathcal{M}}(u, v)$ for each $v \in S_1$. To this aim, block at Lines 7–13 visits the adjacency list of u and, using S, tests whether $v \in N(u)$ belongs to S_1 (Line 8). In the affirmative case, Lines 10 and 11 test whether (u, v) is a level edge (see Lemma 3) and, possibly, overwrites $w_{\mathcal{M}}(u, v)$ (see Lemma 2). Finally, Line 12 marks v to record that the edge (u, v) has been already visited and added to $\mathcal{M}$ as a level edge. Subsequently, for each pair (u, v) such that $v \in S_1$ and v is *unmarked* (see Line 15), Lines 16–17 test whether the pair (u, v) is a level edge (see Line 16) and, possibly, add (u, v) to $\mathcal{M}$ and set $w_{\mathcal{M}}(u, v)$ (see Line 17). Finally, Line 18 unmarks each $v \in V$.

Lemma 4. *Procedure* COMPUTEOVERLAY *requires* $O(|S_1|(m + n \log n))$ *time.*

Proof. Lines 1–2 require $O(n + m)$ time. Line 5 requires $O(n + m)$ time and is performed $|S_1|$ times, thus requiring $O(|S_1|(n+m))$ overall time. Line 6 is a Dijkstra's computation and hence requires $O(m+n \log n)$ time; since it is performed $|S_1|$ times, it requires $O(|S_1|(m + n \log n))$ overall time. Lines 7–13 require $O(n)$ worst case time and are performed $|S_1|$ times, thus requiring $O(n|S_1|)$ overall time. Lines 14–17 require $O(n)$ worst case time and are performed $|S_1|$ times, thus requiring $O(n|S_1|)$ overall time. Line 18 requires $O(n)$ worst case time and is performed $|S_1|$ times, thus requiring $O(n|S_1|)$ overall time. It follows that the total time needed to build $\mathcal{M}$ is $O(|S_1|(m + n \log n))$.

The component tree $T_{\mathcal{M}}$ is computed by visiting the subgraphs of G induced by nodes in $V \setminus S_i$, for each $i = 1, 2, \ldots, l$. This can be done in $O(l(n+m))$ worst case time. Since $l \leq |S_1|$, the time needed to compute $T_{\mathcal{M}}$ does not increase the overall preprocessing time. The component tree $T_{\mathcal{M}}$ is stored in a data structure denoted as $\mathtt{T}_{\mathcal{M}}$ and described in what follows:

Input a graph $G = (V, E, w)$, the array $\mathtt{S}$, the list $\mathtt{L}_{S_1}$

Output the graph $\mathcal{M} = (V, E \cup E_1 \cup E_2 \cup \ldots \cup E_l, w_{\mathcal{M}})$

Procedure ComputeOverlay

```
1.  for each (u,v) ∈ E do
2.    w_M(u,v) := (w(u,v), 0, 0)
3.  for each u ∈ L_{S_1} do
4.    begin
5.      G_aux := Label(G, u, S)
6.      Dijkstra(G_aux, u)
7.      for each v ∈ N(u) do
8.        if S[v] ≥ 1 then
9.          begin
10.           if (s_u(v) < min{S[u], S[v]} and d(u,v) ≠ ∞) then
11.             overwrite w_M(u,v) as (d(u,v), s_u(v) + 1, min{S[u], S[v]})
12.         mark(v)
13.         end
14.     for v := 1 to n do
15.       if S[v] ≥ 1 and unmarked(v) then
16.         if (s_u(v) < min{S[u], S[v]} and d(u,v) ≠ ∞) then
17.           add (u,v) to M with w_M(u,v) := (d(u,v), s_u(v) + 1, min{S[u], S[v]})
18.     for each v ∈ V do unmark(v)
19.   end
```

Fig. 2.

- for each $i = 1, 2, \ldots, l$, we store in a circularly linked list, denoted as $\mathtt{C}_i$, the connected components at level i of the set $\mathcal{C}_i$. For each $C \in \mathcal{C}_i$, the corresponding element in $\mathtt{C}$ contains the nodes in $C \setminus \bigcup_{v \in C} C_{i-1}^v$ and a link to its parent $\mathtt{C}_{i+1}$. Given a node $v \in V$, we denote as $\mathtt{C}_i^v$ the element of $\mathtt{C}_i$ corresponding to C_i^v;
- components in $\mathcal{C}_0$ (i.e., leaf components) are represented by an array $\mathtt{C}_0$. This array is indexed by nodes in V and $\mathtt{C}_0[v]$ contains a link to the element of $\mathtt{T}_{\mathcal{M}}$ corresponding to the parent of C_0^v in $T_{\mathcal{M}}$;
- the list $\mathtt{C}_{l+1}$ contains only one element representing the nodes in S_l.

3.4 Distance queries

As in [4], we answer s-t distance queries in two phases. First, we compute the subgraph $\mathcal{M}_{st} = (V_{st}, E_{st})$ of $\mathcal{M}$ described in Section 2, then we run Dijkstra's algorithm on $\mathcal{M}_{st}$. Procedure Compute$\mathcal{M}_{st}$ in Figure 3 shows the computation of $\mathcal{M}_{st}$ by using our data structures. In detail, Line 1 finds the path from C_0^s to C_0^t in the component tree. Lines 2–6 add to the edge set E_{st} of $\mathcal{M}_{st}$ all edges of level i incident to a node in component $C = C_i^x$, with $x \in \{s, t\}$ and $i < L$. Lines 7–10 add to E_{st} all edges of level L.

Lemma 5. *Procedure* Compute$\mathcal{M}_{st}$ *requires* $O(m + |S_1|^2)$ *worst case time.*

Input a multi-level overlay graph $\mathcal{M}$, the component tree $\mathtt{T}_{\mathcal{M}}$, nodes s and t

Output the graph $\mathcal{M}_{st}$

Procedure COMPUTE$\mathcal{M}_{st}$

1. Find the path $(\mathtt{C}_0^s, \mathtt{C}_k^s, \mathtt{C}_{k+1}^s, \ldots, \mathtt{C}_L^s = \mathtt{C}_L^t \ldots, \mathtt{C}_{k'+1}^t, \mathtt{C}_{k'}^t, \mathtt{C}_0^t)$ in $\mathtt{T}_{\mathcal{M}}$ where $C_L^s = C_L^t$ is the lowest common ancestor of C_0^s and C_0^t in $T_{\mathcal{M}}$
2. **for each** $\mathtt{C} \in \{\mathtt{C}_0^s, \mathtt{C}_k^s, \mathtt{C}_{k+1}^s, \ldots, \mathtt{C}_{L-1}^s\} \cup \{\mathtt{C}_0^t, \mathtt{C}_{k'}^t, \mathtt{C}_{k'+1}^t, \ldots, \mathtt{C}_{L-1}^t\}$ **do**
3. **for each** $v \in \mathtt{C}$ **do**
4. **for each** (v, z) in $\mathcal{M}$ **do**
5. **if** $(v, z) \in \bigcup_{j=i}^{L-1} E_j$ **then**
6. add (v, z) to E_{st} and z to V_{st}
7. **for each** $v \in \mathtt{C}_L^s$ **do**
8. **for each** (v, z) in $\mathcal{M}$ **do**
9. **if** $(v, z) \in E_L$ **then**
10. add (v, z) to E_{st} and z to V_{st}

Fig. 3.

Proof. Line 1 requires $O(m)$ time. In fact, in the worst case, each set $\mathtt{C} \in \{\mathtt{C}_0^s, \mathtt{C}_k^s, \mathtt{C}_{k+1}^s, \ldots, \mathtt{C}_{L-1}^s, \mathtt{C}_L^s\} \cup \{\mathtt{C}_0^t, \mathtt{C}_{k'}^t, \mathtt{C}_{k'+1}^t, \ldots, \mathtt{C}_{L-1}^t\}$, contains only one node. Therefore the number of these sets visited by the algorithm is at most $|V_{st}| \leq n = O(m)$. Lines 2-10 require $O(m + |S_1|^2)$ time. In fact, they consider the edges of $\mathcal{M}$ which belong either to E or to $\bigcup_{i=1}^{l} E_i$, and $|\bigcup_{i=1}^{l} E_i| \leq |S_1|^2$. For each considered edge, Lines 2-10 requires constant time. In fact, the test at Line 5 can be done by checking whether $((i \leq f(v,z) \leq L-1) \vee (i \leq \ell(v,z) \leq L-1) \vee (f(v,z) < i \wedge \ell(v,z) > L-1))$, and the test at Line 9 can be done by checking whether $f(v,z) \leq L \leq \ell(v,z)$. Hence, Lines 2–10 require $O(m + |S_1|^2)$ time.

Corollary 1. *An s-t distance query is answered in* $O(m + |S_1|^2 + |V_{st}| \log |V_{st}|)$ *time.*

4 Maintenance of Multi-Level Overlay Graphs

In this section we propose a dynamization of the algorithm given in Section 3.3, whose aim is to maintain the information on $\mathcal{M}(G; S_1, ..., S_l)$, when a sequence of update operations on the weights of G are performed. The dynamic environment we consider is defined as follows.

- We are given the following data structures:
  1. a weighted directed graph $G = (V, E, w)$;
  2. a sequence $S_1, S_2, \ldots, S_l$ of subsets of V such that $V \supset S_1 \supset S_2 \supset \ldots \supset S_l$, stored in the array $\mathtt{S}[\,]$ as defined in Section 3;
  3. the set S_1 stored in the list $\mathtt{L}_{S_1}$ as defined in Section 3.3;
  4. a multi-level overlay graph $\mathcal{M}(G; S_1, ..., S_l) = (V, E \cup E_1 \cup E_2 \cup \ldots \cup E_l)$, where E_i, $1 \leq i \leq l$, is the set of i-level edges , stored as adjacency lists;

5. the component tree $T_{\mathcal{M}}$ of $\mathcal{M}(G; S_1, ..., S_l)$ stored in the data structure $\mathtt{T}_{\mathcal{M}}$ as defined in Section 3;

– We are given a sequence $\sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_h \rangle$ of *modifications*, where a modification is either a *weight decrease* or a *weight increase* operation on an edge of G.
– Every time a modification occurs we have to update the information on $\mathcal{M}(G; S_1, ..., S_l)$, without recomputing it from scratch.

First of all, notice that the topology of the original graph G never changes as a consequence of a *weight decrease* or a *weight increase* operation, and the same happens to data structures $\mathtt{S}[\ ]$, $\mathtt{L}_{S_1}$ and $\mathtt{T}_{\mathcal{M}}$. This implies that we can answer s-t distance queries as described in Section 3.4, by simply constructing $\mathcal{M}_{st}$ and computing the distance from s to t in $\mathcal{M}_{st}$. Hence, in what follows we concentrate on the description of the dynamic algorithm to update $\mathcal{M}(G; S_1, ..., S_l)$. As shown in Section 3, the information needed to compute $\mathcal{M}(G; S_1, ..., S_l)$ can be stored in $|S_1|$ shortest paths trees. In particular, for each node $u \in S_1$, we need to store and maintain a shortest paths tree T_u such that, for each node $v \in T_u$, the distance of v is the pair $(d(u, v), s_u(v))$. Using this information we can recognize if edge (u, v) appears as a level edge: by Lemma 3, (u, v) is a level edge if and only if $s_u(v) < \min\{maxlevel(u), maxlevel(v)\}$ and there exists a path from u to v in G. As a consequence, every time a *weight decrease* or a *weight increase* operation occurs on G, it is sufficient to update the $|S_1|$ shortest paths trees T_u, $u \in S_1$. To this aim, we apply to each T_u, the fully dynamic algorithm proposed in [20] to update shortest paths.

The algorithm in [20] works for any graph and its complexity depends on the existence of a so called *k-bounded accounting function* for G as defined below.

Definition 4. [20] *Let $G = (V, E, w)$ be a weighted graph, and $s \in V$ be a source node. An* accounting function *for $G = (V, E, w)$ is any function $A : E \to V$ such that, for each $(x, y) \in E$, $A(x, y)$ is either x or y, which is called the owner of (x, y). A is* k-bounded *if, for each $x \in V$, the set of the edges owned by x has cardinality at most k.*

As an example, if G is planar then, there exists a 3-bounded accounting function for G, while for a general graph with m edges $k = O(\sqrt{m})$. Furthermore, it is easy to see that, if G has average degree equal to d ($d = m/n$), then there exists a k-bounded accounting function for G where $k = O(d)$.

In detail, for any sequence of weight increase and weight decrease operations, if the final graph has a k-bounded accounting function, then the complexity of the algorithm in [20] is $O(k \log n)$ worst case time per output update.

To obtain this bound, every time a node z changes the distance to the source, the algorithm in [20] needs to know the *right* edges adjacent to z that have to be scanned. To efficiently deal with this problem, the algorithm requires some auxiliary data structure that stores the information given in the next definition.

Definition 5. [20] *Let $G = (V, E, w)$ be a weighted graph, and $s \in V$ be a source node. The* backward_level *(*forward_level*) of edge (z, q) and of node q, relative to*

node z, is the quantity $b\_level_s(z,q) = d(s,q) - w(z,q)$ ($f\_level_s(z,q) = d(s,q) + w(z,q)$).

The intuition behind Definition 5 is that the level of an edge (z,q) provides information about the shortest available path from s to q passing through z. For instance, let us suppose that, while processing a *weight decrease* operation, the new distance of z, denoted as $d'(s,z)$, decreases below $b\_level_s(z,q)$, i.e., there exists an edge (z,q) such that $b\_level_s(z,q) - d'(s,z) = d(s,q) - w(q,z) - d'(s,z) > 0$, i.e., $d(s,q) > d'(s,z) + w(q,z)$. This means that we have found a path to q shorter than the current shortest path to q. In this case, scanning the edges (z,q) in nonincreasing order of b_level ensures that only the *right* edges are considered, i.e., edges (z,q) such that also q decreases the distance from s. The case of a *weight increase* operation is analogous.

To apply the above strategy, the algorithm of [20] needs to maintain explicitly the information on the b_level and the f_level for all the neighbors of each node. This might require the scanning of each edge adjacent to an updated node.

To bound the number of edges scanned by the algorithm each time that a node is updated, the set of edges adjacent to each node is partitioned in two subsets: any edge (x,y) has an *owner*, denoted as *owner*(x,y), that is either x or y. For each node x, *ownership*(x) denotes the set of edges owned by x, and *not-ownership*(x) denotes the set of edges with one endpoint in x, but not owned by x. If G has a k-bounded accounting function then, for each $x \in V$, *ownership*(x) contains at most k edges. Furthermore, the edges in *not-ownership*(x) are stored in two priority queues as follows:

1. $B_{s,x}$ is a max-based priority queue; the priority of edge (x,y) (of node y) in $B_{s,x}$, denoted as $b_s(x,y)$, is the computed value of $b\_level_s(x,y)$;
2. $F_{s,x}$ is a min-based priority queue; the priority of edge (x,y) (of node y) in $F_{s,x}$, denoted as $f_s(x,y)$, is the computed value of $f\_level_s(x,y)$.

While the definition of accounting function can be borrowed from [20] as it is, the definition of backward and forward levels have to be adapted to our context. To this aim, we need to define two further binary operators in $\mathcal{K}$ working on quantities defined in G_u: $\ominus_{\mathcal{K}}$ and $\max_{\mathcal{K}}$.

Definition 6. *For each $v \in V$, for each $(q,v) \in E$, and for each $u \in S_1$,*

$$\begin{aligned} d_u(v) \ominus_{\mathcal{K}} w_u(q,v) &= (d(u,v), s_u(v)) \ominus_{\mathcal{K}} (w(q,v), m_u(v)) \\ &= (d(u,v) - w(q,v), s_u(q)). \end{aligned}$$

Definition 7. *Given $a_1, a_2 \in \mathcal{K}$,*

$$\max_{\mathcal{K}}\{a_1, a_2\} = \begin{cases} a_1 & \text{if } a_2 \leq_{\mathcal{K}} a_1 \\ a_2 & \text{otherwise.} \end{cases}$$

It is easy to see that $\mathcal{K}$ is closed under $\max_{\mathcal{K}}$ and that $\max_{\mathcal{K}}$ is associative, while $\ominus_{\mathcal{K}}$ is defined on a subset of $\mathcal{K} \times \mathcal{K}$, given by distances and weights in G_u. According to the definition of operators $\ominus_{\mathcal{K}}$ and $\oplus_{\mathcal{K}}$, we redefine the notions of *backward_level* and *forward_level* as follows.

Definition 8. *Let $u \in S_1$, and let (v, q) and q be an edge and a node in G_u, respectively. The* backward_level *and* forward_level *of (v, q) are defined, respectively, as follows:*

$$b\_level_u(v, q) = d_u(q) \ominus_{\mathcal{K}} w_u(v, q)$$

$$f\_level_u(v, q) = d_u(q) \oplus_{\mathcal{K}} w_u(v, q)$$

We store these information in the following data structures:

- for each $v \in V$, *ownership*(v), that is the set of edges owned by v, stored as a linked list (note that, an ownership function for the graph $G = (V, E, w)$ is also an ownership function for graphs G_u, for each $u \in S_1$; hence, these information have to be stored only once);
- for each $v \in V$, *not-ownership*(v), that is the set of edges with an endpoint in v but not owned by v. For each $v \in V$ and for each G_u, $u \in S_1$, *not-ownership*(v) is stored in two priority queues as follows:
  1. $B_u(v)$ is a max-based priority queue; the priority of edge (v, q) in $B_u(v)$, is the computed value of $b\_level_u(v, q)$ in G_u with respect to source u. Here, the maximum is computed as in Definition 7;
  2. $F_u(v)$ is a min-based priority queue; the priority of edge (v, q) in $F_u(v)$, is the computed value of $f\_level_u(v, q)$ in G_u with respect to source u.

Hence, in order to use the algorithm in [20] to update trees T_u, $u \in S_1$, we have to compute and store the above data structures before the sequence of edge modifications occurs. Algorithm ComputeOverlay given in Section 3.3 is not suitable to be used in the dynamic environment described above since it does not store trees T_u, $u \in S_1$. In fact, it computes only one shortest paths tree at a time and computes $\mathcal{M}$ stepwise. Thus, we propose a new preprocessing algorithm, denoted as PreprocessOverlay and shown in Figure 4. This algorithm is similar to ComputeOverlay but it first computes all the $|S_1|$ shortest paths trees along with the above auxiliary data structures, and then uses these trees to compute $\mathcal{M}$.

PreprocessOverlay works as follows. Line 1 computes an accounting function of G as the sets *ownership*(v) and *not-ownership*(v), for each $v \in V$. The instructions at Lines 3–9 are performed for each $u \in S_1$. In particular, Lines 4 and 5 compute and store the graphs G_u and the shortest paths trees T_u. Lines 6–8 compute the queues $B_u(v)$ and $F_u(v)$ for each node $v \in V$. Lines 10 and 11 initialize $w_{\mathcal{M}}(u, v)$ for each $(u, v) \in E$. Then, Lines 12–26 compute $w_{\mathcal{M}}(u, v)$, for each $(u, v) \in \bigcup_{i=0}^{l} E_i$ using the information on $d(u, v)$ and $s_u(v)$, for each $u \in S_1$ and for each $v \in V$, stored in the trees T_u. The computation of $w_{\mathcal{M}}(u, v)$ is performed as in ComputeOverlay.

The correctness of the Procedure PreprocessOverlay is a straightforward consequence of Lemmata 2 and 3, and Theorems 1 and 2. The time complexity of Procedure PreprocessOverlay is given in the next lemma.

Lemma 6. *Procedure* PreprocessOverlay *requires $O(|S_1|(m+n) \log n)$ time.*

Input a graph $G = (V, E, w)$, the array $\mathtt{S}$, the list $\mathtt{L}_{S_1}$

Output the graph $\mathcal{M} = (V, E \cup E_1 \cup E_2 \cup \ldots \cup E_l, w_{\mathcal{M}})$

Procedure PREPROCESSOVERLAY

```
1.  Compute an accounting function for G
2.  for each u ∈ L_{S_1} do
3.    begin
4.      G_u := LABEL(G, u, S)
5.      T_u := Dijkstra(G_u, u)
6.      for each v ∈ V do
7.        for each (v, q) ∈ not − ownership(v) do
8.          compute b_level_u(v, q), f_level_u(v, q) and add (v, q) to B_u(v) and F_u(q)
9.    end
10. for each (u, v) ∈ E do
11.   w_M(u, v) := (w(u, v), 0, 0)
12. for each u ∈ L_{S_1} do
13.   begin
14.     for each v ∈ N(u) do
15.       if S[v] ≥ 1 then
16.         begin
17.           if (s_u(v) < min{S[u], S[v]} and d(u, v) ≠ ∞) then
18.             overwrites w_M(u, v) as (d(u, v), s_u(v) + 1, min{S[u], S[v]})
19.           mark(v)
20.         end
21.     for v := 1 to n do
22.       if S[v] ≥ 1 and unmarked(v) then
23.         if (s_u(v) < min{S[u], S[v]} and d(u, v) ≠ ∞)then
24.           add (u, v) to M with w_M(u, v) := (d(u, v), s_u(v) + 1, min{S[u], S[v]})
25.     for each v ∈ V do unmark(v)
26.   end
```

Fig. 4.

Proof. Line 1 requires $O(m)$ time (see [23]). Lines 4–5 require $O(|S_1|(m + n \log n))$ time. Lines 6–8 requires $O(|S_1|m \log n)$ time. Lines 10–11 requires $O(n + m)$ time. As in COMPUTEOVERLAY, Lines 12–26 require $O(n)$ worst case time and are performed $|S_1|$ times, thus requiring $O(n|S_1|)$ overall time. Summing up these values, the total time needed by PREPROCESSOVERLAY to build $\mathcal{M}(G; S_1, \ldots, S_l)$ is $O(|S_1|(m + n) \log n)$.

The space requirements to store $\mathcal{M}(G; S_1, \ldots, S_l)$ and the additional data structures used for the maintenance of $\mathcal{M}$ is $O((n + m)|S_1|)$.

The data structure computed by PREPROCESSOVERLAY has to be updated during the sequence $\sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_h \rangle$ of modifications on G. Our dynamic solution starts after each σ_i and works in three phases as follows:

Procedure DYNAMICOVERLAY

1. Update G_u, for each $u \in S_1$;

2. Apply the fully dynamic algorithm for shortest paths given in [20] to each T_u, $u \in S_1$;
3. Perform Lines 10–26 of PreprocessOverlay to build $\mathcal{M}$ using the new values of $d(u, v)$ and $s_u(v)$, updated at phase 2 above.

Let δ_u be the set of nodes in G_u that change either the distance or the shortest path to u as a consequence of a *weight decrease* or a *weight increase* operation. If we denote as Δ the quantity $\sum_{u \in S_1} |\delta_u|$ and considering a k-bounded accounting function for G, then the cost of the algorithm is given in the next lemma.

Lemma 7. *The fully dynamic algorithm requires* $O(|S_1|n + m + k\Delta \log n)$ *time per operation.*

Proof. Phase 1 requires $O(|S_1|n)$ time. By definition of Δ, Phase 2 requires $O(k\Delta \log n)$ worst case time as shown in [20]. Phase 3 requires $O(|S_1|n + m)$ worst case time as shown in the proof of Lemma 6. Thus, the fully dynamic algorithm requires $O(|S_1|n + m + k\Delta \log n)$ time per operation.

The correctness of Phases 1 and 3 above is straightforward, while the correctness of Phase 2 comes from [20].

5 Discussion

In this section we propose a critical evaluation of our dynamic solution. The aim of this discussion is to capture the values of parameters $|S_1|$ and Δ that make our fully dynamic solution better than the recomputation from scratch. Since no theoretical results is known for the construction of a multi-level overlay graph of a given graph, we compare the new fully dynamic solution DynamicOverlay with the optimal space solution ComputeOverlay given in Section 3.3, that requires $O(|S_1|(m + n \log n))$ time.

We first bound the value of Δ. Notice that by definition $\Delta = O(|S_1 \times V|) = O(|S_1|n)$. We analyze the cases of sparse graphs, random graphs and dense graphs. In any case, we derive the values of Δ for which the dynamic algorithm is better than the recomputation from scratch, that is the values of Δ for which $O(|S_1|n+m+k\Delta \log n)$ is asymptotically better than $O(|S_1|(m+n \log n))$. More precisely, the values of Δ such that:

$$|S_1|n + m + k\Delta \log n = o(|S_1|(m + n \log n))$$

Since $|S_1|n + m = o(|S_1|(m + n \log n))$, then we need the values of Δ such that:

$$k\Delta \log n = o(|S_1|(m + n \log n)) \tag{1}$$

Sparse graphs In this case $m = O(n)$. This implies that $k = O(1)$. Hence, by inequality (1) we obtain:

$$\Delta \log n = o(|S_1|n \log n)$$

$$\Delta = o(|S_1|n)$$

Random graphs In this case we consider random graphs that are connected with high probability, that is graphs such that $m = O(n \log n)$ (see [24]). This implies that $k = O(\log n)$. Hence, by inequality (1) we obtain:

$$\Delta \log^2 n = o(|S_1| n \log n)$$

$$\Delta = o(|S_1| n / \log n)$$

Dense graphs In this case $m = O(n^2)$. This implies that $k = O(n)$. Hence, by inequality (1) we obtain:

$$n\Delta \log n = o(|S_1| n^2)$$

$$\Delta = o(|S_1| n / \log n)$$

Summarizing, in the case of sparse graphs DynamicOverlay is asymptotically better than the recomputation from scratch by applying ComputeOverlay; while, in the case of random graphs and dense graphs, DynamicOverlay is better than the recomputation from scratch by applying ComputeOverlay when Δ is at least a $\log n$ factor far from its maximum value.

Now we need to bound the value of $|S_1|$. Let us consider the space needed by the dynamic algorithm, which is $O(|S_1|(n+m) + |\bigcup_{i=1}^{l} E_i|)$, compared with the space needed by the static solution, which is $O(n + m + |\bigcup_{i=1}^{l} E_i|)$. Notice that, the value $|S_1|$ appears in the space requirements of the dynamic algorithm. To keep the space occupancy of the dynamic algorithm within that of the static algorithm, we need to fix $|S_1| = O(1)$. In this case, the time needed to perform an s-t query, given in Section 3.4, becomes $O(m + |V_{st}| \log |V_{st}|)$. A very ambitious open problem in this area is to develop a theoretical framework that help to properly choose the sets $S_1, S_2, \ldots, S_l$ in order to speed up as much as possible shortest path queries.

Acknowledgements

We like to thank Prof. Luigia Berardi for the constructive discussion and useful comments on algebraic structures.

References

1. Bast, H., Funke, S., Matijevic, D., Sanders, P., Schultes, D.: In transit to constant shortest-path queries in road networks. In: Workshop on Algorithm Engineering and Experiments (ALENEX07), SIAM (2007) 46–59
2. Goldberg, A., Harrelson, C.: Computing the shortest path: A* search meets graph theory. In: Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA05), SIAM (2005) 156–165
3. Goldberg, A., Kaplan, H., Werneck, R.: Reach for A*: Efficient point to point shortest path algorithms. In: Workshop on Algorithm Engineering and Experiments (ALENEX06), SIAM (2006)

4. Holzer, M., Schulz, F., Wagner, D.: Engineering multi-level overlay graphs for shortest-path queries. In: Proceedings of the Eight Workshop on Algorithm Engineering and Experiments (ALENEX06), SIAM (2006) 156–170
5. Holzer, M., Schulz, F., Wagner, D., Willhalm, T.: Combining speed-up techniques for shortest-path computations. ACM J. of Experimental Algorithmics **10** (2006)
6. Möhring, R.H., Schilling, H., Schutz, B., Wagner, D., Willhalm, T.: Partitioning graphs to speed-up Dijkstra's algorithm. In: Workshop on Experimental and Efficient Algorithms (WEA05). Volume 3503 of LNCS. (2005) 189–202
7. Pyrga, E., Schulz, F., Wagner, D., Zaroliagis, C.: Experimental comparison of shortest path approaches for timetable information. In: 6th Workshop on ALgorithm ENgineering and EXperiments (ALENEX04), SIAM (2004) 88–99
8. Schulz, F., Wagner, D., Willhalm, T.: Dijkstra's algorithm on-line: An empirical case study from public railroad transport. ACM Journal of Experimental Algorithmics **5** (2000)
9. Schulz, F., Wagner, D., Zaroliagis, C.: Using multi-level graphs for timetable information in railway systems. In: Workshop on ALgorithm ENgineering and EXperiments (ALENEX02). Volume 2409 of LNCS., Springer (2002) 43–59
10. Wagner, D., Willhalm, T.: Geometric speed-up techniques for finding shortest paths in large sparse graphs. In: Proceedings of 11-th European Symposium on Algorithms (ESA03). LNCS, Springer (2003) 776–787
11. Delling, D., Holzer, M., Müller, K., Schulz, F., Wagner, D.: High-performance multi-level graphs. Technical Report 0012, Project ARRIVAL (2006)
12. Sanders, P., Schultes, D.: Highway hierarchies hasten exact shortest path queries. In: 13th European Symposium on Algorithms (ESA). Volume 3669 of LNCS., Springer (2005)
13. Sanders, P., Schultes, D.: Engineering highway hierarchies. In: 14th European Symposium on Algorithms (ESA). Volume 4168 of LNCS., Springer (2006)
14. Köhler, E., Möhring, R., Schilling, H.: Acceleration of shortest path and constrained shortest path computation. In: Workshop on Experimental and Efficient Algorithms (WEA05). Volume 3503 of LNCS., Springer (2005)
15. Willhalm, T., Wagner, D.: Shortest paths speed-up techniques. In: Algorithmic Methods for Railway Optimization. LNCS, Springer (2006)
16. Delling, D., Wagner, D.: Landmark-based routing in dynamic graphs. In: 6th Workshop on Experimental Algorithms (WEA07). LNCS, Springer (2007) 52–65
17. Wagner, D., Willhalm, T., Zaroliagis, C.: Dynamic shortest path containers. Electronic Notes in Theoretical Computer Science **92** (2003)
18. Demetrescu, C., Italiano, G.F.: A new approch to dynamic all pairs shortest paths. Journal of ACM **51** (2004) 968–992
19. Schultes, D., Sanders, P.: Dynamic highway-node routing. In: 6th Workshop on Experimental Algorithms (WEA07). LNCS, Springer (2007) 66–79
20. Frigioni, D., Marchetti-Spaccamela, A., Nanni, U.: Fully dynamic algorithms for maintaining shortest paths trees. Journal of Algorithms **34** (2000) 251–281
21. Bruera, F., Cicerone, S., D'Angelo, G., Stefano, G.D., Frigioni, D.: On the dynamization of shortest path overlay graphs. Technical Report 0026, ARRIVAL (2006)
22. Mohri, M.: Semiring frameworks and algorithms for shortest-distance problems. Journal of Automata, Languages and Combinatorics **7(3)** (2002) 321–350
23. Frigioni, D., Marchetti-Spaccamela, A., Nanni, U.: Fully dynamic shortest paths in digraphs with arbitrary arc weights. Journal of Algorithms **49** (2003) 86–113
24. Bollobas, B.: Random Graphs. London Academic Press (1985)