The Power of Dynamic Distance Oracles: Efficient Dynamic Algorithms for the Steiner Tree

Jakub Łącki  
University of Warsaw  
Warsaw, Poland  
j.lacki@mimuw.edu.pl

Jakub O’cwieja  
University of Warsaw  
Warsaw, Poland  
j.ocwieja@mimuw.edu.pl

Marcin Pilipczuk  
University of Warwick  
Warwick, United Kingdom  
malcin@mimuw.edu.pl

Piotr Sankowski  
University of Warsaw  
Warsaw, Poland  
sank@mimuw.edu.pl

Anna Zych  
University of Warsaw  
Warsaw, Poland  
anka@mimuw.edu.pl

ABSTRACT

In this paper we study the Steiner tree problem over a dynamic set of terminals. We consider the model where we are given an n-vertex graph \( G = (V,E,w) \) with positive real edge weights, and our goal is to maintain a tree which is a good approximation of the minimum Steiner tree spanning a terminal set \( S \subseteq V \), which changes over time. The changes applied to the terminal set are either terminal additions (incremental scenario), terminal removals (decremental scenario), or both (fully dynamic scenario). Our task here is twofold. We want to support updates in sublinear \( o(n) \) time, and keep the approximation factor of the algorithm as small as possible.

We show that we can maintain a \((6 + \epsilon)\)-approximate Steiner tree of a general graph in \( \tilde{O}(\sqrt{n} \log D) \) time per terminal addition or removal. Here, \( D \) denotes the stretch of the metric induced by \( G \). For planar graphs we achieve the same running time and the approximation ratio of \((2 + \epsilon)\).

Moreover, we show faster algorithms for incremental and decremental scenarios. Finally, we show that if we allow higher approximation ratio, even more efficient algorithms are possible. In particular we show a polylogarithmic time \((4 + \epsilon)\)-approximate algorithm for planar graphs.

One of the main building blocks of our algorithms are dynamic distance oracles for vertex-labeled graphs, which are of independent interest. We also improve and use the online algorithms for the Steiner tree problem.

Categories and Subject Descriptors

F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems; G.2.2 [Discrete Mathematics]: Graph Theory

Keywords

Steiner tree; dynamic algorithm; dynamic distance oracles; approximation

1. INTRODUCTION

Imagine a network and a set of users that want to maintain a cheap multicast tree in this network during a conference call [16]. The users can join and leave, but in the considered time scale the network remains static. In other words we are considering the following problem. We are given a graph \( G = (V,E,w) \) with positive edge weights \( w : E \rightarrow \mathbb{R}^+ \). The goal is to maintain information about approximate Steiner tree in \( G \) for a dynamically changing set \( S \subseteq V \) of terminals.

This problem was first introduced in the pioneering paper by Imase and Waxman [25] and its study was later continued in [27, 19, 20]. However, all these papers focus on minimizing the number of changes to the tree that are necessary to maintain a good approximation, and ignore the problem of efficiently finding these changes. The problem of maintaining a Steiner tree is also one of the important open problems in the network community [13], and while it has been studied for many years, the research resulted only in several heuristic approaches [5, 2, 24, 30] none of which has been formally proved to be efficient. In this paper we show the first sublinear time algorithm which maintains an approximate Steiner tree under terminal additions and deletions.

Our paper deals with two variants of the problem of maintaining the Steiner tree. Throughout this paper, we assume that in the online Steiner tree problem (we also say online setting) the goal is to maintain a Steiner tree making few changes to the tree after each terminal addition or removal. On the other hand, in the dynamic Steiner tree problem (dynamic setting), the requirement is that each update is processed faster than by recomputing the tree from scratch.

This aligns with the usual definition of a dynamic algorithm used in the algorithmic community (see e.g. [17]). In this paper we study both settings and use the techniques for the online setting to show new results in the dynamic setting.

As our point of reference we observe that it is possible to construct \( \tilde{O}(n) \) time dynamic algorithm for Steiner tree using the dynamic polylogarithmic time minimum spanning forest (dynamic MSF) algorithm [23]. This solution is obtained by first computing the metric closure \( \overline{G} \) of the graph \( G \), and then maintaining the minimum spanning tree (MST)
over the set of terminals $S$ in $\overline{G}(S)$. It is a well-known fact that this yields a 2-approximate Steiner tree. In order to update $\overline{G}(S)$ we need to insert and remove terminals together with their incident edges, which requires $\Theta(n)$ calls to the dynamic MSF structure. However, such a linear bound is far from being satisfactory as does not lead to any improvement in the running time for sparse networks where \( m = O(n) \).¹

In such networks after each update we can actually compute the 2-approximate Steiner tree in $O(n \log n)$ time from scratch [28]. Hence, the main challenge is to break the linear time barrier for maintaining constant approximate tree. Only algorithms with such sublinear complexity could be of some practical importance and could potentially be used to reduce the communication cost of dynamic multicast trees. Our paper aims to be a theoretical proof of concept that from algorithmic complexity perspective this is indeed possible.

As observed by [10] and [11], the dynamic problems with vertex updates are much more challenging, but are actually closer to real-world network models than problems with edge updates. In computer networks, vertex updates happen more often, as they correspond to software events (server reboot, misconfiguration, or simply activation of a user), whereas edge updates are much less likely, as they correspond to physical events (cable cut/repair).

Finally, we note that the Steiner tree problem is one of the most fundamental problems in combinatorial optimization. It has been studied in many different settings, starting from classical approximation algorithms [6, 29, 4, 31, 8, 9], through online [25, 27, 19, 20] and stochastic models [21, 18], ending with game theoretic approaches [3, 7]. Taking into account the significance and the wide interest in this problem, it is somewhat surprising that no efficient dynamic algorithms for this problem have been developed so far. It might be related to the fact that this would require combining ideas from the area of approximation algorithms with the tools specific to dynamic algorithms. This is the first paper that manages to do so.

In this extended abstract we only present the main ideas, skipping the proofs and the formal analysis, which will be presented in the full version of this paper.

1.1 Our results

The main result of this paper are sublinear time algorithms for maintaining an approximate Steiner tree. We provide different algorithms for incremental, decremental and fully dynamic scenarios. An incremental algorithm allows only to add terminals, a decremental algorithm supports removing terminals, whereas a final fully dynamic algorithm supports both these operations. Our results are summarized in Table 1. The overall approximation ratio of the algorithms we obtain is $(6 + \varepsilon)$ for general graphs and $(2 + \varepsilon)$ for planar graphs. In particular, we can maintain a fully dynamic $(6 + \varepsilon)$-approximate tree in $O(\sqrt{n \log D})$ amortized time per update in an arbitrary weighted graph, where $D$ is the stretch of the metric induced by the graph. This extended abstract aims to present a brief overview of this result. The result is a composition and a consequence of many ideas that are of independent interest. We believe that the strength of this paper lies not only in the algorithms we propose, but also in the byproducts of our construction. We outline these additional results below.

Dynamic vertex-color distance oracles.

The algorithms for online Steiner tree assume that the entire metric (i.e., the distances between any pair of vertices) is given explicitly. This assumption is not feasible in case of a metric induced by a graph. Hence, to obtain the necessary distances efficiently, we develop a data structure called dynamic vertex-color distance oracle. This oracle is given a weighted undirected graph and maintains an assignment of colors to vertices. While the graph remains fixed, the colors may change. The oracle can answer, among other queries, what is the nearest vertex of color $c$ to a vertex $v$. We develop two variants of approximate vertex-color distance oracles: incremental and fully dynamic. In the first variant, each vertex is initially given a distinct color and the color sets (i.e., two sets representing vertices of the same color) can be merged. The fully dynamic oracles additionally support (restricted) operations of splitting color sets.² Note that these update operations are much more general than the operation of changing the color of a single vertex, which was considered in earlier works [22, 12].

For planar graphs we propose two $(1 + \varepsilon)$-approximate oracles. The incremental oracle supports all operations in $O(\varepsilon^{-1} \log^2 n \log D)$ amortized time (in expectation), whereas the fully dynamic oracle supports operations in worst case time $O(\varepsilon^{-1} \sqrt{n \log^2 n \log D})$. For general graphs we introduce a $3$-approximate fully dynamic oracle that works in $O(\sqrt{n \log n})$ expected time. Our construction of oracles is generic, that is we show how to extend oracles that may answer vertex-to-vertex queries and satisfy certain conditions into dynamic vertex-color oracles. For that we introduce the concept of a generic distance oracle, which captures the common properties of many distance oracles and allows us to use a uniform approach for different oracles for planar and general graphs.

Online Steiner tree.

We show an online algorithm that decrementally maintains $(2 + \varepsilon)$-approximate Steiner tree, applying after each terminal deletion $O(\varepsilon^{-1})$ changes to the tree (in amortized sense). This improves over the previous 4-approximate algorithm. In addition to that, we show a fully dynamic $(2 + \varepsilon)$-approximate online algorithm, which makes $O(\varepsilon^{-1} \log D)$ changes to the tree (in amortized sense) after each operation. One of the new techniques used to obtain these results is the lazy handling of high degree Steiner nodes for arbitrary degree threshold. This improves over an algorithm by Imase and Waxman [25], which makes $O(\varepsilon^{3/2})$ changes to process $t$ addition or removal operations, and maintains a 4-approximate tree. An algorithm performing a smaller number of changes was given in [20], but, as we discuss in the next section, it departs slightly from the classical Imase-Waxman model.

Query Steiner tree.

In the query model, as defined in [15], for a fixed graph $G$, we are asked queries to compute an approximate Steiner

¹It is widely observed that most real-world networks are sparse [14].

²Because of these two operations we believe that it is more natural to change slightly the previously used vocabulary and assign colors instead of labels to vertices.
tree for a set $S \subseteq V$ as fast as possible. This models a situation when many sets of users want to setup a new multicast tree. We obtain an algorithm, which after preprocessing in $O(\sqrt{n}(m + n \log n))$ expected time uses $O(n\sqrt{n} \log n)$ space and computes a 6-approximate Steiner tree in $O(|S|\sqrt{n} \log n)$ expected time. In the planar case, we can compute $(2 + \varepsilon)$-approximate tree in $O(|S|\varepsilon^{-1} \log^2 n \log D)$ expected time, using $O(\varepsilon^{-1}n \log^2 n \log D)$ preprocessing time and space. In other words, we show a more efficient solution for computing many multicast trees in one fixed graph than computing each tree separately. This preprocessing problem is related to the study initiated in [1] where compact multicast routing schemes were shown. In comparison with [1], our schemes are not only compact but efficient as well.

**Nonrearrangeable incremental Steiner tree.**

In the nonrearrangeable incremental Steiner tree problem one has to connect arriving terminals to the previously constructed tree without modifying it. We show how to implement the $O(\log n)$-approximate online algorithm for this problem given by Imase and Waxman [25] so that it runs in $O(\sqrt{n} \log n)$ expected time for non-planar graphs and $O(r \log^2 n \log D)$ expected time for planar graphs. Here $r$ denotes the final number of terminals. This gives an improvement over the naive execution of this algorithm that requires $O(r^2)$ time and resolves one of the open problems in [13].

**Bipartite emulators.**

As an interesting side result, we also show a different, simple approach to dynamic Steiner tree, which exposes a trade-off between the approximation ratio and the running time. It is based on **bipartite emulators**: low-degree bipartite graphs that approximate distances in the original graph. We run the dynamic MSF algorithm on top of a bipartite emulator to obtain sublinear time dynamic algorithms. In particular, we obtain a 12-approximate algorithm for general graphs that processes each update in $O(\sqrt{n})$ expected amortized time and a $(4 + \varepsilon)$-approximate algorithm for planar graphs processes updates in $O(\varepsilon^{-1} \log^6 n)$ amortized time. While our emulators are constructed using previously known distance oracles [34, 33], our contribution lies in introducing the concept of bipartite emulators, whose properties make it possible to solve the Steiner tree problem with a modification of the dynamic MSF algorithm [23].

We want to stress that the construction of our algorithms for the Steiner tree, in particular the approach that combines online Steiner tree algorithm with a distance oracle, is highly modular. Not only any improvement in the construction of the vertex-color distance oracles will result in better Steiner tree algorithms, but the vertex-color distance oracles themselves are constructed in a generic way out of distance oracles in [33, 34]. The approximation factor of $(6 + \varepsilon)$ for Steiner tree in general graphs comes from using 3-approximate oracles combined with a 2-approximation of the Steiner tree given by the MST in the metric closure and $(1 + \varepsilon)$-approximate online MST algorithm. In other words we hit two challenging bounds: in order to improve our approximation factors, one would need either to improve the approximation ratio of the oracles which are believed to be optimal, or devise a framework not based on computing the MST. The second challenge would require to construct simple and fast (e.g., near-linear time) approximation algorithms for Steiner tree that would beat the MST approximation ratio of 2. Constructing such algorithms is a challenging open problem.

### 1.2 Related results

The problems we deal with in this paper and related have received a lot of attention in the literature. We present a brief summary in this section.

**Vertex-color distance oracles.**

Our vertex-color distance oracles fall into the model studied in the literature under the name of vertex-label distance oracles. Dynamic vertex-color oracles for general graphs have been introduced by Hermelin et al. [22] and improved by Chechik [12]. These oracles allow only to change a color (label) of a single vertex, as opposed to our split and merge operations. The oracle by Chechik [12] has expected size $O(n^{1+1/k})$, and reports $(4k - 5)$-approximate distances in $O(k)$ time. This oracle can support changes of vertices’ colors in $O(n^{1/k} \log n)$ time. Our results have much better approximation guarantee and more powerful update operations, at the cost of higher query time. A vertex-color oracle for planar graphs has been shown by Li, Ma and Ning [26], but it does not support updating colors. The incremental variant of our oracle allows merging colors, and has only slightly higher running time.

**Online Steiner tree.**

There has been an increasing interest in the online Steiner tree and the related online MST problem in recent years, which started with a paper by Megow et al. [27]. They showed that in the incremental case one can maintain an approximate online MST in $\mathcal{O}[S]$ (and consequently an approximate Steiner tree) with only a constant number of changes to the tree per terminal insertion (in amortized sense), which

| Setting                          | apx.       | update time                     | preprocessing time                        |
|----------------------------------|------------|---------------------------------|-------------------------------------------|
| general, fully dynamic           | $6 + \varepsilon$ | $\tilde{O}(|S| \sqrt{m} \log D)$ | $\tilde{O}(|S| \sqrt{(m + n \log D)})$ |
| general, incremental            | $6 + \varepsilon$ | $\tilde{O}(|S| \sqrt{m})$      | $\tilde{O}(|S| \sqrt{m})$               |
| general, decremental            | $6 + \varepsilon$ | $\tilde{O}(|S| \sqrt{m})$      | $\tilde{O}(|S| \sqrt{m})$               |
| planar, fully dynamic            | $2 + \varepsilon$ | $\tilde{O}(|S| \sqrt{m} \log D)$ | $\tilde{O}(n \log D)$                   |
| planar, incremental             | $2 + \varepsilon$ | $\tilde{O}(\log^2 n \log D)$  | $\tilde{O}(n \log D)$                   |

Table 1: Summary of our algorithms for maintaining Steiner tree in general and planar graphs. For an input graph $G$, $n = |V(G)|$, $m = |E(G)|$, and $D$ is the stretch of the metric induced by $G$. The dependence of the running times on $\varepsilon^{-1}$ is polynomial. The update times are amortized, some of them are also in expectation.
resolved a long standing open problem posed in [25]. The result of [27] was improved to worst-case constant by Gu, Gupta and Kumar [19]. Then, Gupta and Kumar [20] have shown that constant worst-case number of changes is sufficient in the decremental case. Their paper also shows a fully dynamic algorithm that performs constant number of changes in amortized sense, but in a slightly different model. In [20], a newly added terminal is treated as a new vertex in the graph, with new distances given only to all currently active (not yet deleted) terminals; the remaining distances are assumed implicitly by the triangle inequality. However, in the classical Imase-Waxman model, the entire algorithm runs on a fixed host graph that is given at the beginning, and terminals are only activated and deactivated. After a terminal is deleted, it may still be used in the maintained tree as a Steiner node. In particular, in our algorithms it is crucial that the newly added terminal may be connected directly to a Steiner node. This is not allowed in [20], so it seems that their analysis cannot be used in the Imase-Waxman model that is studied here.

While the online algorithms for Steiner tree have been studied extensively, our paper is the first one to show that they can be turned into algorithms with low running time. Moreover, in the decremental case and fully dynamic case we are the first to show $(2 + \varepsilon)$-approximate algorithms.

2. PRELIMINARIES

Let $G = (V, E, d_G)$ be a graph. For $u, v \in V$, $\delta_G(u, v)$ denotes the distance between $u$ and $v$ in $G$. We define $G^\downarrow = (V, \{v\}, d_G)$ to be the metric closure of $G$, i.e., for $u, v \in V$, $d_G^\downarrow(u, v) = \delta_G(u, v)$. For a graph $G$ and a set $S$ of vertices, let $MST(G)$ stand for the minimum spanning tree in $G$, and let $ST(G, S)$ be an optimal Steiner tree in $G$ that spans $S$. In each of our algorithms the ultimate goal is to maintain a good approximation of $MST(G[S])$, which is a $2$-approximation of $ST(G, S)$.

In our construction we use an algorithm that dynamically maintains a minimum spanning forest of a graph, subject to edge insertions and deletions. We call it the dynamic MSF algorithm.

**Theorem 2.1** ([23]). There exists a fully dynamic MSF algorithm, that for a graph on $n$ vertices supports in edge additions and removals in $O(m \log^2 n)$ total time.

While the online algorithms assume that $G$ is given for granted, the dynamic algorithms in order to be efficient view the graph via a distance oracle. Hence, they see only approximate distances, modeled as a complete graph $GD$ with edge weights corresponding to the oracle’s answers. The function of $GD$ for dynamic algorithms is the same as the function of $G^\downarrow$ for online algorithms. We remark, however, that the fact that the distances are approximate (in particular, $GD$ satisfies the triangle inequality only with some slack) causes some significant problems in the formal analysis. We omit this layer in this extended abstract and smoothly switch between $G^\downarrow$ and $GD$ when needed.

The oracles used in our work are abstracted as a generic distance oracle, summarized in Section 5. However, for the sake of clarity, in this paper (in Section 4) instead of the generic oracle we rely on a slightly modified version of an approximate distance oracle due to Thorup and Zwick [34], later referred to as TZ oracle. Given a weighted graph $G = (V, E, d_G)$, the oracle is constructed by sampling a random subset $A \subseteq V$ of size $\sqrt{n}$. For a $v \in V$, we define $p(v)$ to be the vertex in $A$ that is nearest to $v$. Moreover, a piece of a vertex $v$, denoted $B(v)$ ([34] uses the term bunch) is the set of vertices that are closer to $v$ than $p(v)$. During initialization, for every $v \in V$ and $w \in B(v)$ we compute the distance between $v$ and $w$. Moreover, we compute distances from every $v \in A$ to every vertex in the graph.

The piece distance between $u$ and $w$ is $\delta(u, w)$ if $u \in B(w)$ or $w \in B(u)$ or $\infty$ otherwise. We say that $w$ is piece-visible from $u$ if the piece distance from $u$ to $w$ is finite. On the other hand, the portal distance between $u$ and $w$ is $\min_{p \in A} \delta_G(u, p) + \delta_G(p, w)$. Note that we precompute all piece and portal distances. As shown in [34] the minimum of the piece and portal distance yields a $3$-approximate distance. In this extended abstract by GD we denote a complete graph on the vertex set $V$ with edge lengths being the distances returned by the TZ oracle.

3. ONLINE ALGORITHMS

We start with presenting our online algorithms for the Steiner tree problem. In this setting the goal is to maintain an approximate Steiner tree and make few changes to it after a terminal is added or deleted; in our dynamic setting a small number of changes will help us obtain small running time.

In this section we focus on the following problem: given a metric closure $G$ and its dynamically-changing subset of vertices $S$, maintain a tree that spans $S$, whose weight is $(1 + \varepsilon)$-approximation of the weight of $MST(G[S])$. As mentioned in Section 2, $MST(G[S])$ gives a $2$-approximation of $ST(G, S)$. In order to maintain good approximation ratio of $MST(G[S])$ it suffices to apply the following rules to the maintained tree (see Fig. 1): (a) as long as we can replace an edge $e'$ in the tree with another edge $e$ of significantly (by a factor of $(1 + \varepsilon)$) lower cost, proceed with the replacements; we call such a replacement $\varepsilon$-efficient; (b) if a large-degree vertex is removed from $S$, keep it in the tree as a nonterminal (until its degree drops below some fixed threshold).

Arguments in [27] show that if a tree $T$ does not admit any $\varepsilon$-efficient replacement that replaces an edge of cost at least $\varepsilon d_{G^\downarrow}(T)/n$ (called henceforth $\varepsilon$-heavy), then $T$ is an $(1 + O(\varepsilon))$-approximation of $MST(G[V(T)])$. We combine this idea with the approach of Imase and Waxman [25], who proved that keeping nonterminals of degree more than $2$ incurs only a loss of $2$ in the approximation ratio. A novel idea in our paper is a generalization of this rule to arbitrary degree threshold, which allows to reduce this loss from $2$ to $1 + O(\varepsilon)$. In the end, we show that if a tree $T$ (a) spans the set of terminals and nonterminals, but the nonterminals have degree larger than $1 + [\varepsilon^{-1}]$; (b) is an approximate MST (does not admit $(1 + \varepsilon)$-efficient replacements), then $T$ is a $(1 + O(\varepsilon))$-approximation of the MST on the set of terminals.

**Lemma 3.1.** Let $\varepsilon \geq 0$ be a constant, let $\eta = 1 + [\varepsilon^{-1}]$, let $G = (V, \{\varepsilon\}, d_G)$ be a complete weighted graph and $S \subseteq V$. Let $T_{MST} = MST(G[S])$ be an MST of $S$ and let $T$ be a tree in $G$ that spans $S \cup N$ and does not admit $(1 + \varepsilon)$-efficient replacements. Furthermore, assume that each vertex of $V(T) \cap N$ is of degree larger than $\eta$ in $T$. Then $d_G(T) \leq (1 + O(\varepsilon))d_{G^\downarrow}(T_{MST})$. 
Deletion step and decremental scheme.

In the decremental scheme, the main idea is to maintain the MST on the terminal set, but to postpone the deletion of terminals that are of degree above some fixed threshold $\eta = 1 - \lfloor \varepsilon^{-1} \rfloor$ in the spanning tree. If a degree of some non-terminal $v$ drops to or below $\eta$, we delete $v$ and try to optimally reconnect the remaining pieces of the maintained tree (see Fig. 2a). Since $\eta \in O(\varepsilon^{-1})$, such reconnection requires only $O(\varepsilon^{-1})$ new edges, and hence the number of changes applied to the tree is bounded by $O(\varepsilon^{-1})$ per update.

Addition step and incremental scheme.

In the incremental algorithm we maintain a tree $T$ spanning the terminal set. When a new terminal $v$ arrives, we:

1. add an edge to $T$ connecting $v$ with $V(T)$ of cost (approximately) $\min_{u \in V(T)} d_T(uv)$;
2. apply a sequence of $\varepsilon$-efficient replacement pairs, removing from the tree always an edge of maximum possible cost;
3. after all the replacements, we require that there does not exist an $\varepsilon$-efficient $\varepsilon$-heavy replacement pair in $T$ with the new edge incident to $v$.

See Fig. 2b for illustration. Following a nontrivial analysis in [20] we can prove that the total number of replacements is $O(r \varepsilon^{-1})$, where $r$ is the number of added terminals.

We emphasize here one important improvement over the previous online algorithms that is crucial for efficient implementation: we only care about all replacements with new edge incident to $v$. We show that approximation guarantees still hold in this case, as long as we always remove an edge of maximum possible cost in a replacement.

Fully dynamic scheme.

In order to obtain a scheme for a fully dynamic algorithm, we merge the ideas of two previous sections. In a deletion step, we behave in exactly the same manner as in the decremental scheme. In an addition step, we perform similarly as in the incremental scheme, but there are two significant differences. First, we do not have the guarantee that the cost of the tree will not decrease much in the future, so we cannot stop replacing edges at some cost threshold: although the low cost edges may contribute only a little to the weight of the tree, this may change after many terminals are removed and the weight of the tree drops. Second, we need to watch out for non-terminal vertices whose degree may drop to the threshold $\eta$ as a consequence of a replacement. Again as in the incremental scheme, we show that in the addition step it suffices to care only about replacement pairs with the new edge incident to the newly added vertex.

In addition to that, as mentioned before, the analysis of [20] does not help us anymore to bound the number of modifications to the tree performed while adding the terminals. We use a different, simpler analysis (similar to the one given in [27]) to show, that the number of changes to the tree that are a result of terminal addition is bounded by $O(r \varepsilon^{-1} \log D)$ where $r$ is the number of additions. The log $D$ factor comes from the fact, that we keep replacing low cost edges. Terminal removals are still handled within $O(\varepsilon^{-1})$ changes per removal. To sum up, we obtain an upper bound of $O(\varepsilon^{-1} \log D)$ changes to the tree per update in the fully dynamic case.

Finally we remark, that the online algorithms can also be applied to graph $GD$, which approximates $\overline{G}$ within a constant factor. The asymptotic bounds for the number of changes to the tree remain the same, and the Lemma 3.1 can be applied to $GD$ instead of $\overline{G}$. We use that fact in the next sections, where we implement the online algorithms using the TZ distance oracle, which produces such an approximate metric $GD$.

4. DYNAMIC ALGORITHMS

In this section we show how to turn the online algorithms into efficient dynamic algorithms.

Decremental algorithm.

In the decremental scenario we maintain an approximate Steiner tree as terminals are deleted. For a fixed $\varepsilon > 0$ we plan to maintain a $(1 + \varepsilon)$-approximation of $\text{MST}(GD[S])$.

Following the decremental scheme, we set $\eta = 1 + \lfloor \varepsilon^{-1} \rfloor = O(\varepsilon^{-1})$. We start with the tree $T = \text{MST}(GD[S_1])$, where $S_1$ is the initial set of terminals. When a vertex $v$ of degree more than $\eta$ is removed from the terminal set $S$, we mark it as a nonterminal vertex, but do nothing more. Otherwise, we need to remove $v$ from $T$ and reconnect the connected components that emerge into a new tree. The main challenge lies in finding the reconnecting edges efficiently.

In order to do that, we use the dynamic MSF algorithm (see Theorem 2.1) on an auxiliary graph $H$ that we maintain.
Our goal is to maintain a tree $T$ which is an MST spanning terminals and nonterminals of high degree which used to be terminals. Hence, we assure that $H$ is a subgraph of $GD$ that contains all edges of this MST (for simplicity, we assume here that the MST is unique). Moreover, the only nonempty (i.e., containing edges) connected component of $H$ is composed exactly of terminals and the aforementioned nonterminals.

As a result, the dynamic MSF algorithm maintains tree $T$.

The simplest approach to maintaining $H$ would be to add, for every two $u, w \in S$, an edge $uw$ of length $d_{GD}(u, w)$. This, however, would obviously be inefficient, i.e., work in linear time. Instead of that, we use the structure of the TZ distance oracle. For every two $u, w \in V(T)$ which are mutually piece-visible we add to $H$ an edge $uw$ of length being equal to the piece distance.

Moreover, we will be adding some edges corresponding to portal distances, but they will be chosen in a careful way to ensure that the number of such edges is low.

We initialize the edge set of $H$ to be the set of all edges of the initial tree $T$ spanning $S_1$ and all edges between mutually piece-visible terminals. Moreover, for every portal $p \in A$, we maintain an ET-tree (Euler Tour tree, see [32]) $ET_p$, such that $V(ET_p) = V(T)$ and $ET_p$ is isomorphic to $T$. The key of a node $w \in ET_p$ is $d_{GD}(w, p)$. Note that all the necessary distances are precomputed during the initialization of the TZ oracle.

The main challenge is what happens if a vertex $v$ of degree $s \leq \eta$ is removed. Then the tree $T$ decomposes into trees $T_1, \ldots, T_k$ that we need to reconnect with a set of edges of minimum total cost. We rely on the dynamic MSF algorithm here and make sure that $H$ contains a superset of the desired edges. First of all, $H$ surely contains all edges corresponding to piece distances, as this is maintained as an invariant of our algorithm. Thus, we only need to assure that $H$ also contains the necessary edges corresponding to portal distances. It is easy to observe that it suffices to add to $H$ the MST of edges corresponding to portal distances between trees $T_i$. In order to do that, we remove $v$ from every $ET_p$ for $p \in A$, and cut the ET-trees in the same way as $T$. This allows us to compute for every tree $T_i$ and every portal $p \in A$ the minimum distance between $p$ and a vertex of $T_i$. Consequently, we may easily obtain portal distances between the trees $T_i$.

More formally, we consider a complete graph $G_c$ over trees $T_i$, in which the edge length between two trees is the portal distance between the nearest vertices in these trees. We add to $H$ only edges corresponding to $MST(G_c)$. This MST can be simply computed by generating the graph $G_c$ explicitly, using the fact that $\eta$ is constant and the total number of portals is relatively small. That is, we simply go through all portals and for each portal we generate edges corresponding to all portal distances that use this portal.

The main factors that influence the running time are: the number of pairs of mutually piece-visible terminals (all such edges are initially added to $H$), the time for computing the reconnecting edges corresponding to portal distances, and the time for updating the ET-trees, when a vertex of degree at most $\eta$ is removed. Note that $r$ delete operations may cause at most $r$ vertices to be removed. In the full version of the paper we do not generate all edges of $G_c$, but only $O(\varepsilon^{-1})$ of them per portal. So the update time is dominated by the time needed to link and cut the ET-trees. In general graphs we obtain a $(6 + \varepsilon)$-approximate algorithm handling updates in $O(\varepsilon^{-1}\sqrt{n}\log n)$ expected amortized time.

Figure 2: Panel (a) illustrates the deletion step: when a vertex $v$ is deleted, the tree splits into subtrees $T_i$, and between every pair of subtrees the shortest reconnecting edge $e_{i,j}$ is found (depicted as blue thick edge). Then, we find the set of reconnecting edges by computing a minimum spanning tree of the auxiliary graph $G_c$ with vertex set $\{T_1, T_2, \ldots, T_k\}$ and edges $e_{i,j}$ connecting $T_i$ and $T_j$. Panel (b) illustrates the addition step: when a new terminal $t_i$ is added, it is first connected to the closest terminal with an edge $e_0$, and then $e'$ is replaced for $e$. 
Incremental algorithm.

In this section, our goal is to approximate the Steiner tree as the terminals are added. Denote the terminals that are added by \( t_1, t_2, t_3, \ldots \). Let \( S_i = \{ t_1, \ldots, t_i \} \) and let \( T_i \) denote the tree maintained by the algorithm after adding \( i \) terminals. We maintain a \((1 + \varepsilon)\)-approximation of MST in the subgraph \( GD[S_i] \) of \( GD \) induced by \( S_i \).

Fix a constant \( \varepsilon > 0 \). By rounding up edge lengths, we assume that the edge lengths in \( GD \) are powers of \( 1 + \varepsilon \). Technically speaking, we define the level of an edge \( uv \), as \( 1 \nu l(uv) := \log_{1+\varepsilon} d_{GD}(uv) \). Assume that all edge lengths of \( GD \) belong to the interval \([1, D]\) and \( h = \lfloor \log_{1+\varepsilon} D \rfloor \). The algorithm again builds up on the structure of TZ oracle. It represents the current tree \( T_i \) by maintaining for every \( j \in \{1, \ldots, h\} \) (\( j \) is referred to as a layer) and every portal \( p \in A \) a forest \( ET_{j,p} \) of ET-trees. For a fixed \( j \), the forests \( ET_{j,p} \) (for all \( p \in A \)) have the same topology. They are obtained from \( T \) by removing edges of level greater than \( j \) (see Fig. 3). The key of a vertex \( w \in ET_{j,p} \) is \( \delta_C(w, p) \). Note that the number of the forests is \( O(e^{-\frac{1}{2}\sqrt{n}\log D}) \).

Moreover, for every layer \( j \) and every portal \( p \in A \), we maintain a binomial heap \( H_{j,p} \). It contains, for every tree \( R \) in every forest \( ET_{j,p} \), the distance between \( p \) and \( R \). \( H_{j,p} \) can be maintained under link and cut operations on forests \( ET_{j,p} \). Observe that the total size of \( H_{j,p} \) is upper-bounded by the total size of all forests \( ET_{j,p} \). These data structures can be maintained under link and cut operations to the tree \( T \). They also allow to perform the following.

1. For a vertex \( v \in V \setminus V(T) \), we can find the (approximately) nearest vertex \( u \in V(T) \). This can be achieved by computing piece and portal distances from \( u \) to \( T \). Piece distance can be obtained easily, by simply iterating through all vertices that are piece-visible from \( v \). To obtain portal distance, we check distance to \( T \) via every possible portal.
2. For a portal \( p \), we consider the sum of \( \delta_C(v, p) \) and the minimum key in the tree \( ET_{i,p} \).
3. Consider a vertex \( v \in V(T) \). We may find the (approximately) nearest vertex in \( ET_{j,p} \) (other than the tree containing \( v \)). First, we first find the nearest tree \( u \) r.t. piece distance by checking the piece distances to all vertices that are piece-visible from \( v \). Then, for each portal \( p \in A \), we consider the sum of \( \delta_C(v, p) \) and the distance between \( p \) and the tree which is nearest to \( p \). To do this, we find the minimum element in the heap \( H_{j,p} \) (or the second smallest, if the minimum one is the tree containing \( v \)).

The process of adding a terminal \( t_i \) consists of two steps. First, we find the shortest edge in \( GD \) connecting \( t_i \) to any of \( t_1, \ldots, t_{i-1} \) and add this edge to \( T_{i-1} \) to obtain tree \( T_i \). This can be achieved by using the first of the above operations. Then, we apply a sequence of \( \varepsilon \)-efficient replacements to \( T_i \) in order to decrease its weight. Recall that we are allowed to consider only the replacements in which the new edge is incident to the newly added terminal \( t_i \).

In order to find the replacements, we use the trees \( ET_{j,p} \). For every \( p \) the forest topology of \( ET_{j,p} \) is the same, for the sake of argument let us denote it by \( F_j \). Fix a layer number \( j \). We want to find a replacement pair \((e, e')\) such that \( 1 \nu l(e') > j \) and \( 1 \nu l(e) \leq j \). Denote by \( C \) the tree of \( F_j \) that contains \( t_i \). By definition of \( C \), it consists of edges of level at most \( j \) and the path in \( T_i \) from \( t_i \) to every \( t \not\in C \) contains an edge of level \( > j \). We find the vertex \( t \not\in C \) which is the nearest to \( t_i \), using the second of the above two operations. Assume that we find a vertex \( t' \). If \( 1 \nu l(t, t') \leq j \), we have found a replacement pair. Let \( e' \) be the heaviest edge on the path from \( t_i \) to \( t' \) in \( T_i \) and \( e = t_it' \). Clearly, we can replace \( e' \) with \( e \), and since the weight of \( e' \) is lower than the weight of \( e' \) and the weights differ at least by a factor of \( 1 + \varepsilon \), this replacement is \( \varepsilon \)-efficient.

The running time of the algorithm described above depends on \( \log D \). In order to circumvent this dependency, we observe that we do not need to replace edges whose length is at most \( \varepsilon/(2n) \) of the weight of the current tree, as their total weight is negligible. This allows us to limit the number of levels that we consider to roughly \( O(e^{-1}\log n) \).

One of the factors influencing the running time is the number of replacements: \( O(r\varepsilon^{-r}) \) in the course of \( r \) terminal additions. For each replacement, we issue a constant number of operations on each \( ET_{j,p} \), so the running time is, roughly speaking, a product of the number of replacements, the number of ET-trees and the running time of a single ET-tree operation, what amounts to \( O(r\varepsilon^{-r}2\sqrt{n}) \). Nevertheless, using the generic incremental oracle for each layer instead of the construction presented here, in the end we obtain \((6 + \varepsilon)\)-approximate algorithm for general graphs handling insertions in \( O(e^{-1}\sqrt{n}) \) expected amortized time per operation.

Fully dynamic algorithm.

The fully dynamic algorithm is obtained by appropriately merging the ideas of the decremental and incremental algorithms. In fact we maintain the invariants of both of these algorithms.

Fix \( \varepsilon > 0 \) and set \( \eta = 1 + [\varepsilon^{-1}] \). We round up edge lengths, so that all edges’ lengths in \( GD \) are powers of \( 1 + \varepsilon \). We maintain a tree \( T \) that spans terminals and nonterminals of degree more than \( \eta \). This is done by using dynamic MSF algorithm on a graph \( H \) that we update. The tree \( T \) is an MST of the set of terminals and high-degree nonterminal vertices, and, since the length of every edge is a power of \( 1 + \varepsilon \), this is equivalent to the fact that it does not admit any \( \varepsilon \)-efficient replacements.

Assume that the edge lengths in \( GD \) are in \([1, D]\). Let \( h = \lfloor \log_{1+\varepsilon} D \rfloor \). As in the incremental algorithm, we maintain a collection of ET-trees \( ET_{j,p} \). Recall that the level of an edge \( uv \) is \( 1 \nu l(uv) := \log_{1+\varepsilon} d_{GD}(uv) \). For a fixed \( j \), ET-trees \( ET_{j,p} \) are trees obtained from \( T \) by removing edges of level greater than \( j \). In particular, trees \( ET_{h,p} \) are equal to \( T \).

When a terminal \( v \) is deleted, we use exactly the same procedure as in the decremental algorithm. On the other hand, if a vertex \( v \) is added, we use a procedure similar to the incremental algorithm. Namely, we connect \( v \) to \( T \) using the shortest edge in \( GD \) and then apply all occurring \( \varepsilon \)-replacements. In order to find all replacements, we need to consider replacements at all possible \( O(e^{-1}\log D) \) levels. When we detect that an edge \( e' \) can be replaced with \( e \), we add edge \( e \) to \( H \). Since every replacement that we find is \( \varepsilon \)-efficient, the dynamic MSF algorithm will surely update the maintained tree accordingly. Note that \( e' \) may be left in \( H \). The ET-trees are updated to reflect the changes done by the algorithm.

After all replacements are made, the tree \( T \) is an approximate MST of the set of terminals and nonterminal vertices of degree more than \( \eta \). The last step is to add to \( H \) all edges \( vz \) for all vertices \( v \) which are piece-visible from \( v \), as this is the invariant of the decremental algorithm.
5. ORACLE ABSTRACTION

As announced in the introduction, our construction is modular. We use distance oracles based on [34, 33], abstracted under a notion of a generic oracle. We then show how to extend a generic oracle into a vertex-color distance oracle. In this section we briefly describe these oracles.

Generic distance oracles.

Let \( G = (V, E, d_G) \) be a weighted graph and \( \alpha \geq 1 \). An \( \alpha \)-approximate generic oracle for \( G \) associates with every \( v \in V \):

- a set of portals, denoted by \( portals(v) \), which is a subset of \( V \),
- and a family of pieces, denoted by \( pieces(v) \), were each element (referred to as piece) piece is a weighted planar graph on a subset of \( V \).

The piece distance from \( v \) to \( w \) is defined as \( R_{v,w} = \min_{p \in pieces(v)} \delta_p(v,w) \) (note that if none of \( pieces(v) \) contains \( w \), then \( R_{v,w} = +\infty \)). The oracle additionally stores, for each \( v \in V \) and \( p \in portals(v) \), an approximate distance \( D_{v,p} \geq \delta_G(p,v) \). For every pair \( v, w \in V \) we require that \( R_{v,w} = R_{w,v} \), and either \( R_{v,w} = \delta_G(v,w) \) or there is a portal \( p \in portals(v) \cap portals(w) \), such that there is a walk from \( v \) to \( w \) of length at most \( \alpha \cdot \delta_G(v,w) \) that goes through \( p \), and \( D_{v,p} + D_{w,p} \leq \alpha \delta_G(v,w) \). The portal distance between \( v \) and \( w \) is \( \min_{p \in portals(v) \cap portals(w)} D_{v,p} + D_{w,p} \) (see Fig 4).

Vertex-color distance oracles.

Each vertex-color distance oracle maintains a partition of \( V \) into sets (colors) \( C_1, \ldots, C_k \). Each color can be active or not. The oracle allows us, for instance, to find the distance from a given vertex to the nearest vertex of a given color. In the incremental variant, the only update operation is merging two colors. That is, we support the following set of operations:

- **distance** \((v, i)\) – compute the approximate distance between \( v \) and the nearest to \( v \) vertex of color \( i \),
- **nearest** \((v, j)\) – compute the approximate distance between \( v \) and the \( j \)-th nearest to \( v \) active color (for a constant \( j \)).
that a piece squares. We have that portals(v) = \{b, j\} for all vertices v. In panel (b) there are two pieces. In this example we assume that a piece \(r_i \in \text{pieces}(v)\) if \(v \in V(r_i)\). For every \(u, v \in V\), the minimum of piece and portal distances between \(u\) and \(v\) is at most 2\(\Delta(u, v)\). For example, the piece distance between \(e\) and \(g\) is infinite, but the portal distance is 4, which is twice as much as the exact distance.

- **activate\((i)\)** – activate set (color) \(C_i\),
- **merge\((i, j)\)** – merge two different active sets \(C_i\) and \(C_j\) into one active set \(C_i\), where \(l \in \{i, j\}\).

The fully dynamic variant also supports splitting sets \(C_i\), but only in a restricted way. The oracle associates with every set \(C_i\) a tree \(T_i\) that spans \(C_i\). Trees \(T_i\) may be arbitrary trees spanning vertices of \(C_i\), and their edges are not necessarily present in \(G\). Splitting a set is achieved by specifying an edge \(e\) in \(T_i\). The edge \(e\) is removed from \(T_i\) and the two connected components that are created specify how the set should be split. Formally speaking, the fully dynamic oracle supports, apart from **distance**, **nearest**, and **activate**, also the operation **deactivate\((i)\)** that deactivates set (color) \(C_i\), as well as the following operations:

- **merge\((i, j, u, v)\)** – merge active sets \(C_i\) and \(C_j\) associated with trees \(T_i\) and \(T_j\) into an active color set \(C_l\), \(l \in \{i, j\}\), and associate it with \(T_l := T_i \cup T_j \cup \{uv\}\),
- **split\((l, u, v)\)** – if \(uv\) is an edge of a spanning tree \(T_L\) of an active color set \(C_l\), split \(C_l\) into active color sets \(C_i\) and \(C_j\) associated with the two connected components of \(T_l \setminus \{uv\}\).

In Section 4, for the sake of simplicity, the TZ oracle was used to implement our algorithms. In the full version of the paper we use instances of some generic oracle instead. For example, to implement the incremental algorithm, we set up a vertex-color oracle for every layer. Recall, that every layer \(j\) is associated with a forest \(F_j\), which is exactly the maintained tree \(T\) without the edges of levels higher than \(j\). The color sets of the oracle at layer \(j\) are the connected components (trees) of \(F_j\). In order to find the new reconnecting edge in \(G_D\) or the replacements at some layer \(j\), we query the appropriate oracle, and then possibly merge some colors due to the executed replacement. This can be done using an incremental oracle as the colors are only merged. For the decremental scenario we employ one instance of a fully dynamic oracle. The oracle has a single active color corresponding to vertices of \(V(T)\). When the tree \(T\) falls apart due to removal of a vertex \(v\) of constant degree, this one color class is split by cutting the edges incident to \(v\).

We then need to find the portal distances between the components, and our fully dynamic oracle can do it for us.

Employing the generic oracles gives us flexibility to choose the most appropriate oracle variant for a particular scenario and a particular input graph. The different running times and approximation ratios in Table 1 come mostly from the fact that different variants of the oracles are used.

6. ACKNOWLEDGMENTS

Jakub Ocieja, Marcin Pilipczuk, Piotr Sankowski and Anna Zych are partially supported by ERC grant PAAI no. 259515. The research of Marcin Pilipczuk has received funding from the European Research Council under the European Union’s Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement n. 267959. Piotr Sankowski and Anna Zych are partially supported by Polish National Science Centre grant UMO-2014/13/B/ST6/01811. Piotr Sankowski is partially supported by Foundation for Polish Science. Jakub Łącki is a recipient of the Google Europe Fellowship in Graph Algorithms, and this research is supported in part by this Google Fellowship.

7. REFERENCES

[1] I. Abraham, D. Malkhi, and D. Ratajczak. Compact multicast routing. In 23rd International Symposium on Distributed Computing (DISC 2009). Springer Verlag, September 2009.
[2] E. Aharoni and R. Cohen. Restricted dynamic steiner trees for scalable multicast in datagram networks. In INFOCOM ’97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution, Proceedings IEEE, volume 2, pages 876–883 vol.2, Apr 1997.
[3] E. Anshelevich, A. Dasgupta, J. Kleinberg, E. Tardos, T. Wexler, and T. Roughgarden. The price of stability for network design with fair cost allocation. In In FOCS, pages 295–304, 2004.
[4] S. Arora. Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems. J. ACM, 45(5):753–782, 1998.
