Near-Linear Algorithms for Visibility Graphs over a 1.5-Dimensional Terrain

Matthew J. Katz
Department of Computer Science, Ben-Gurion University of the Negev, Beer Sheva, Israel

Rachel Saban
Department of Computer Science, Ben-Gurion University of the Negev, Beer Sheva, Israel

Micha Sharir
School of Computer Science, Tel Aviv University, Israel

Abstract

We present several near-linear algorithms for problems involving visibility over a 1.5-dimensional terrain. Concretely, we have a 1.5-dimensional terrain T, i.e., a bounded x-monotone polygonal path in the plane, with n vertices, and a set P of m points that lie on or above T. The visibility graph $VG(P,T)$ is the graph with P as its vertex set and $\{(p,q) \mid p \text{ and } q \text{ are visible to each other}\}$ as its edge set. We present algorithms that perform BFS and DFS on $VG(P,T)$, which run in $O(n \log n + m \log^3(m+n))$ time.

We also consider three optimization problems, in which P is a set of points on T, and we erect a vertical tower of height h at each $p \in P$. In the first problem, called the *reverse shortest path problem*, we are given two points $s, t \in P$, and an integer k, and wish to find the smallest height h^* for which $VG(P(h^*),T)$ contains a path from s to t of at most k edges, where $P(h^*)$ is the set of the tips of the towers of height h^* erected at the points of P. In the second problem we wish to find the smallest height h^* for which $VG(P(h^*),T)$ contains a cycle, and in the third problem we wish to find the smallest height h^* for which $VG(P(h^*),T)$ is nonempty; we refer to that problem as "Seeing the most without being seen". We present algorithms for the first two problems that run in $O^*((m+n)^{6/5})$ time, where the $O^*(\cdot)$ notation hides subpolynomial factors. The third problem can be solved by a faster algorithm, which runs in $O((n+m)\log^3(m+n))$ time.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases 1.5-dimensional terrain, visibility, visibility graph, reverse shortest path, parametric search, shrink-and-bifurcate, range searching

Digital Object Identifier 10.4230/LIPIcs.ESA.2024.77

Funding *Matthew J. Katz*: Partially supported by Israel Science Foundation Grant 495/23.
Rachel Saban: Partially supported by the Lynne and William Frankel Center for Computer Science and by Israel Science Foundation Grant 495/23.
Micha Sharir: Partially supported by Israel Science Foundation Grant 495/23.

1 Introduction

Let $T = (v_1, \ldots, v_n)$ be a *1.5-dimensional terrain*, that is, a bounded x-monotone polygonal line with $v_1, \ldots, v_n$ as vertices in this left-to-right order. For any two points a, b on or above T, we say that a and b *see each other* if every point on the line segment ab is either on or above T. Let P be a set of m points that lie on or above T. The *visibility graph* $VG(P,T)$ is defined as the graph whose set of vertices is P, and whose edges are all the pairs (u,v), for $u, v \in P$, such that u and v are mutually visible; see Figure 1(a).

In this paper we consider several problems on visibility graphs. These problems are Breadth-First Search (BFS), Depth-First Search (DFS), and graph emptiness. Since $VG(P,T)$ can have a quadratic number of edges, we need to replace the standard algorithms, whose running time depends on the number of edges, by algorithms whose performance only depends

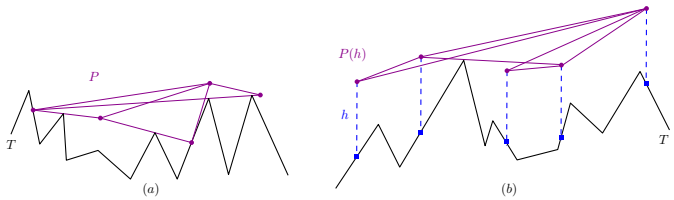

Figure 1 (a) The visibility graph $VG(P,T)$. (b) The visibility graph $VG(P(h),T)$ for a set of towers of height h.

on the number m of graph vertices, and on n. Somewhat surprisingly, we obtain algorithms that run in near-linear time. Specifically, in Section 2 we present an algorithm for running BFS on $VG(P,T)$, which runs in $O(n\log n+m\log^3(m+n))$ time. A fairly routine adaptation of this algorithm, given in Section 4, yields an algorithm for running DFS on $VG(P,T)$, with the same running time bound.

We also consider several optimization problems on visibility graphs, each related to one of the preceding problems. For this, we assume that P is a set of m points on T. For a real parameter $h>0$, we erect a *tower* of height h at each $p\in P$, which is a vertical line segment of length h whose bottom endpoint is p. We denote the tip of the tower as $p(h)=p+(0,h)$, and denote by $P(h)$ the set of all tower tips. We then consider the graph $VG_h(P,T)=VG(P(h),T)$ (see Figure 1(b)), and study various optimization problems associated with this graph, in which we want to find the smallest height h^* for which some graph property holds for $VG_{h^*}(P,T)$. Specifically, we consider the following three problems.

Reverse shortest path: For two given points s, $t\in P$ and an integer k, find the smallest height h^* for which $VG_{h^*}(P,T)$ contains a path from s to t of at most k edges.

Graph acyclicity: Find the smallest height h^* for which $VG_{h^*}(P,T)$ contains a cycle.

Graph emptiness, or seeing the most without being seen: Find the smallest height h^* for which $VG_{h^*}(P,T)$ is nonempty (i.e., there exist two mutually visible tower tips).

The first and third problem were introduced by Agarwal et al. [3] and Katz and Sharir [18], respectively. The solution of Agarwal et al. runs in $O^*(n+m^{4/3})$ time, and uses an implementation of BFS with the same running time as the driving decision procedure. Katz and Sharir [18] claimed to have obtained an $O^*(n^{6/5})$-time solution, using the shrink-and-bifurcate strategy (see below), but their solution overlooks an issue that may arise when using an off-the-shelf decision algorithm in conjunction with this strategy.

The first two problems have associated *decision procedures*, in which we specify h, and ask whether $VG_h(P,T)$ has the desired property. Concretely, for the reverse shortest path problem, we test whether $VG_h(P,T)$ contains a path of length at most k between s and t, and for the graph acyclicity problem, we test whether $VG_h(P,T)$ contains a cycle. For the graph emptiness problem, we test whether $VG_h(P,T)$ is empty (has no edges). The first decision procedure uses the BFS algorithm, the second uses the DFS algorithm, and the third procedure, which we present in Section 5, uses a different approach, and also runs in near-linear time.

The first two optimization algorithms use a variant of parametric search, originally developed by Ben Avraham et al. [6] and recently used by the authors in [16], which employs a strategy known as *shrink-and-bifurcate*. It is applicable in setups where we have an efficient decision procedure which is not parallelizable, or at least we do not know how to make it parallelizable in an efficient manner to fit into the standard parametric searching mold. We review this technique, adapted to our context, in some detail in Sections 3.2 and 3.3. Using

this technique, we obtain, in Sections 2 and 4, optimization procedures for the first two problems that run in $O^*((m+n)^{6/5})$ time, where the $O^*(\cdot)$ notation hides subpolynomial factors. The third problem has a special structure, leading to a faster $O((n+m)\log^3(m+n))$ time algorithm; see Section 5.

Related work. Much effort has been devoted to identifying families of graphs in which fundamental graph tasks, such as BFS, matching, etc., can be implemented more efficiently than in general graphs. One such task is BFS, which takes $O(|V|+|E|)$ time in general graphs, where V and E are, respectively, the sets of vertices and edges of the graph. The challenge in this case is to do it more efficiently, in time that depends (ideally, nearly linearly) only on $n = |V|$, in families of geometric graphs in which the number of edges can be quadratic in n.

For unit-disk graphs, for example, Cabello and Jejčič [10] presented an $O(n \log n)$ implementation of BFS, and subsequently Chan and Skrepetos [11] presented an alternative $O(n)$ implementation (after pre-sorting the points by their x- and y-coordinates). Moreover, Cabello and Jejčič [10] also describe an $O(n^{1+\varepsilon})$ implementation of Dijkstra's algorithm for weighted unit-disk graphs, which was followed by a more efficient $O(n \log^2 n)$ implementation by Wang and Xue [22]; see also [17].

For intersection graphs of arbitrary disks (or more generally, for proximity graphs of disks), Agarwal et al. [16] presented an $O(n \log^4 n)$ implementation of BFS. They also consider two natural definitions of the weighted version of proximity graphs of disks, for which they present nearly-linear implementations of Dijkstra's algorithm, in the spirit of [10].

The RSP problem, in the context of unweighted unit-disk graphs (which are often used to model wireless networks), was posed by Cabello and Jejčič [10], who observed that it can be solved conceptually easily in $O^*(n^{4/3})$ time, by running a binary search through the $O(n^2)$ inter-point distances (using an efficient distance selection algorithm). Then, Wang and Zhao [23] managed to improve this bound, obtaining an algorithm that solves the problem in $O^*(n^{5/4})$ time. The bound was further improved by Agarwal et al. [16], who presented an $O^*(n^{6/5})$ algorithm, using the shrink-and-bifurcate technique mentioned above. In the context of weighted unit-disk graphs, the situation is similar. The RSP problem can be solved easily in $O^*(n^{4/3})$ time, but Wang and Zhao [24] and Agarwal et al. [16] were able to obtain improved $O^*(n^{5/4})$-time and $O^*(n^{6/5})$-time solutions, respectively, for that version too. Finally, in the context of arbitrary disk graphs, both unweighted and weighted, Agarwal et al. [16] presented $O(n^{5/4})$-time solutions to several versions of the RSP problem.

1.5-dimensional terrains have received much attention, since they are used, e.g., to model straight hilly roads. Many of the problems studied for such terrains revolve around visibility, such as various guarding problems, see, e.g., [2, 8, 13, 14]. Visibility graphs and, in particular, the problem of computing them efficiently in various polygonal domains, have also been studied extensively, see, e.g., [7, 15, 20].

In general, a faster BFS algorithm on a graph G is easy to obtain from a *clique cover* of G, namely, a representation of the edge set E of G as the (typically edge disjoint) union of cliques (a representation as the union of bicliques is also suitable for this purpose). The resulting BFS algorithm runs in time proportional to the size of the clique cover, namely to the sum of the sizes of its vertex sets. In general, though, finding a small-size clique cover may be impossible: Agarwal et al. [1] have shown that, for visibility graphs of n pairwise disjoint segments in the plane, the smallest size of a clique cover may be $\Omega(n^2/\log^2 n)$. On the other hand, they showed that the visibility graph of the *vertex set* of any simple polygon can be represented by a clique cover of size $O(n \log^3 n)$. In our case, though, the input points are not vertices of T. Making them vertices artificially, by connecting each point of P to

T by a straight vertical segment, say, may block visibilities of some pairs. In short, it is a challenging open problem to find a clique cover of near-linear size for our setup. The best that a suitable extension of our technique seems to yield is a cover of size $O^*(n^{4/3})$.

2 BFS in visibility graphs over a terrain

Let T, P, n and m be as defined above. In this section we present a near-linear algorithm for performing BFS in the graph $VG(P,T)$, from a given start point $s \in P$. Since this graph can have a quadratic number of edges, the challenge is indeed to reduce the running time to nearly linear in $m+n$.

We use a variant of a technique due to Varadarajan [21], who applied it to a certain path simplification problem. His algorithm runs in near-$n^{4/3}$ time, but we will only use ideas related to the high-level structure of the algorithm, to obtain a much faster, near-linear algorithm. The same infrastructure will also be used in our other algorithms.

The general scheme goes as follows. As usual, we construct the BFS layer by layer; the i-th layer is denoted by L_i, starting with $L_0 = \{s\}$. Suppose we have already constructed L_i and now want to construct L_{i+1}. We maintain dynamically the set of points of P that the BFS has not reached yet; denote by P_i this set at the end of the i-th iteration.

We construct and maintain the following dynamic data structure on P_i. We query it with the points of L_i, where the goal of the queries is to find and report all the points of P_i that are visible from at least one point of L_i, and do it so that no point is reported more than once. By definition, these points comprise L_{i+1}, and we delete them from the structure, thereby obtaining P_{i+1}.

Dropping the index i for convenience, the setup is thus as follows. We have a set P of points, each lying on or above T, which we want to maintain under deletion of points, so as to be able to answer efficiently queries, each of which specifies a query point q (also lying on T or above T), and seeks to report all the points of P that are visible from q.

The procedure uses divide and conquer. It splits T at its median vertex v_μ into a left portion T_L and a right portion T_R. The set P is thereby split into a left part P_L and a right part P_R. We maintain dynamically two data structures, one for P_R and one for P_L. The goal of the former (resp., latter) structure is to answer visibility reporting queries with points q that lie on or above T_L (resp., T_R). We then continue to process recursively P_L and P_R, with respect to the corresponding parts T_L and T_R of the terrain. When we delete a point, we delete it from all the $\log n$ recursive nodes at which it participates. At the bottom of recursion, we have pieces of T that contain no vertices. If the corresponding subset of P has more than one point, they form a clique in $VG(P,T)$, which is then trivial to process.

Reporting left-right visibilities. This is the main step in the divide-and-conquer procedure. For concreteness, we consider left-to-right visibility queries, where we seek visible neighbors in P_R of query points in P_L; the other direction is handled in a fully symmetric manner. We want to maintain P_R dynamically, under deletions, in a data structure that supports the following type of queries. Each query specifies a point q that lies on or above the left portion T_L of T, and asks for reporting all the points of P_R that are visible from q. When we query with the points of P_L in the current BFS layer, we add the reported points to the next layer of the BFS, and delete them from the structure. We then repeat this step to the symmetric setup, of querying with points of P_R in P_L, and then recursively over the left subproblem and the right subproblem.

This left-to-right visibility step is performed as follows. Let H denote the upward-directed vertical ray emanating from v_μ, which we refer to as the *divider*. We first construct, for each point p of P_R, the lowest (most counterclockwise) ray $\rho^-(p)$ that emanates from p to the left and sees (some point of) H. When querying with a point q, we also construct the lowest (most clockwise) ray $\rho^+(q)$ that emanates from q to the right and sees (some point of) H. We will shortly describe how this ray construction can be performed efficiently. As is easy to verify, q sees p iff the segment qp lies counterclockwise to $\rho^+(q)$ (around q) and clockwise to $\rho^-(p)$ (around p). See Figure 2. We thus need to report all points $p \in P_R$ that satisfy this latter condition.

To this end, we prepare the following multi-level data structure (for each fixed instance of the divide-and-conquer recursion). The first-level structure is a balanced binary tree $\Pi^{(1)}$, which stores the intercepts of the rays $\rho^-(p)$, for $p \in P_R$, at H in their y-order. At each node η of $\Pi^{(1)}$, we create two second-level structures $\Pi^{(2)}_\eta$, $\Pi^{(2*)}_\eta$, on the subset P^η_R of points stored at the leaves of the subtree rooted at η. Both of these structures are for dynamic halfplane range reporting (see [4]), except that $\Pi^{(2)}_\eta$ is constructed on the points of P^η_R in the primal plane, whereas $\Pi^{(2*)}_\eta$ is constructed in the dual plane, on the points dual to the lines supporting the rays $\rho^-(p)$ of the points $p \in P^\eta_R$. The structures $\Pi^{(2)}_\eta$ and $\Pi^{(2*)}_\eta$ are constructed and maintained using the dynamic algorithm of Overmars and van Leeuwen [19], in which one constructs and maintains the upper convex hull of the corresponding set of (primal or dual) points. A deletion costs $O(\log^2 m)$ time, and a query takes $O(\log m)$ time.[1]

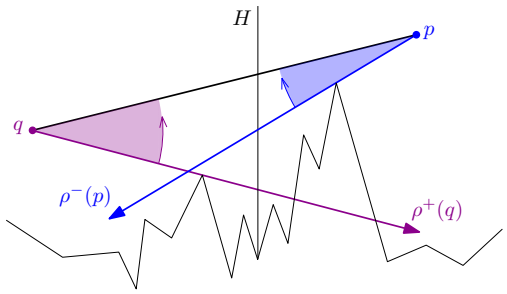

Figure 2 q and p are mutually visible because qp lies counterclockwise to $\rho^+(q)$ and clockwise to $\rho^-(p)$.

A query with a point q (that lies on or above T_L) is performed as follows. We first search in $\Pi^{(1)}$ for the intercept q_H of $\rho^+(q)$ at H, and obtain the subsets P^-_R and P^+_R of P_R, whose intercepts lie below q_H and above q_H, respectively, each as the union of $O(\log m)$ canonical subsets, stored at $O(\log m)$ respective nodes η of $\Pi^{(1)}$.

Consider a canonical subset P^η_R of P^-_R, rooted at a node η of $\Pi^{(1)}$. We search, in the primal plane, with the upper halfplane $h(\rho^+(q))$, bounded by the line supporting $\rho^+(q)$, in $\Pi^{(2)}_\eta$, and report all the points of P^η_R that lie in that halfplane. See Figure 3(a). The collection of all the reported points, over all nodes η, constitutes one portion of the output to the query.

The points of P^+_R are handled differently. For each node η of $\Pi^{(1)}$ such that P^η_R is a canonical subset in the representation of P^+_R, we now search, in the dual plane, with the lower halfplane $h(q^*)$ bounded by the line q^* dual to q, in the structure $\Pi^{(2*)}_\eta$, and report all

[1] One can use instead more asymptotically efficient structures, such as that of [9], at the cost of making the algorithm considerably more involved, which we prefer to avoid.

the points p of P_R^η for which the point dual to the line supporting the ray $\rho^-(p)$ lies in $h(q^*)$. The collection of all the reported points, over all nodes η, constitutes the second portion of the output to the query. As already said, as soon as a point is detected, we immediately delete it from all the $O(\log n)$ substructures (within the top divide-and-conquer recursion) in which it is stored. We repeat this procedure for each of the $O(\log n)$ instances of the divide-and-conquer recursion in which q participates.

As there are $O(\log n)$ instances of the divide-and-conquer recursion in which q participates, and at each of them we process $O(\log m)$ canonical subsets, the overall cost of a query is $O(\log n \log^2 m)$. By the same reasoning, a deletion takes $O(\log n \log^3 m)$ time.

The correctness of the query procedure is established in the following lemma.

▶ **Lemma 1.** *(i) For $p \in P_R^-$, if p lies in the upper halfplane $h(\rho^+(q))$, then q lies in the upper halfplane $h(\rho^-(p))$. (If p lies below $\rho^+(q)$ then p is not visible from q.)*
(ii) For $p \in P_R^+$, if q lies in the upper halfplane $h(\rho^-(p))$, then p lies in the upper halfplane $h(\rho^+(q))$. (If q lies below $\rho^-(p)$ then p is not visible from q.)

Proof. We prove the first claim; the proof of the second claim is similar. Let L_p denote the vertical line through p. By assumption, $\rho^+(q)$ intersects H at a point above $\rho^-(p) \cap H$, and $\rho^+(q)$ intersects L_p at a point below $\rho^-(p) \cap L_p = p$. Hence $\rho^+(q)$ and $\rho^-(p)$ must intersect at some point w between H and L_p, which then implies that $\rho^+(q)$ lies above $\rho^-(p)$ to the left of w. In particular, q lies above $\rho^-(p)$, as asserted. See Figure 3(b). ◀

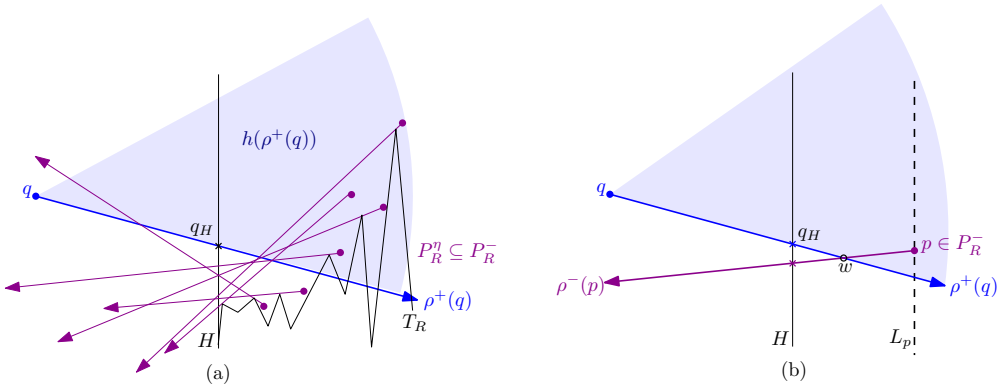

Figure 3 (a) For a canonical set P_R^η of intercepts below q_H, we need to report all the points of P_R^η that lie in the (shaded) upper halfplane $h(\rho^+(q))$. (b) p lies in the upper halfplane of $\rho^+(q)$ (shaded), i.e., $\rho^+(q)$ intersects L_p below p. Since it intersects H at a point above $\rho^-(p)$, $\rho^+(q)$ and $\rho^-(p)$ intersect each other between the lines H and L_p.

Computing the critical rays. To complete the description, we present an efficient algorithm for constructing the rays $\rho^+(q)$ and $\rho^-(p)$. We continue to use the same notations as above. Consider, without loss of generality, the case of points $p \in P_L$ and their rightward-directed rays $\rho^+(p)$; the other case is treated in a fully symmetric manner. Take the vertices $v_1, \ldots, v_\mu$ of T_L, and store them, in this order, at the leaves of a balanced binary tree Q. Each node ξ of Q is associated with the canonical set $T_L(\xi)$ of the vertices stored at the leaves of the subtree rooted at ξ. We form the upper convex hull $C(\xi)$ of $T_L(\xi)$, and store it at ξ. Since the convex hull of a set of points sorted by their x-coordinates can be computed in linear time, the overall cost of constructing the canonical hulls is $O(n \log n)$.[2]

[2] This bound holds since the same canonical sets arise at all the recursive levels of the divide and conquer construction.

Given a query point q (for which we want to compute $\rho^+(q)$), we search with q in Q, and obtain the set of vertices of T_L that lie to the right of q, as the disjoint union of $O(\log n)$ canonical subsets $T_L(\xi)$. We then find the tangent from q to $C(\xi)$, for each such ξ. The highest (most counterclockwise) of these tangents is the desired ray $\rho^+(q)$. The cost of a query is $O(\log^2 n)$, as we have $O(\log n)$ tangent-finding steps, each taking $O(\log n)$ time. We can reduce the cost to $O(\log n)$, using *fractional cascading* [12]. Hence this step takes a total of $O(m \log^2 n)$ time, over all recursive instances of the divide and conquer.

This completes the description of the BFS procedure. We thus have:

▶ **Theorem 2.** *Let T be a 1.5-dimensional polygonal terrain with n vertices, and let P be a set of m points that lie above or on T. We can perform BFS on the visibility graph $VG(P,T)$ in time $O(n \log n + m \log^3 n + m \log n \log^2 m) = O(n \log n + m \log^3(m+n))$.*

3 RSP in visibility graphs over a terrain

Assume now that the points of P lie on T. For a point $p \in P$, let $p(h)$ be the point $p + (0,h)$, i.e., $p(h)$ is the tip of a vertical tower of height h erected from p. For a subset P' of P, we write $P'(h)$ to denote the set $\{p(h) \mid p \in P'\}$. We consider the visibility graph $VG(P(h),T)$ of the tower tips, which we also denote as $VG_h(P,T)$. The problem addressed in this section is the *reverse shortest path (RSP)* problem, in which we are given two points s, t of P and an integer k, and want to find the smallest height h^* for which $VG_{h^*}(P,T)$ has a path between s and t of at most k edges.

This is an optimization problem, whose decision problem is the BFS algorithm (we start at s, stop when t is reached, if at all, and check whether the layer containing t is of index at most k). The problem is that BFS is difficult to parallelize, so we cannot use standard parametric search to solve the RSP problem, and instead we resort to the shrink-and-bifurcate technique of [6, 16], as mentioned in the introduction, and as will be reviewed later in this section. Nevertheless, some preliminary portions of the BFS algorithm are easy to parallelize, and we will simulate them using standard parametric search. This preliminary stage is essential for the efficient simulation of the rest of the algorithm – see below.

3.1 The preliminary stage

In this stage we simulate at h^* the first steps of the BFS algorithm. Specifically, at each recursive instance of the top divide-and-conquer recursion, we have a left portion T_L and a right portion T_R of T, separated by some vertex v of T. We have a set P_L of tower bases on T_L and a set P_R of bases over T_R. We erect an upward vertical divider H from v, and perform the following two steps:

Step 1: For each $p \in P_L$, find the vertex of T_L that determines the lowest (most clockwise) rightward-directed ray $\rho^+(p(h^*))$ from $p(h^*)$ that sees H, and store it with p. Similarly, for each $p \in P_R$, find the vertex of T_R that determines the lowest (most counterclockwise) leftward-directed ray $\rho^-(p(h^*))$ from $p(h^*)$ that sees H, and store it with p.

Step 2: Sort the points $p \in P_L \cup P_R$, in the vertical order in which the rays $\rho^+(p(h^*))$ (for $p \in P_L$) and $\rho^-(p(h^*))$ (for $p \in P_R$) meet H. From the combined sorted list of these intercepts we obtain the two separate sorted lists, for P_L and for P_R.

To perform this preliminary stage, we simulate these steps using standard parametric search. The simulation goes in parallel over all the recursive instances of the divide-and-conquer partitions. Consider one such instance, and for specificity consider the handling of P_L and T_L (handling P_R and T_R is done in a fully symmetric manner). Each point

$p \in P_L$ retrieves the set of vertices of T_L that lie between it and the respective divider H, as the union of $O(\log n)$ canonical sets, each associated with its (precomputed) upper convex hull. We compute the upper tangents from $p(h^*)$ to each hull, and take the highest (most counterclockwise) of them as the desired ray $\rho^+(p(h^*))$.

In the simulation of this step, everything can be efficiently parallelized. Handling all the recursive instances of the divide and conquer is performed in parallel, as the handling of each point $p \in P_L$. Retrieving the canonical hulls for each p takes $O(\log n)$ time, and does not depend on h^* at all. Finding the upper tangents from points p (i.e., $p(h^*)$) to hulls C is done in parallel over all points p and hulls C. For a fixed pair p, C, finding the upper tangent from $p(h^*)$ to C is done by binary search over the vertices of C, which takes $O(\log n)$ time, and thus $O(\log n)$ parallel depth. The final step, of finding, for each p, the most counterclockwise of the tangents from p, is a maximum-finding step on $O(\log n)$ elements, which can also be easily performed in $O(\log n)$ parallel depth. In conclusion, with $O(m \log^2 n)$ processors and $O(\log n)$ parallel depth, we can simulate Step 1. At each parallel layer we perform $\log(m \log^2 n) = O(\log(m+n))$ calls to the decision procedure, namely to the full BFS algorithm, each of which takes $O(n \log n + m \log^3(m+n))$ time. Altogether, the overall cost of the simulation of Step 1 is $O(m \log^3 n + (n \log n + m \log^3(m+n)) \log(m+n) \log n)$; to simplify this bound, we write it as $O(n \log^3(m+n) + m \log^5(m+n))$.

Step 2 is also easy to simulate in parallel. We have $O(m)$ critical rays, and we need to simulate the sorting of their intercepts with H in their vertical order. The sorting can be simulated in $O(\log m)$ parallel depth with $O(m)$ processors, using, e.g., the sorting network of [5]. Repeating this over all instances of the divide and conquer, we conclude that the total time for this step is $O(m \log m \log n + (n \log n + m \log^3(m+n)) \log m \log n) = O(n \log^3(m+n) + m \log^5(m+n))$.

Upon termination of the simulation of the preliminary stage, we have an interval $I = [h_1, h_2]$ that contains h^*, so that, for each $h \in I$, the discrete nature of each critical ray (i.e., the vertex of T through which it passes) is fixed, as is the order of their intercepts along H, and this holds for each instance of the divide-and-conquer recursion.

3.2 The shrinkage stage

Next, we want to shrink $I = [h_1, h_2]$ into a subinterval that still contains h^* and at most some prescribed number L of critical heights. Here a critical height h is of the following form. There are two points $p_L \in P_L$, $p_R \in P_R$, and a vertex v of T (either of T_L or of T_R) between p_L and p_R, such that the line segment $p_L(h)p_R(h)$ passes through v, and all the other vertices of T between p_L and p_R lie (on or) below the segment. We denote this height as h_{p_L,p_R}. In other words, h_{p_L,p_R} is the smallest height h at which $p_L(h)$ and $p_R(h)$ see each other. Our goal is to estimate the number of heights h_{p_L,p_R} that satisfy $h_1 \le h_{p_L,p_R} \le h_2$ (over all the divide-and-conquer instances). If this estimate is smaller than L, we output I and terminate the shrinkage stage. If it is larger, the analysis in [6, 16] shows that, by picking a random sample of a suitable size from the critical heights in I, and by running a binary search through the heights in the sample, we can shrink I to a smaller interval that contains h^* and, with high probability, contains at most L critical heights.

Observe that, by definition, if $v \in T_L$ (resp., $v \in T_R$) then $p_L(h_{p_L,p_R})p_R(h_{p_L,p_R})$ overlaps the ray $\rho^+(p_L(h_{p_L,p_R}))$ (resp., $\rho^-(p_R(h_{p_L,p_R}))$), and passes above the other ray $\rho^-(p_R(h_{p_L,p_R}))$ (resp., $\rho^+(p_L(h_{p_L,p_R}))$). Assume for specificity that $v \in T_L$. Then, if $h > h_{p_L,p_R}$ (resp., $h < h_{p_L,p_R}$), the segment $p_L(h)p_R(h)$ passes above (resp., below) $\rho^+(p_L(h))$; in the former case it clearly also passes above $\rho^-(p_R(h))$. See Figure 4 (Left).

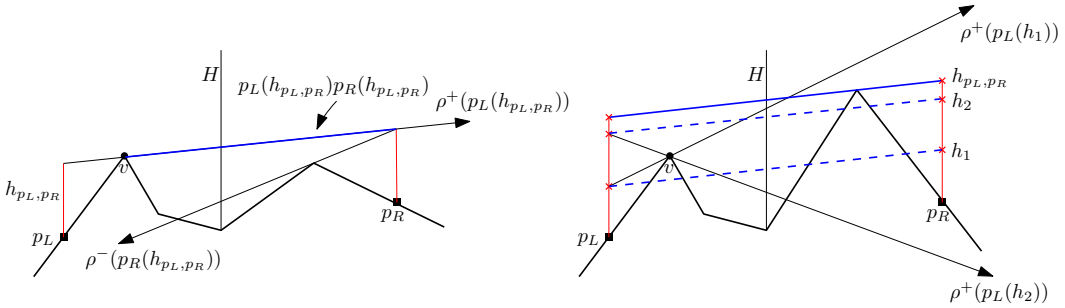

Figure 4 Left: $p_L(h_{p_L,p_R})p_R(h_{p_L,p_R})$ overlaps $\rho^+(p_L(h_{p_L,p_R}))$ and passes above $\rho^-(p_R(h_{p_L,p_R}))$. Right: $p_L(h_1)p_R(h_1)$ passes below $\rho^+(p_L(h_1))$ and $p_L(h_2)p_R(h_2)$ passes above $\rho^+(p_L(h_2))$, but $h_{p_L,p_R} \notin I$.

In other words, h_{p_L,p_R} is in I if $p_L(h_1)p_R(h_1)$ passes below $\rho^+(p_L(h_1))$, and $p_L(h_2)p_R(h_2)$ passes above $\rho^+(p_L(h_2))$. (This is not an equivalent formulation, since it ignores the condition on the other ray $\rho^-(p_R)$, but it suffices for our purposes, in the sense that the heights that satisfy this modified constraint form a superset of the true critical heights. Thus an interval with a small number of heights in this superset contains also a small number of true critical heights.) See Figure 4 (Right). The preceding discussion pertains to the case where the intermediate vertex v belongs to T_L. A fully symmetric analysis holds when v is in T_R.

We can formulate the above condition as a batched range searching problem. To simplify the formulation, it helps to keep the tower bases fixed, and to shift the terrain downwards by h_1 and by h_2. The above condition (namely, our necessary condition for h_{p_L,p_R} to belong to I) can be regarded as a range searching query, where each range is defined by p_L (and v), and consists of the set of all points of P_R that lie above the ray $\rho^+(p_L(h_2))$ shifted down by h_2, and below the ray $\rho^+(p_L(h_1))$ shifted down by h_1; see Figure 5.

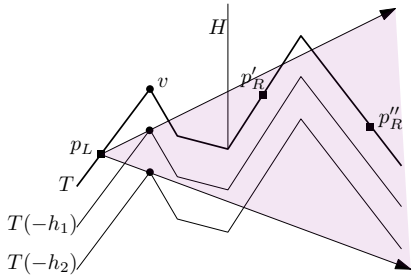

Figure 5 The necessary condition for h_{p_L,p_R} (for $p_R \in P_R$) to belong to I can be regarded as a wedge range searching query.

This brings our setup more or less to the setup considered in [6], except that the ranges considered there were annuli rather than the wedges that arise here. We can apply the same machinery, suitably modified, to estimate the number of critical heights in I. In doing so, two things should be emphasized. First, the ranges were defined by points of P_L because we have considered the case where the anchor vertex v belongs to T_L. As already said, a symmetric setup should also be applied to cover the case where v is a vertex of T_R. Second, what we have just described covers only one instance of the divide-and-conquer process, and we should apply this procedure to all instances. This requires a careful choice of the threshold numbers L of critical heights at each instance, an analysis that will appear in

the full version of the paper.[3] Taking care of these issues, we obtain a randomized interval shrinking procedure that, with high probability, produces an interval I that contains h^* and at most L other critical heights, in randomized expected time $O^*(m^{4/3}/L^{1/3})$. See [6, 16] for more details.

3.3 The bifurcation stage

We follow the paradigm in [6, 16]. It proceeds in phases, where in each phase we simulate some portion of the BFS algorithm. Each phase starts with some interval I that contains h^* and progressively fewer other critical heights. We bifurcate the simulation whenever we have to resolve a comparison whose critical height h is in I, exploring both outcomes $h < h^*$ and $h \geq h^*$. The phase terminates either when we have collected some threshold number of unresolved comparisons (i.e., bifurcations), or when every path in the resulting bifurcation tree is of some threshold depth. We then determine the outcome of all the collected unresolved comparisons, running a binary search over their critical heights, and using the (unsimulated) BFS procedure to guide the search. This produces a smaller interval I' that still contains h^*, and we start a new bifurcation phase from the leaf of the bifurcation tree to which h^* belongs. The procedure terminates when the entire BFS procedure has been simulated, and then h^* is readily obtained.

This is the general approach, following the paradigm used in [6, 16], but it requires some nontrivial fine-tuning to fit into our setup. First, there is no need to simulate the preliminary stages of the BFS, as this has already been done. Second, and more important, at each step of the latter portion of the BFS, we process a visited tower $p(h^*)$ and wish to find an unvisited neighbor. For this, the BFS uses a two-level dynamic data structure. Accessing the first level requires no simulation. Indeed, assume that p is in P_L (at some node of the divide-and-conquer process). We access the first level to obtain the set of unvisited towers in $P_R(h^*)$, for which the intercepts of their critical rays $\rho^-(q(h^*))$ lie above (or below) the intercept of $\rho^+(p(h^*))$. Since we already have a fixed sorted order of the intercepts, this can be computed explicitly, without any simulation.

A direct simulation of the second-level structure at h^* is problematic for the following reason. A standard approach to the two-dimensional dynamic halfplane range reporting maintains the convex hull of the input set, which is typically done using the technique of Overmars and van Leeuwen [19]. So the machinery needs to perform the construction and dynamic reconstruction of various convex hulls of primal or dual points. The basic operation involved in these constructions is the *orientation test* of a triple of points. Unfortunately, the critical heights produced in these tests are not of the types considered (that is, allowed) in the shrinkage stage, nor can a variant of that stage handle them, as each of them depends on a *triple* of points, whereas the range-searching machinery used in the shrinkage stage can only handle critical heights defined by *pairs* of points.

To address this issue, we proceed as follows. Consider, for specificity, the handling of the set P_R (for queries with points of P_L). For each canonical set $P_\mu \subseteq P_R$ of the first-level structure, we construct two dynamic halfplane range reporting data structures, as does the BFS, but we construct them at the fixed, known height h_2, rather than at h^*. As we recall, the primal structure, $\Pi_\mu^{(2)}$, is for $P_\mu(h_2)$, and the dual structure, $\Pi_\mu^{(2*)}$, is for the set $P_\mu^*(h_2)$ of points dual to the lines containing the rays $\rho^-(p(h_2))$, for $p \in P_\mu$.

[3] Informally and briefly, at depth j of the divide and conquer we have subproblems of average size $m/2^j$, and we want to shrink I so that it contains at most $L/(2^j \log m)$ critical heights of these subproblems. An easy analysis shows that, with high probability, I contains upon termination of this process at most L critical heights, and the expected cost of the procedure is $O^*(m^{4/3}/L^{1/3})$.

Consider now a query with some point $q \in P_L(h^*)$, whose goal is to report the points of $P_R(h^*)$ that it sees and have not yet been visited by the BFS. We first partition P_R into two subsets, P_R^- and P_R^+, where P_R^- (resp., P_R^+) consists of the points $p \in P_R$ for which the intercepts of their corresponding rays $\rho^-(p(h^*))$ lie on H below (resp., above) the intercept of $\rho^+(q)$. We obtain each of these subsets as a collection of $O(\log |P|)$ canonical subsets. As already noted, this step requires no simulation.

For each canonical subset P_μ in the representation of P_R^-, we query the data structure $\Pi_\mu^{(2)}$ with the upper halfplane defined by the line containing $\rho^+(q(h_2))$, the critical ray from $q(h_2)$. Let P'_μ be the subset of points reported. For each $p \in P'_\mu$, we compute the minimum height $h_{q,p}$ at which q and p see each other, that is, the critical height associated with q and p. If $h_{q,p} \leq h_1$, we conclude that $q(h^*)$ and $p(h^*)$ see each other, add p to the next layer of the BFS tree, and delete p from $\Pi_\mu^{(2)}$ and from all substructures in which it appears. If however $h_{q,p} > h_1$ (by construction, we also have $h_{q,p} \leq h_2$, since $h_{q,p}$ is in I), then $h_{q,p}$ is one of the at most L critical values in the interval I. In a similar manner, for each canonical subset P_μ in the representation of P_R^+, we query the dual data structure $\Pi_\mu^{(2*)}$ with the lower halfplane defined by the line dual to $q(h_2)$. Again, we take each reported point p (the point inducing a reported dual point to its critical ray), and test whether $h_{q,p} \leq h_1$ or $h_{q,p} > h_1$. In the former case we conclude that p should be added to the next BFS layer (at h^*), and in the latter case we obtain a critical height among the at most L critical heights in I.

In both primal and dual setups, we bifurcate the simulation at each of the critical heights in I that has been discovered by the above procedure, and follow the simulation accordingly, as described earlier. (That is, these critical values are counted towards the bifurcation threshold that determines when the current bifurcation phase should terminate.)

This completes the review and the modifications of the bifurcation step. As in [6, 16], a careful choice of the various threshold parameters yields an algorithm that runs in $O^*((m+n)^{6/5})$ time. That is, we obtain:

▶ **Theorem 3.** *Let T be a 1.5-dimensional polygonal terrain with n vertices, and let P be a set of m points that lie on T. Given two points $s, t \in P$ and an integer k, we can find the smallest height h^* for which the visibility graph $VG(P(h^*), T)$, of the tips of the towers of height h^* erected at the points of P, contains a path from $s(h^*)$ to $t(h^*)$ of at most k edges. The algorithm runs in $O^*((m+n)^{6/5})$ randomized expected time.*

By setting $k = n - 1$ and running a similar simulation of the full BFS algorithm from an arbitrary point $s \in P$ (without specifying a target point t), we obtain the following corollary.

▶ **Corollary 4.** *Given T and P as above, we can determine the minimum height h^* for which the graph $VG(P(h^*), T)$ is connected, in randomized expected $O^*((m+n)^{6/5})$ time.*

4 DFS in visibility graphs over a terrain

The machinery in Section 2 can be adapted to yield a near-linear algorithm for performing Depth-First Search (DFS) in the graph $VG(P, T)$. At each step of the DFS, we are at some point $p \in P$, and wish to find some neighbor of P that has not yet been visited. Let U be the (dynamically changing) set of unvisited points of P. We initially set $U = P \setminus \{p_0\}$, where p_0 is the start point of the DFS. We maintain U under deletions, removing each point when it is visited by the algorithm.

Except for the fact that we run DFS instead of BFS, and thus use a different flow of execution, the technical details are basically the same: We maintain the same divide-and-conquer structure, compute, at each instance of the structure, the critical visibility rays, sort

their intercepts along the corresponding divider H, and, for each resulting canonical set of intercepts (that is, of left points or right points), we compute the upper convex hull of the set.

A query with a point $q \in P$ is performed as follows. We collect all the $O(\log n)$ instances of the divide-and-conquer structure at which q is stored. Consider one such instance, and assume, for concreteness, that q belongs to the left corresponding set P_L. We take the critical ray $\rho^+(q)$ and its intercept $w_q = \rho^+(q) \cap H$ with the divider H. We collect all the unvisited points of P_R whose intercepts lie above w_q as the union of $O(\log m)$ canonical sets. We do the same for the intercepts below w_q. Denote the former (resp., latter) subset as $P_R^+(p)$ (resp., $P_R^-(p)$).

For each canonical set P_η of $P_R^+(p)$, we query in the dual halfplane range reporting structure with the lower halfplane bounded by the line dual to q, and for each canonical set P_η of $P_R^-(p)$, we query in the primal structure with the upper halfplane bounded by the line supporting $\rho^+(q)$. If one of the queries yields a point in the respective halfplane, we report it as a neighbor of q and stop the search; the justification for this step is exactly as in Section 2. We repeat this step to each canonical set P_η of $P_R^+(p)$ and $P_R^-(p)$ in the present instance, and to each instance of the divide and conquer that stores q. We stop as soon as a neighbor is found. When we do find a neighbor, we delete it from the structure: it belongs to $O(\log n \log m)$ sets P_η, for $O(\log n)$ instances of the divide and conquer and for $O(\log m)$ canonical sets at each instance, and we delete it from each of them. If no neighbor is found in all the searches, we conclude that all the neighbors of q have already been visited, and back up the DFS tree from q to its parent, and resume the search from there, or if q is the root of its tree, we pick some unvisited point and start a new DFS tree from it.

Correctness follows from the same considerations as in Section 2. The running time, as in Section 2, is $O(n \log n + m \log^3 n + m \log n \log^2 m)$. Simplifying this bound as in Section 2, we thus conclude:

▶ **Theorem 5.** *Given a terrain T with n vertices, and a set P of m points on or above T, we can perform DFS on $VG(P,T)$ in $O(n \log n + m \log^3(m+n))$ time.*

Detecting cycles in $VG(P,T)$. A basic application of DFS on undirected graphs is detecting cycles in the graph (this is a task that standard BFS cannot perform). In a standard application of DFS, detecting a cycle amounts to testing whether the graph has a *back edge*, namely an edge that connects a node v to a proper ancestor of v (in the DFS forest) other than its parent. This is a trivial task to perform when we can process the edges of the graph, but is more involved when we can access explicitly only its vertices (the points of P in $VG(P,T)$), and requires the following roundabout approach.

In addition to maintaining the set U of unvisited points, we also build the complementary set R of visited points, inserting into R each newly visited point. Let p be the point just visited, and let $\tilde{p}$ denote the parent of p in the DFS forest (we do not perform this step when p is the root of some DFS tree). We first delete (temporarily) p and $\tilde{p}$ from R, and then test whether p has a neighbor in (the modified) R, using the same machinery as above. If there is such a neighbor, we have found a back edge, and can therefore report that the graph has a cycle. If not, we add p and $\tilde{p}$ back to R, and continue with the execution of the DFS. We thus obtain:

▶ **Theorem 6.** *Given a terrain T with n vertices, and a set P of m points on or above T, we can test whether $VG(P,T)$ is acyclic in $O(n \log n + m \log^3(m+n))$ time.*

Remark. A possible extension of the setup, which is often used in sensor networks, is to assume that each point of P has a broadcasting range. In this case, visibility is not sufficient (albeit it is necessary) to ensure communication between the points. Moreover, if the points have different ranges, the graph $VG(P,T)$ becomes directed. In this case, running DFS on the graph has many additional applications, such as topological sorting, computing strongly connected components, and so on. We do not consider this extension in the present work, and leave it as an interesting problem for further research.

Finding the smallest towers with a cyclic visibility graph. We can turn the cycle detection algorithm into an optimization procedure involving towers of a common height erected at the points of P, now assumed to lie on T, where the problem is to find the smallest height h^* for which $VG_{h^*}(P,T)$ (as defined in Section 3) contains a cycle. The procedure is implemented in a manner very similar to the RSP algorithm. That is, we run the same preliminary stage, to find the combinatorial representation of all the critical rays (using now the cycle detection algorithm as the decision procedure, instead of the BFS algorithm used earlier), and then simulate the main procedure, in much the same way as before. Omitting the straightforward details, we obtain:

▶ **Theorem 7.** *Given a terrain T with n vertices, and a set P of m points on T, we can compute the smallest height h^* for which $VG_{h^*}(P,T)$ contains a cycle, in $O^*((m+n)^{6/5})$ randomized expected time.*

5 Seeing the most without being seen

In this section we study the following *maximum-height independent towers* problem. Let T and P be as in the preceding sections, with $|T| = n$ and $|P| = m$. The problem is to compute the smallest height h^*, such that if we place a tower of height h^* at each of the points $t \in P$, then some pair of tips of these towers see each other (so for any $h < h^*$, no pair of tips see each other). See Figure 6.

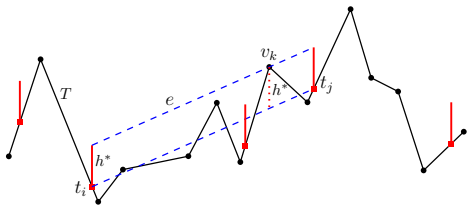

Figure 6 The maximum-height independent towers problem.

Consider the scene in which a tower of height h^* is positioned at each of the points $t \in P$. Then there must exist a pair of points $t_i, t_j \in P$, such that the segment e between the tips of the corresponding towers passes through a vertex v_k of T, and none of the vertices of T between t_i and t_j lies above e; see Figure 6. In other words, the vertical distance between v_k and the segment t_it_j is h^* (with v_k lying above t_it_j), and it is the maximum vertical distance between any intermediate vertex and t_it_j. As before, and more generally, we call a height h *critical* if there exist a pair of points $t_i, t_j \in P$ that satisfies the above property, with some in-between vertex v, with height h.

The decision problem that underlies our optimization problem, which is of independent interest, is, for a given h, test whether $VG_h(P,T)$ is empty, or, for that matter, test whether $VG(Q,T)$ is empty, for any set Q of points on or above T. This problem has been studied by

Ben-Moshe et al. [7, Section 5], who present an $O((n+m)\log m)$-time algorithm to determine whether there exist two points in the set Q that see each other. A suitable adaptation (in fact, simplification) of the algorithm presented in this section also solves this graph emptiness problem, by an algorithm that also runs in near-linear time (albeit slightly less efficiently than the algorithm in [7, Section 5]). However, due to the special nature of the optimization problem, there is no need for a separate decision procedure that needs to be simulated, in the parametric searching style. Putting it differently, each time we perform a comparison with a critical height h, we can conclude that $h^* \le h$, for otherwise the pair of points involved in the comparison would be mutually visible above T. This one-way search for h^* makes the whole algorithm considerably simpler, and efficient. See below for full details.

Our algorithm runs in several stages. It maintains a candidate value of h^*, which, for simplicity of notation, we also denote as h^*. At any time during execution, h^* is a critical height for some pair of tower tips (and an in-between terrain vertex), which would see one another for $h \ge h^*$ but not for $h < h^*$. Thus the optimum height h^* is at most the current value of this parameter.

The algorithm uses the same divide-and-conquer approach as before. It partitions T at its median vertex v_μ into a left portion T_L and a right portion T_R. The set P of tower bases is split accordingly into a left subset P_L and a right subset P_R. The algorithm computes recursively the optimal heights h^*_L and h^*_R for the respective subproblems involving T_L, P_L and T_R, P_R, and sets $h^* := \min\{h^*_L, h^*_R\}$.

In the merge step, we only need to consider interactions between a left tower and a right tower, whose corresponding critical height is at most the current value of h^*. As before, let H denote the divider at which the left-right split of T and P takes place. We compute, for each left tower t (resp., right tower t'), that is for its top $t(h^*) = t + (0, h^*)$ (resp. $t'(h^*)$), the corresponding critical ray $\rho^+(t(h^*))$ (resp., $\rho^-(t'(h^*))$). Each of the rays $\rho^+(t(h^*))$ must pass through some vertex of T_L, and each ray $\rho^-(t'(h^*))$ passes through some vertex of T_R (either of these vertices could be v_μ). These rays are constructed similarly to the construction in Section 2 (note that here the current height h^* is known).

We now pass to the dual plane. For technical reasons, as in the previous sections, it is more convenient to keep the towers by representing them by their bases and shifting T (but not the bases) downwards by h^*. For each left tower t, the set of all rightward directed rays that emanate from t and pass above (counterclockwise to) $\rho^+(t)$ is mapped to a rightward-directed ray that lies on the dual line t^* and emanates from the point $\rho^+(t)^*$ dual to $\rho^+(t)$ (so only the apex of this dual ray depends on h^*). Symmetrically, for each right tower t', the set of all leftward directed rays that emanate from t' and pass above (clockwise to) $\rho^-(t')$ is mapped to a leftward-directed ray that lies on the dual line $(t')^*$ and emanates from the point $\rho^-(t')^*$ dual to $\rho^-(t')$. See Figure 7 for an illustration.

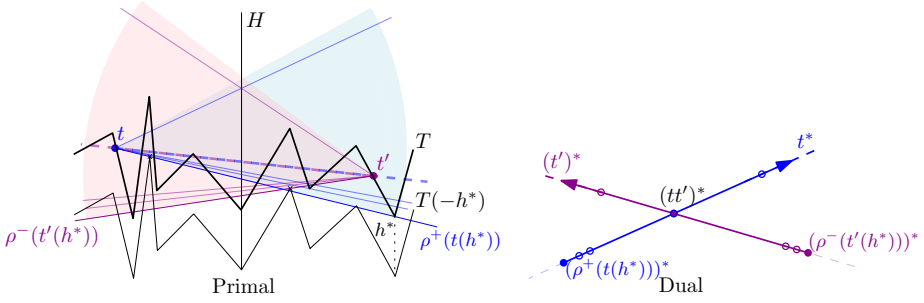

Figure 7 An intersection in the dual plane indicates a visible pair in the primal plane.

A left tower t and a right tower t' see each other if and only if the segment tt' lies counterclockwise to $\rho^+(t)$ and clockwise to $\rho^-(t')$. (We remind the reader that all this discussion is with respect to the scenario in which we consider visibility between the tower bases relative to the terrain being shifted down by h^*.) In the dual, tt' is mapped to the intersection point $(tt')^*$ of the lines t^* and $(t')^*$, and the above visibility condition is mapped to the condition that the rays $\rho^+(t)^*$ and $\rho^-(t')^*$ intersect one another (at the point $(tt')^*$).

The merge step is therefore mapped to the following setup. We have, in the dual plane, $|P_L|$ rightward-directed rays and $|P_R|$ leftward-directed rays, and the task is to compactly represent all the intersections between a rightward-directed ray and a leftward-directed ray. Fortunately, we have the following property (which is special to the problem considered in this section).

▶ **Lemma 8.** *The rightward-directed rays are pairwise openly disjoint, and so are the leftward-directed rays.*

Proof. Assume to the contrary that a pair of, say, rightward-directed rays $\rho^+(t_1)^*$ and $\rho^+(t_2)^*$ properly cross one another. The intersection point is dual to the line ℓ that connects t_1 and t_2, or rather (ignoring the downward shift of T and going back to the original scenario) $t_1(h^*)$ and $t_2(h^*)$. Assume without loss of generality that t_1 lies to the left of t_2 (they both lie to the left of H). But then the intersection point ℓ^* (dual to ℓ) lies in both dual rays, and in particular it belongs to $\rho^+(t_1)^*$. This means that $t_1(h^*)t_2(h^*)$ is a free segment above T_L (because $t_1(h^*)$ can see H in this direction). But this is impossible, since we only consider values $h^* \leq h^*_L$, and, by definition, two left towers at such a height h^* cannot see one another (with a visibility segment that lies above T). ◀

That is, we seek the largest value of h^* for which no pair of dual rays cross one another, over all recursive instances of the divide and conquer.

In view of the lemma, we compactly represent all the intersections between a rightward-directed ray and a leftward-directed ray, where the rightward-directed rays are pairwise openly disjoint, and so are the leftward-directed rays. To this end, we construct a segment tree Π over the (x-projections of the) rays, in $O(m \log m)$ time. Let v be a node of Π, and let $R(v)$ denote the set of rays that are stored at v. Every such ray straddles the vertical slab Σ_v associated with v from side to side. We also associate with v the set Q_v of ray endpoints that lie in Σ_v. Each endpoint is stored at $O(\log m)$ nodes of Π. Call the rays that are stored at v *long rays*, and those with an endpoint inside Σ_v *short rays*.

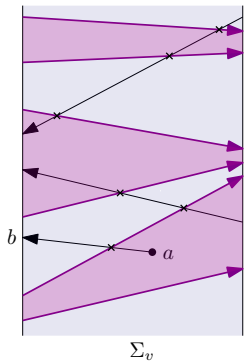

Figure 8 The long rightward-directed rays partition Σ_v into a sequence of trapezoids.

For each node v we need to represent all the intersections between a pair of long rays at v, and all the intersections between a long ray and a short ray at v. Each of these tasks can be implemented in time nearly linear in $|R(v)| + |Q_v|$, as follows. Recall first that we only

need to consider intersections between a rightward-directed ray and a leftward-directed ray. We take the set of long rightward-directed rays, sort them in increasing y-order, and store them at the leaves of a secondary binary search tree T_v^r. The long rightward-directed rays partition Σ_v into a sequence of trapezoids, where the bottommost and the topmost ones are unbounded. See Figure 8. We then iterate over the long and the short leftward-directed rays at v. Each such ray ρ, confined to Σ_v, is a segment with two endpoints a and b, where a lies either on the right boundary of Σ_v (when ρ is long) or in the interior of Σ_v (when ρ is short), and b lies on the left boundary of Σ_v, and ρ intersects all the long rightward-directed rays between the trapezoid that contains a and the trapezoid that contains b. These rays can be represented as the union of $O(\log m)$ canonical subsets associated with nodes of T_v^r, and we store ρ at these $O(\log m)$ nodes. To complete this step, we apply a symmetric procedure in which the roles of the rightward-directed and leftward-directed rays are interchanged, using another secondary tree T_v^l. (For long-long intersection detection, one of the procedures suffices.) At this point, we have obtained a representation of all the intersections at v as a collection of $O(|R(v)|)$ complete bipartite graphs (bi-cliques), one per each node of T_v^r and T_v^l, of total size $O((|R(v)| + |Q_v|) \log m)$. After applying this step to each node v of Π, we get a compact representation of all the intersections between rightward-directed rays and leftward-directed rays as a collection of edge-disjoint bi-cliques of total size $O(m \log^2 m)$.

Each crossing point between a rightward-directed ray $\rho^+(t)^*$ and a leftward-directed ray $\rho^-(t')^*$ corresponds to a critical height $h_{t,t'} < h^*$, that can be computed in $O(\log^2 n)$ time. Indeed, in the primal, such an intersection is dual to the line tt', and we need to compute the downward shift of the terrain at which tt' touches a vertex, and all the other in-between vertices are below it. As before, this amounts to computing the upper tangent to each of the $O(\log n)$ in-between canonical hulls, that is parallel to tt', and reporting the one closest to tt'; altogether this takes $O(\log^2 n)$ time.

Thus, our goal is to find the minimum height corresponding to such an intersection point. Notice though that we cannot afford to compute all these heights, since the number of intersection points might be $\Theta(m^2)$. Instead, we proceed as follows. The representation of all intersections as a collection of bi-cliques allows us to pick a random sample of intersection points of size $O(m \log m)$, in $O(m \log m)$ time. For each intersection point in the sample, we compute its corresponding critical height, and let h_s be the minimum of these $O(m \log m)$ heights. Then, with high probability, the number of intersection points with corresponding height at most h_s is only $O(m)$. We now set $h^* := h_s$, and repeat the entire merge algorithm for the new value of h^*, except that now, since the expected number of intersection points is only $O(m)$, we can afford to compute all their corresponding heights explicitly and return the minimum of all these heights. We thus obtain:

▶ **Theorem 9.** *Let T be a 1.5-dimensional polygonal terrain with n vertices, and let P be a set of m points that lie on T. We can find the smallest height h^* for which the edge set of the visibility graph $VG(P(h^*), T)$, of the tips of the towers of height h^* erected at the points of P, is nonempty, in $O((n+m)\log^3(m+n))$ randomized expected time.*

References

1 P. K. Agarwal, N. Alon, B. Aronov, and S. Suri. Can visibility graphs be represented compactly? *Discrete Comput. Geom.*, 12:347–365, 1994.

2 P. K. Agarwal, S. Bereg, O. Daescu, H. Kaplan, S. C. Ntafos, M. Sharir, and B. Zhu. Guarding a terrain by two watchtowers. *Algorithmica*, 58:352–390, 2010.

3 P. K. Agarwal, M. J. Katz, and M. Sharir. On reverse shortest paths in geometric proximity graphs. *Comput. Geom.*, 117:102053, 2024.

4 P. K. Agarwal and J. Matoušek. Dynamic half-space range reporting and its applications. *Algorithmica*, 13(4):325–345, 1995.

5 M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3(1):1–19, 1983.

6 R. Ben Avraham, O. Filtser, H. Kaplan, M. J. Katz, and M. Sharir. The discrete and semicontinuous Fréchet distance with shortcuts via approximate distance counting and selection. *ACM Trans. Algorithms*, 11, 2015, Art. 29.

7 B. Ben-Moshe, O. A. Hall-Holt, M. J. Katz, and J. S. B. Mitchell. Computing the visibility graph of points within a polygon. In *Proc. Sympos. on Computational Geometry (SoCG)*, pages 27–35, 2004.

8 B. Ben-Moshe, M. J. Katz, and J. S. B. Mitchell. A constant-factor approximation algorithm for optimal 1.5D terrain guarding. *SIAM J. Comput.*, 36:1631–1647, 2007.

9 G. S. Brodal and R. Jacob. Dynamic planar convex hull with optimal query time. In *Proc. 7th Scandinavian Workshop on Algorithm Theory*, volume 1851 of *Lecture Notes in Computer Science*, pages 57–70. Springer, 2000. See also arXiv:1902.11169.

10 S. Cabello and M. Jejčič. Shortest paths in intersection graphs of unit disks. *Comput. Geom. Theory Appls.*, 48:360–367, 2015.

11 T. M. Chan and D. Skrepetos. All-pairs shortest paths in unit disk graphs in slightly subquadratic time. In *Proc. 27th Int. Sympos on Algorithms and Computation (ISAAC)*, pages 24:1–24:13, 2016.

12 B. Chazelle and L. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1:133–162, 1986.

13 S. Friedrichs, M. Hemmer, J. King, and C. Schmidt. The continuous 1.5D terrain guarding problem: Discretization, optimal solutions, and PTAS. *J. Comput. Geom.*, 7:256–284, 2016.

14 M. Gibson, G. Kanade, E. Krohn, and K. R. Varadarajan. Guarding terrains via local search. *J. Comput. Geom.*, 5:168–178, 2014.

15 J. Hershberger. An optimal visibility graph algorithm for triangulated simple polygons. *Algorithmica*, 4:141–155, 1989.

16 H. Kaplan, M. J. Katz, R. Saban, and M. Sharir. The unweighted and weighted reverse shortest path problem for disk graphs. In *Proc. 31st European Symposium on Algorithms*, pages 67:1–67:14, 2023. Also in arXiv:2307.14663.

17 H. Kaplan, W. Mulzer, L. Roditty, P. Seiferth, and M. Sharir. Dynamic planar Voronoi diagrams for general distance functions and their algorithmic applications. *Discrete Comput. Geom.*, 64:838–904, 2020.

18 M. J. Katz and M. Sharir. Efficient algorithms for optimization problems involving distances in a point set. In arXiv:2111.02052.

19 M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Syst. Sci.*, 23(2):166–204, 1981.

20 M. Pocchiola and G. Vegter. Computing the visibility graph via pseudo-triangulations. In *Proc. 11th Sympos. on Computational Geometry (SoCG)*, pages 248–257, 1995.

21 K. R. Varadarajan. Approximating monotone polygonal curves using the uniform metric. In *Proc. 12th Sympos. on Computational Geometry (SoCG)*, pages 311–318, 1996.

22 H. Wang and J. Xue. Near-optimal algorithms for shortest paths in weighted unit-disk graphs. *Discrete Comput. Geom.*, 64:1141–1166, 2020.

23 H. Wang and Y. Zhao. Reverse shortest path problem for unit-disk graphs. In *Proc. 17th Algorithms and Data Structures Sympos. (WADS)*, pages 655–668, 2021. Also in arXiv:2104.14476.

24 H. Wang and Y. Zhao. Reverse shortest path problem in weighted unit-disk graphs. In *Proc. 16th Internat. Conf. on Algorithms and Computation (WALCOM)*, volume 13174 of *Lecture Notes in Computer Science*, pages 135–146. Springer, 2022.