Reachability in a Planar Subdivision with Direction Constraints*

Daniel Binham[1], Pedro Machado Manhães de Castro[2], and Antoine Vigneron[3]

1 **Visual Computing Center, KAUST, Thuwal, Saudi Arabia**
ringscore@yahoo.com
2 **Centro de Informática, Universidade Federal de Pernambuco, Recife, Brazil**
pmmc@cin.ufpe.br
3 **School of Electrical and Computer Engineering, UNIST, Ulsan, Republic of Korea**
antoine@unist.ac.kr

Abstract

Given a planar subdivision with n vertices, each face having a cone of possible directions of travel, our goal is to decide which vertices of the subdivision can be reached from a given starting point s. We give an $O(n \log n)$-time algorithm for this problem, as well as an $\Omega(n \log n)$ lower bound in the algebraic computation tree model. We prove that the generalization where two cones of directions per face are allowed is NP-hard.

1998 ACM Subject Classification I.3.5 Computational Geometry and Object Modeling

Keywords and phrases Design and analysis of geometric algorithms, Path planning, Reachability

Digital Object Identifier 10.4230/LIPIcs.SoCG.2017.17

1 Introduction

We consider a motion planning problem where a point robot moves within a planar subdivision, with constraints on its direction of travel. Within each face of the subdivision, there is a cone of possible directions of travel, and we want to decide which vertices are reachable from a given starting position. (See Figure 1a.)

This type of constraints appear, for instance, for motion planning in the presence of flows [6]. In this model, a vehicle moves within a flow field, say wind or current. If the speed of the vehicle is less than the speed of the flow, then it can only travel in a cone of directions, with axis parallel to the direction of the current, and whose angle depends on the ratio between the speed of the robot and the speed of the current. (See Figure 1b.)

Our results. Our main result is an $O(n \log n)$-time algorithm to compute all the vertices that are reachable from a given source point s, where n is the size of the input subdivision, and each face has a cone of possible directions of travel (Section 6). We also give a matching $\Omega(n \log n)$ lower bound in the algebraic computation tree model. This result is based on Ben-Or's topological lower bound [1], and holds even in the special case where only one direction of travel is given for each face. Finally, we prove that the generalization where

* D. Binham was supported by KAUST base funding, and A. Vigneron was supported by the 2016 Research Fund (1.160054.01) of UNIST (Ulsan National Institute of Science and Technology).

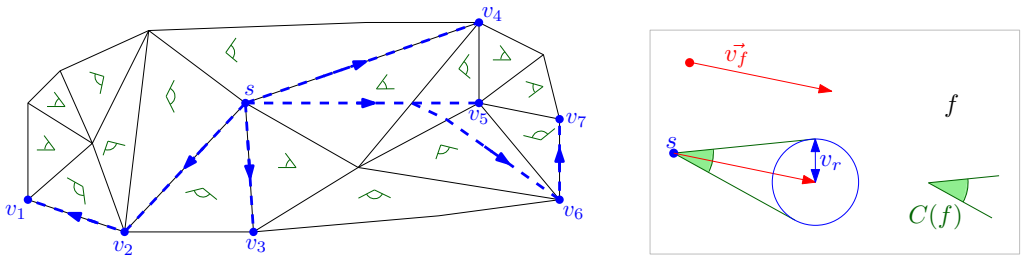


Figure 1 (a) The input to our reachability problem is the triangulation, the cone of direction in each face, and the starting point s. The output is the set of reachable vertices $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$. (b) Within the region f, the velocity of the flow is $\vec{v_f}$ and the control speed of the robot is v_r. The blue circle represents the points that can be reached from s in unit time. Hence, the possible directions of travel are given by the cone $C(f)$.

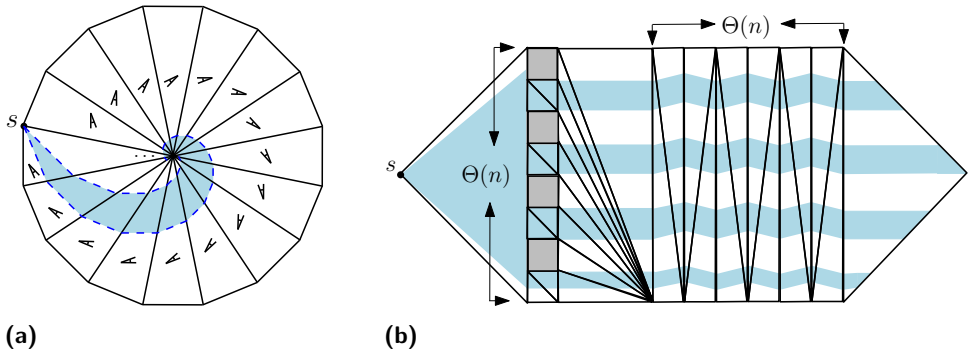


Figure 2 (a) The reachable region (shaded) is a spiral formed by an infinite sequence of blocks. (b) The reachable region has quadratic complexity, without any spiral.

two cones of directions per face are allowed is NP-hard. Our proof is a reduction from the partition problem, and it even holds when only two directions of travel are allowed throughout the subdivision.

A natural approach to compute all reachable vertices would be to construct the reachable region piece by piece, handling one face at a time. Unfortunately, this algorithm would not necessarily terminate as the direction constraints may force a path to follow a spiral with arbitrarily many edges (Figure 2a). Even when there are no such spirals, the complexity of the reachable region can still be quadratic (Figure 2b). So in order to achieve a near-linear running time, our algorithm uses efficient data structures to implicitly encode the boundary of the part of the reachable region that has already been constructed. This data structure is used to propagate a boundary path in $O(\log n)$ time, if it follows a previously constructed boundary. More details can be found in Section 6.

Comparison with previous work. The most directly related problem is to find a descending path (that is, a path that never goes up) between two points on a terrain. This is a special case of reachability under direction constraints: After projecting the terrain to horizontal, we get an instance of our problem where each face has a cone of possible directions which is a halfplane. De Berg and Van Kreveld [5] gave a data structure that can answer descending path reachability queries between two points in $O(\log n)$ time, after $O(n \log n)$ preprocessing time. Another related paper [4] shows how to compute a collection of n paths of steepest

descent in $O(n \log n)$ time [4]. This work uses a data structure similar to our data structure for recording beams (Section 6), but it uses it in a different way as it proceeds by sweeping a horizontal plane over the whole terrain. This approach cannot be applied to our problem, as there is no notion of elevation. Recently, Cheng and Jin [2] gave the first FPTAS for finding a shortest descending path between two points.

The problem of planning the movement in the presence of a flow was studied by Reif and Sun [6]. A point robot can apply a control velocity, with bounded norm, and each face of the triangulation has a flow of constant velocity (Figure 1b). They give an approximation algorithm for finding a shortest path between two points. However, it only applies when the control velocity is larger than the flow velocity, meaning that all directions of travel are possible. [1] Hence, their algorithm cannot be used to solve our problem, although it has the advantage of providing an approximate shortest path.

Sun and Reif also considered another anisotropic motion planning problem, where a wheeled robot travels on a terrain [7]. The mechanical constraints such as friction and steepness imply that some directions of travel are forbidden, and the speed depends on the direction. They present an approximation algorithm for the single source shortest path problem. This algorithm places Steiner points along the edges and searches the induced graph. The case where some directions of travel are forbidden is only briefly described [7, Theorem 4] and no time bound is given for this case. In any case, the number of Steiner points depends on several extra parameters, such as the minimum angle in the triangulation, or the length of the longest edge.

Cheng et al. considered approximate shortest path problems in anisotropic environments where the cost function within each face is a convex distance function [3]. This model allows different costs for different directions of travel, but again all directions must be allowed, so it does not solve our problem. They give an approximation algorithm whose running time depends on the ratio between the largest and the smallest speed in any direction, and the dependency on the input size is cubic.

In summary, we propose the first provably efficient algorithm to compute a path between two points in a planar subdivision, when there is one cone of possible directions of travel per face. Previous work on path planning with direction constraints either considered a special case where each cone of direction is a halfplane [2, 5], or did not provide any time bound [6, 7]. Unlike the algorithms in other anisotropic models [2, 3, 6, 7], our algorithm does not return an approximate shortest path, but it handles more general direction constraints, and it runs in near-linear time, regardless of the geometry of the input.

2 Notation and preliminary

Problem statement. We are given a planar triangulation $\mathcal{S}$ with n vertices. More precisely, $\mathcal{S}$ is a simplicial complex in $\mathbb{R}^2$. Each face f of $\mathcal{S}$ is a triangle, and is associated with a cone $C(f)$ of possible directions of travel. This cone is specified by a leftmost (clockwise) and rightmost (counterclockwise) direction $d_\ell(f) \in \mathbb{R}^2$ and $d_r(f) \in \mathbb{R}^2$. We assume that $C(f)$ is convex: Its opening angle is at most π. A direction (or vector) d is in $C(f)$ if $d = \lambda d_\ell(f) + \mu d_r(f)$ for some $\lambda, \mu \geq 0$. In addition, halfplanes bounded by lines through $(0, 0)$ are considered to be cones, as well as the whole plane $\mathbb{R}^2$, which is called the *full* cone of directions.

[1] The algorithm by Reif and Sun [6, Section 5] requires that the parameter ρ_r is larger than 1, which means that the control velocity is always larger than the flow velocity. In other words, all directions of travel are allowed. The speed depends on the direction, but it is always positive.

We denote by $\overline{pq}$ the directed closed line segment from point p to point q. We will abuse notation so that $\overline{pq} \in C(f)$ means that the vector $\overrightarrow{pq}$ is in $C(f)$. A segment $\overline{pq}$ in a face f is *feasible* if $\overline{pq} \in C(f)$. Let s and t be two points in this subdivision. A *feasible path* from s to t is a polyline whose edges are feasible segments. Given a starting point s, our goal is to find all the vertices of $\mathcal{S}$ that can be reached by a feasible path.

Model of computation. We assume that, given a point p on the boundary of a face f of $\mathcal{S}$, we can compute in constant time the points q and q' along the boundary of f that are in directions $d_\ell(f)$ and $d_r(f)$. In addition, we assume that we have at our disposal a constant-time logarithm function. It will help us handle spirals efficiently: We will be able to compute in $O(\log n)$ time the exit point of a spiral (Figure 11).

Notation. An *interval* is a closed segment along an edge of $\mathcal{S}$. We allow an interval to be a single point in the interior of an edge of $\mathcal{S}$, but vertices of $\mathcal{S}$ are not called intervals. So any interval u is contained in a unique edge of $\mathcal{S}$. This edge is denoted by $\text{edge}(u)$. A *full interval* is an edge of $\mathcal{S}$. In other words, a full interval u is such that $\text{edge}(u) = u$. An *extreme interval* is an interval $\overline{vq}$ such that v is a vertex of $\mathcal{S}$. In particular, a full interval is an extreme interval with respect to both of its endpoints. A *free interval* is an interval that is not extreme. In other words, a free interval is contained in the interior of an edge of $\mathcal{S}$.

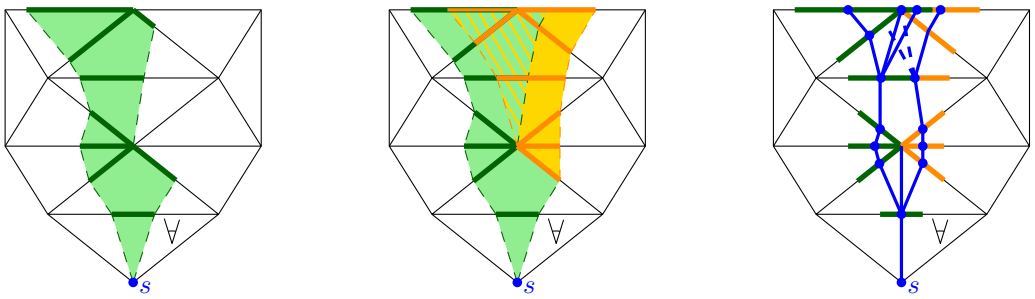
Let u be an interval on the boundary of a face f. We denote by $R(u, f)$ the set of points on the boundary of f that are reachable from u along a direction in $C(f)$. More precisely, $R(u, f)$ is the intersection of $u + C(f)$ with the boundary of f. An *image vertex* of (u, f) is a vertex of f that lies in $R(u, f)$. An *image interval* of (u, f) is a maximal segment of $R(u, f)$. In degenerate cases, an image interval can be a single point in the interior of an edge, but we do not count vertices of $\mathcal{S}$ as image intervals. The list of image vertices and intervals is denoted by $L(u, f)$. An interval u is *non-propagating* if $R(u, f) \subseteq u$. Otherwise, it is *propagating*. We may also say that u *propagates* into f.

Let π be a feasible path from s to t. As the cones of directions are convex, we can replace any subpath of π contained in a face f with a single edge $\overline{pq}$. So a path π can be specified by a sequence $p_1 f_1 p_2 f_2 \dots f_\ell p_{\ell+1}$ where $\overline{p_i p_{i+1}} \in C(f_i)$ and $f_i \neq f_{i+1}$ for all i. As this is the only relevant type of path for our problem, in order to alleviate notation, we will assume that all paths are of this form.

3 Overview

In this section, we present a brief description of our results and the approach used to prove them. We start with the algorithms.

The naive approach would be to compute the whole reachable region block by block, by recursively propagating intervals along the boundary of the faces of the subdivision. (See Figure 3.) One difficulty with this approach is that the blocks partially overlap, so the algorithm would do a lot of double-work, and it is not clear how to use planarity arguments in the analysis. So instead of constructing the whole reachable region, we construct the *skeleton* Ske (Figure 3c), which is a tree connecting the midpoints of the reachable intervals. While constructing this tree, we will prove that any new edge that crosses a previously constructed edge can be *pruned* without affecting the set of reachable nodes computed by the algorithm. Hence, we can ensure that the skeleton is a tree properly embedded in the plane, and we will be able to use planarity arguments in our proofs. For instance, it is easy to see that the



(a) (b) (c)

Figure 3 (a) Naive approach to compute the reachable region block by block. (b) After propagating another branch, some blocks overlap. (c) Our approach using the skeleton (thick, blue). Two edges are pruned (dashed), so that the skeleton remains a tree embedded in the plane.

skeleton can have only one branching node (i.e. degree at least 3) within each face of the subdivision, and thus the skeleton has a linear number of branchings.

We present in Section 4 a description of our first algorithm (Algorithm 1) to construct the skeleton edge by edge. In Section 5, we prove several properties of the skeleton constructed by Algorithm 1. Algorithm 1 is inefficient because it could enter an infinite loop when it encounters a spiral (Figure 2a), and even without spirals, the tree could have a quadratic number of edges (Figure 2b). Note that in both cases, the issue arises from long paths that cross the same sequence of edges: in the case of a spiral, the subsequence is repeated periodically, and in the second example, we have long, horizontal paths crossing the same sequence of edges.

We present in Section 6 an algorithm (Algorithm 2) that overcomes this difficulty using efficient data structures for handling *parallel beams*, that is, paths in the skeleton that cross the same sequence of edges of the triangulation. The idea is the following. Consider a path that crosses the sequence of edges $(e_1, e_2, \ldots, e_m)$ in their interiors. If x_1 is the coordinate of the path along e_1, then the coordinate x_m along e_m is given by a linear map, whose coefficients can be easily determined from the geometry of the faces and the cones of directions. We record these linear functions in a binary tree over $(e_1, \ldots, e_m)$, so that we can implicitly construct in $O(\log m)$ time any path through a subsequence $(e_i, \ldots, e_j)$, given its starting point x_i.

We record such data structures for the left side and the right side of each beam. (See Figure 4.) When a new beam B follows parallel and to the right of an already constructed beam C, it forms a new *tunnel*, which is the empty region between these two beams. We update the data structure in $O(\log m)$ time by first appending a subtree associated with C to the data structure for B. Then we create a single node for the tunnel, which is sufficient for our purpose, as any new beam entering the tunnel can only go parallel to B and C within this tunnel.

As we shall see, Algorithm 2 runs in $O(n \log n)$ time using this data structure. For instance, in the example of Figure 2b, the data structure for each one of the $\Theta(n)$-long horizontal beams can be constructed in $O(\log n)$ time, given the data structure for the beam immediately above it. The analysis relies on several observation on the structure of the skeleton. For instance, we show that it has $O(n)$ maximal tunnels and maximal beams, and that the total number of nodes in our data structures is $O(n)$. Spirals are handled in the same way as tunnels, as they can be seen as a special type of tunnels.

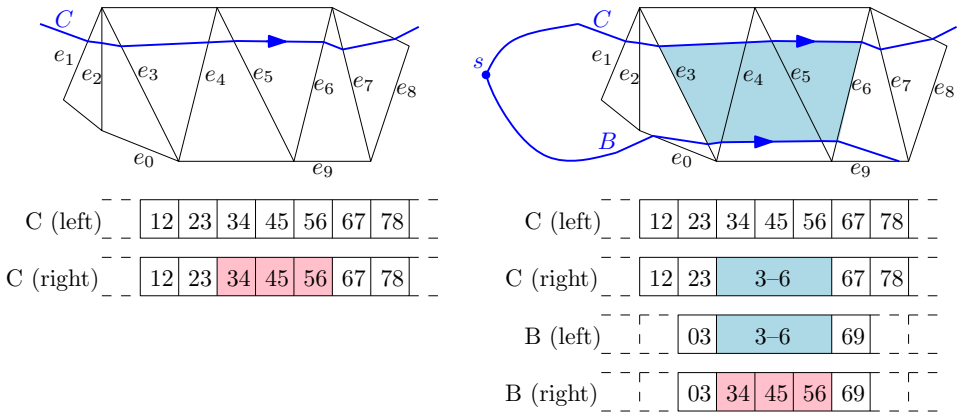


Figure 4 Data structure for beams. One node is created for the right side of C and the left side of B to represent the tunnel (blue). A subtree of the data structure for the right side of C (red) is deleted and inserted into the data structure for the right side of B.

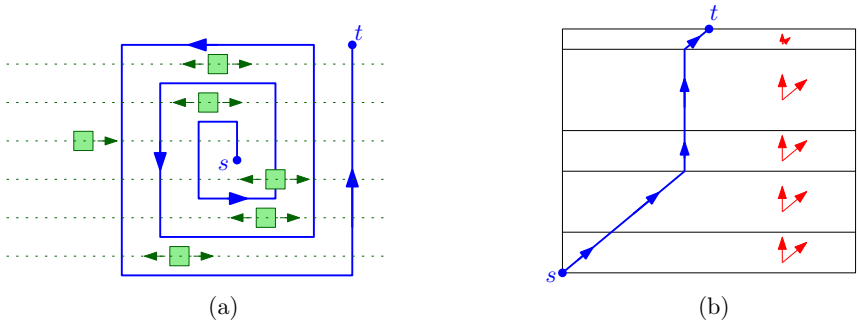


Figure 5 (a) $\Omega(n \log n)$ lower bound. (b) NP-hardness proof.

Finally, we show two hardness results. (Detailed proofs are omitted due to space limitation.) We first prove an $\Omega(n \log n)$ lower bound on our problem using Ben-Or's technique [1]. Figure 5a gives an outline of our construction: The direction constraints force the path to follow a spiral with $\Theta(n)$ edges. Then we place $\Theta(n)$ obstacles that can move left or right, so that the target point t can only be reached if no obstacle overlaps with the spiral. This problem has $n^{\Theta(n)}$ connected components, and hence it requires an algebraic computation tree of depth $\Omega(n \log n)$. Then we prove that the reachability problem where two cones of directions per face are allowed is NP-hard. In fact, our construction only requires that each face allows the directions $(0, 1)$ and $(1, 1)$. Our proof is a reduction to the partition problem: Given a set of integers, can it be partitioned into two subsets with same sum? The reduction is given in Figure 5b. The heights of the rectangles are the input integers, and the target point t is at the midpoint of the top edge. It can only be reached if the instance of the partition problem is positive.

4 First algorithm

In this section, we present our first algorithm (Algorithm 1), which recursively propagates reachable intervals or vertices to other reachable intervals or vertices that lie on the boundary of the same face. As Algorithm 1 constructs intervals one by one, it does not terminate if it encounters an infinite spiral. In Section 6, we present a faster version of this algorithm

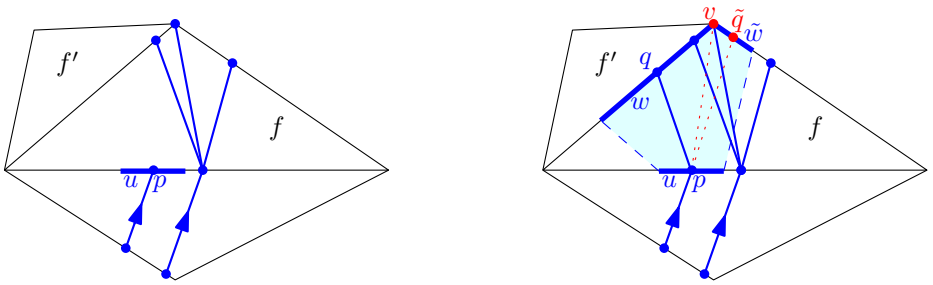


Figure 6 Propagating a pair (u, f), when u is an interval. (Left) The skeleton before propagating (u, f). (Right) When we propagate (u, f), the pair (w, f') is created, and v and $\tilde{w}$ are pruned.

(Algorithm 2) that computes all reachable vertices of $\mathcal{S}$ in $O(n \log n)$ time, using efficient data structures that allow us to implicitly construct a beam or a whole spiral in $O(\log n)$ time.

Algorithm 1 draws a directed tree Ske, called the Skeleton, whose nodes lie on edges of $\mathcal{S}$. In particular, these nodes are either vertices of $\mathcal{S}$ that are found to be reachable by our algorithm, or midpoints of reachable segments of edges of $\mathcal{S}$.

Description. Algorithm 1 *propagates* recursively pairs (u, f), where u is an interval or a vertex on the boundary of a face f. By propagating, we mean that we create children of the pair (u, f) that are of the form (w, f'), where w is an image vertex or interval in $L(u, f)$, and f' is a face other than f. Some of these pairs will be *pruned*, and thus not created. The pairs that have been created, but not yet propagated, are stored in a set $\mathcal{A}$, and are called *active pairs*. After being propagated, a pair is *inactive*. Each pair (u, f) is processed as follows.

First assume that u is an interval. (See Figure 6.) We compute the list $L(u, f)$ of image vertices and intervals. Then for each vertex or interval w in this list, we check whether it needs to be pruned as follows. Let $p = \text{mid}(u)$ and $q = \text{mid}(w)$. We prune w if at least one of the three conditions below is met (Figure 7).

- **(i)** $\overline{pq} \cap \text{Ske} \neq \{p\}$.
- **(ii)** u and w are intervals, and $\text{edge}(u) = \text{edge}(w)$.
- **(iii)** u and w are intervals, $\text{edge}(u) \neq \text{edge}(w)$, and there is a node q_0 of Ske in the interior of $\text{edge}(w)$ such that $\overline{q_0 q} \in C(f_0)$, where f_0 is the face other than f bounded by $\text{edge}(w)$.

Condition (i) ensures that Ske has no self-crossing and no cycle. Condition (ii) will be needed in Algorithm 2 (Section 6), so that we do not need to propagate an interval backwards. Condition (iii) will also be needed in Algorithm 2, in order to ensure that the skeleton cannot enter two-way tunnels. (See Lemma 7.) We will prove later that our pruning scheme is correct in the sense that, when Algorithm 1 terminates, it always outputs all the reachable vertices. However, for some input, it may not terminate, as it may enter infinite spirals, and thus some reachable vertices may never be visited.

If w is not pruned, we insert $\overline{pq}$ into Ske. Then we create all pairs (w, f') such that $w \subset f'$ and $f' \neq f$, and we insert them into the set $\mathcal{A}$ of active pairs. If w is an interval, there is at most one such face f'.

Now suppose that u is a vertex (Figure 8). We still compute the list $L(u, f)$ of image vertices and intervals. If $L(u, f)$ contains an interval w along the edge opposite to u, we apply pruning condition (i). So we prune w if $\overline{pq} \cap \text{Ske} \neq \{p\}$. Again, if w is not pruned, we insert $\overline{pq}$ into Ske, and insert into $\mathcal{A}$ the pair (w, f') if f is adjacent to a face f' along $\text{edge}(w)$. Then we handle each vertex $v \in L(u, f)$ such that $v \neq u$ (if any) as follows. Let

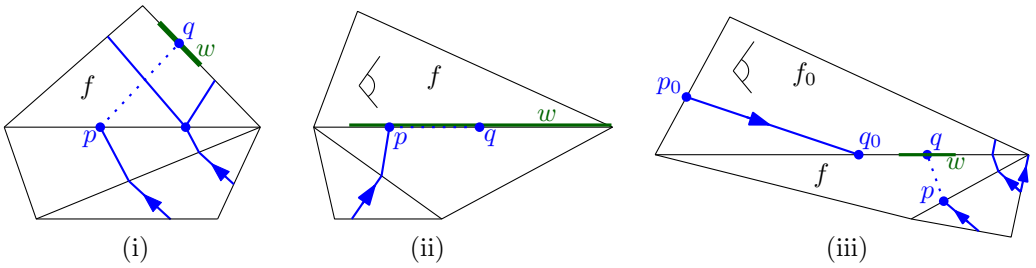


Figure 7 The three pruning conditions. The interval w corresponding to q is pruned, and hence the edge $\overline{pq}$ (dotted) is not constructed.

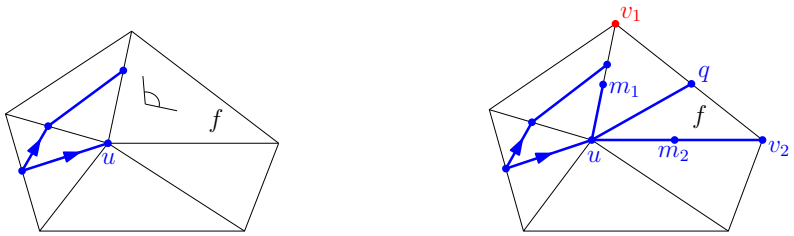


Figure 8 Propagating a pair (u, f), when u is a vertex. (Left) The skeleton before dequeuing (u, f). (Right) When we propagate (u, f), only v_1 is pruned, and the edges $\overline{uq}$, $\overline{um_1}$, $\overline{um_2}$ and $\overline{m_2v_2}$ are inserted into Ske.

m denote the midpoint of $\overline{uv}$. We apply pruning condition (i), so if $\overline{um} \cap \text{Ske} \neq \{u\}$, we prune $\overline{uv}$, and we are done with v. On the other hand, if $\overline{uv}$ is not pruned, then we insert the edge $\overline{um}$ into Ske, and if there is a face f' adjacent to f along $\overline{uv}$, we insert into $\mathcal{A}$ the pair $(\overline{uv}, f')$.

After this, still assuming that $\overline{uv}$ was not pruned, we try to extend the skeleton Ske further to v. So we apply pruning condition (i) to $\overline{mv}$. If $\overline{mv} \cap \text{Ske} \neq \{m\}$, we prune v. Otherwise, we insert the edge $\overline{mv}$ into Ske, and we insert into $\mathcal{A}$ the pair (v, f') for each face $f' \neq f$ that is adjacent to v.

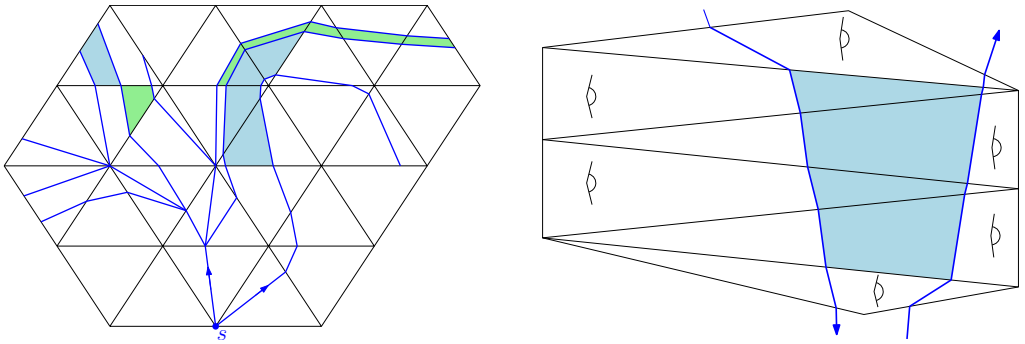
Proof of correctness. We can prove that Algorithm 1 is correct in the following sense:

▶ **Theorem 1.** *If Algorithm 1 terminates, then the reachable vertices are nodes of* Ske.

The condition that the algorithm terminates is necessary, as otherwise it could extend a spiral indefinitely and then some reachable vertices would not be visited. The proof is omitted due to space limitation. The idea is to show that, whenever we prune a vertex or an edge, which results in a pair (w, f') not being created, then some other pair (w', f') must have been created earlier such that $R(w, f') \subset R(w', f')$. It means that (w, f') can be safely pruned. Our proof is based on proving this type of invariants carefully through case analysis. It also uses the property below.

▶ **Lemma 2.** *At any time during the execution of Algorithm 1,* Ske *is a tree embedded in the plane.*

Proof. Pruning condition (i) ensures that Ske does not contain any cycle or crossing edges. When propagating a pair (u, f) where u is an interval, pruning condition (ii) ensures that the new node q is different from $p = \text{mid}(u)$. When u is a vertex of $\mathcal{S}$, then our algorithm does not attempt to propagate u into itself, so it does not create a duplicate node either. ◀



(a) (b)

Figure 9 (a) Four one-way tunnels (shaded). (b) A two-way tunnel. All these tunnels are maximal.

5 Properties of the skeleton

In this section, we consider the properties of the skeleton Ske, at any time during the execution of Algorithm 1. These properties will be needed in Section 6, in order to analyze and prove correctness of Algorithm 2. The proofs of the lemmas in this section are omitted due to space limitation.

A *branching node* of Ske is a node with outdegree at least 2. An edge of Ske that is incident to a branching node or a vertex of $\mathcal{S}$, is called a *special* edge. The other edges of Ske are called *transversal* edges. Hence, a transversal edge connects two points in the interiors of two different edges of $\mathcal{S}$, and is incident to at most one other edge at each endpoint. A *beam* is a subpath made of transversal edges. If it is not contained in any other beam, we say that it is a *maximal beam.*

▶ **Lemma 3.** *There are $O(n)$ branching nodes in* Ske.

▶ **Lemma 4.** *There are $O(n)$ special edges and maximal beams in* Ske.

The *combinatorial structure* of a transversal edge from a point in an edge e of $\mathcal{S}$ to a point along another edge e' is the pair (e, e'). Similarly, the combinatorial structure of a beam is the sequence of edges of $\mathcal{S}$ that it traverses. Two beams are *parallel* if they have the same combinatorial structures. A *one-way tunnel* is formed by two parallel beams such that there is no other edge of Ske in the space delimited by the two beams and the first and last edge of $\mathcal{S}$ that they meet. (Figure 9.) A *two-way tunnel* is analogous, but one sequence is equal to the other sequence reversed. A *tunnel* is either a one-way tunnel or a two-way tunnel. A tunnel is *maximal* if it is not contained in any other tunnel.

We now show that there is a linear number of maximal tunnels. We prove it by charging tunnels to either special edges, or *corners*: A corner of a face f is a pair (v, f) where v is a vertex of f. (See Figure 10.)

▶ **Lemma 5.** *At any time during the course of Algorithm 1, there are $O(n)$ maximal tunnels.*

A *spiral* is a one-way tunnel that is bounded by the same beam B on both sides. Therefore, the beam B, or its subsequence that bounds the spiral, is periodic: it has combinatorial structure $(e_1, e_2, \ldots, e_\ell)$ with $e_{i+p} = e_i$ for all $1 \leq i \leq i + p \leq \ell$ and some $p < \ell$. If a spiral can be extended indefinitely, that is, we can extend the sequence $(e_1, e_2, \ldots, e_\ell)$ into an arbitrarily long sequence with period p, then we say that the spiral is an *infinite spiral.*

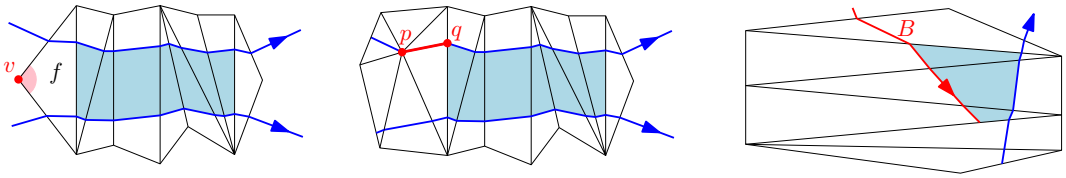


Figure 10 Proof of Lemma 5. (Left) The tunnel (shaded) is charged to the corner (v, f). (Middle) The tunnel is charged to the special edge $\overline{pq}$. (Right) The tunnel is charged to beam B.

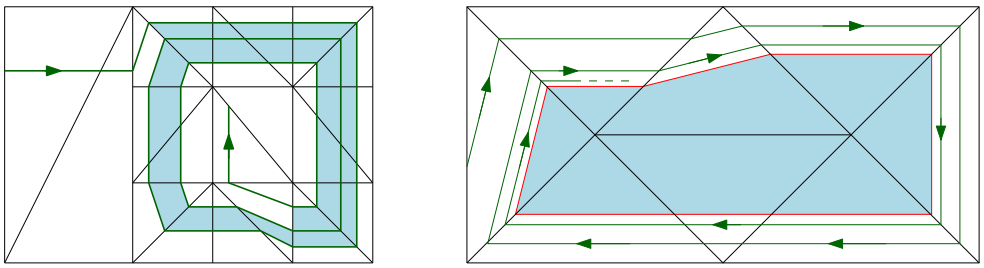


Figure 11 (Left) A finite spiral (shaded). (Right) An infinite spiral that does not converge to a vertex, as it never enters the shaded region.

Otherwise, we say that it is a *finite spiral*. Figure 2a shows an infinite spiral that converges to a single vertex. Figure 11 shows a finite spiral, and an infinite spiral that does not converge to a vertex.

We will see later that our algorithm handles finite spirals in the same way as a regular tunnel (that is, a tunnel which is not a spiral). Infinite spirals behave differently, but the lemma below shows that any beam that enters an infinite spiral cannot exit it, and hence Algorithm 2 will interrupt this beam. (See Figure 12a.)

▶ **Lemma 6.** *Suppose that Algorithm 1 extends an edge $\overline{pq}$ of the skeleton* Ske *inside an infinite spiral. Then the subtree of* Ske *rooted at p is a single beam, which remains within this spiral.*

The lemma below shows that, while we construct the skeleton, the only way to enter a tunnel is to enter a one-way tunnel from its entrance and towards its exit. It means that one-way tunnels will act as one-way gates, and two-way tunnels will act as barriers during the execution of the algorithm. The proof relies on pruning condition (iii), and is illustrated in Figure 12b.

▶ **Lemma 7.** *During the course of Algorithm 1, no new skeleton edge can be created inside an existing tunnel, except if the tunnel is a one-way tunnel, and the new edge has the same combinatorial type as an edge of the tunnel.*

Left-turn and right-turn function. We introduce the left-turn and right-turn functions associated with the combinatorial structure σ of a beam B. Intuitively, the left-turn function f_σ returns the exit point of a path that, from a given starting point, traverses the edge sequence σ, and turns left as much as possible. Similarly, the right-turn function corresponds to an extreme right-turning path. A more precise description follows.

So let $\sigma = (e_1, \ldots, e_\ell)$ denote the combinatorial structure of the beam B. We denote by $x_1, x_2, \ldots, x_\ell$ the coordinates along these edges. We assume that the edges are oriented in a consistent manner, so that the left hand side of B corresponds to smaller values of x_i and

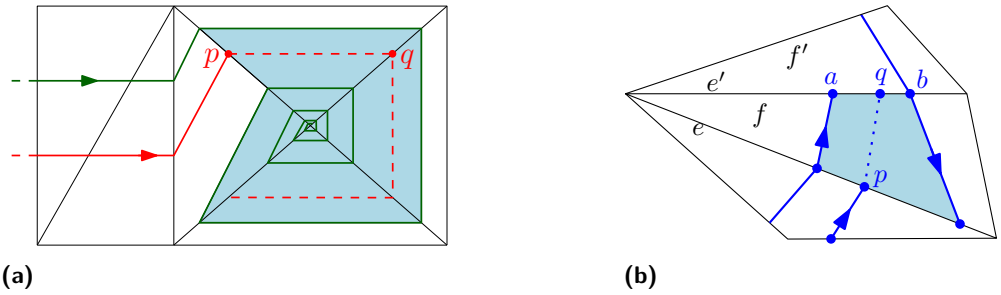


Figure 12 (a) Proof of Lemma 6. The red beam enters an infinite tunnel (shaded), and cannot get out. (b) Proof of Lemma 7. Edge $\overline{pq}$ gets pruned by condition (iii).

the right hand side corresponds to larger values. We assume that the range of x_i is $[0, 1]$, and thus the endpoints of e_i have coordinates $x_i = 0$ and $x_i = 1$.

Then a beam parallel to B is completely defined by its first interval, which lies along e_1. More precisely, we denote by $[a_i, b_i]$ the interval along e_i corresponding to this beam $B(a_1, b_1)$. We assume without loss of generality that $x_i(a_i) \leq x_i(b_i)$, that is, a_i lies on the left-hand side of B. Then $x_\ell(a_\ell)$ is a piecewise linear function $f_\sigma(x_1)$ of x_1, with at most one flat patch and one non-flat patch. More precisely, f_σ is given by two coefficients a_σ and b_σ and an interval $[c_\sigma, d_\sigma]$. The values of $a_\sigma, b_\sigma, c_\sigma, d_\sigma$ depend on the shapes and the cones of directions of the faces spanned by σ. When $x_1 \in [c_\sigma, d_\sigma]$, we have $f_\sigma(x_1) = a_\sigma x_1 + b_\sigma$. When $x_1 \leq c_\sigma$, then $f_\sigma(x_1) = f_\sigma(c_\sigma)$. When $x_1 > d_\sigma$, then there is no beam corresponding to this value of x_1.

Similarly, we define the right-turn function g_σ that gives the right endpoint of the interval along e_ℓ as a function of $x_1(b_1)$. Given a beam whose combinatorial structures σ is the concatenation $\sigma_1.\sigma_2$ of two sequence σ_1 and σ_2, the functions f_σ and g_σ can be determined in constant time, given the functions f_{σ_1}, g_{σ_1}, f_{σ_2} and g_{σ_2}.

6 Faster algorithm

In this section, we present an $O(n \log n)$ time algorithm (Algorithm 2) to compute all the vertices that are reachable from s. Algorithm 2 is based on Algorithm 1, and the speed-up comes from the fact that a new beam parallel to an existing beam can be implicitly constructed in logarithmic time, using appropriate data structures. The differences with Algorithm 1 are the following.

- The skeleton is built in a depth-first manner, always starting with the leftmost subtree, and going from left to right at each branching. So Algorithm 2 starts by constructing a leftmost turning path of Ske, and it constructs each maximal beam in one go.
- When constructing a new transversal edge, if an edge with the same combinatorial structure has been constructed earlier, the new beam follows parallel to a previously constructed beam, forming a new tunnel. Then a procedure called Construct Tunnel extends the new beam in one go, for as long as this tunnel can be extended. We will see that this procedure can be implemented to run in $O(\log n)$ time.
- If a beam begins to form an infinite spiral, or enters one, then it is interrupted, that is, the corresponding active pair never gets propagated. When all active pairs correspond to infinite spirals, Algorithm 2 halts.

On the other hand, similarly as in Algorithm 1, a special edge, or a transversal edge such that no other edge with the same combinatorial structure has been constructed before,

is constructed in constant time. As there are $O(n)$ different combinatorial structures for transversal edges, this contributes $O(n)$ to the running time. We will argue that CONSTRUCT TUNNEL is called only $O(n)$ times, and hence overall running time is $O(n \log n)$.

6.1 Data structures for beams

Let B be a beam with combinatorial structure $\sigma = (e_1, \ldots, e_m)$. Then we represent the left-turn and right-turn functions of any subsequence of σ using two balanced binary trees $T_\ell(B)$ and $T_r(B)$. These trees could be, for instance, Red-Black trees, as we will need to be able to perform insertion, deletion, join and split operations in $O(\log m)$ time [8]. We also record the first interval $[a_1, b_1]$ of B, that is, the interval of e_1 that was propagated to create the beam B.

First suppose that B does not bound any tunnel. Then $T_\ell(B)$ and $T_r(B)$ are two copies of the same binary tree representing the subsequences of σ in a hierarchical manner, and recording both the left-turn and the right-turn functions. Each node corresponds to a subsequence $\sigma_{ij} = (e_i, \ldots, e_j)$ of σ, and records the two indices i and j, as well as the functions $f_{\sigma_{ij}}$ and $g_{\sigma_{ij}}$. The root corresponds to the sequence σ of the whole beam B, and the leaves correspond to the sequences $\sigma_{i(i+1)}$ where $1 \leq i \leq m-1$. The subsequence corresponding to an internal node is the concatenation of the sequences stored at its children, which are consecutive.

If the beam B bounds a tunnel on its left side, starting from edge e_i to edge e_j, we replace the subtree of $T_\ell(B)$ corresponding to $\sigma_{ij} = (e_i, e_{i+1}, \ldots, e_j)$ with a single node that records the functions $f_{\sigma_{ij}}$ and $g_{\sigma_{ij}}$. If B bounds several tunnels on its left side, we do the same for each tunnel, as they correspond to disjoint subsequences of σ. The tree $T_r(B)$ is constructed in a similar way, except that it deals with tunnels on the right side of B.

Each tunnel is bounded by two parallel beams B and C, each one of them being recorded as one node in our data structure. Suppose that B lies on the left and C lies on the right of the tunnel. Then the node of $T_r(B)$ and the node of $T_\ell(C)$ corresponding to this tunnel record a pointer to each other.

As each node of a beam is the midpoint of the corresponding left- and right-turning path, this data structure allows us to implicitly trace a new beam that appears immediately to the left or to the right of B, for as long as it remains parallel to B, in logarithmic time.

If a beam B bounds an infinite spiral, then the infinite periodic subsequence is represented by a single node that we call a *sink* node. By Lemma 6, a beam that enters an infinite spiral cannot get out, so our algorithm will stop extending such beams when it encounters a sink node.

6.2 Procedure Construct Tunnel

Suppose that we are extending a beam B, and the next transversal edge to be constructed in this beam is $\overline{pq}$. Assume that an edge with the same combinatorial structure has been constructed earlier. Then there should exist a beam C adjacent to $\overline{pq}$. Without loss of generality, we assume that C lies to the right of $\overline{pq}$, and that it has combinatorial structure $\sigma = \gamma.(e_1, \ldots, e_m)$, where e_1 is the edge containing p.[2] The procedure CONSTRUCT TUNNEL will consider a constant number of cases below, which can be handled in logarithmic time using our data structure.

[2] When several beams cross e_1, we will show in the full version of this paper how to identify the beam C: we will use a pointer from each vertex v of $\mathcal{S}$ to the closest beam crossing each edge incident to v.

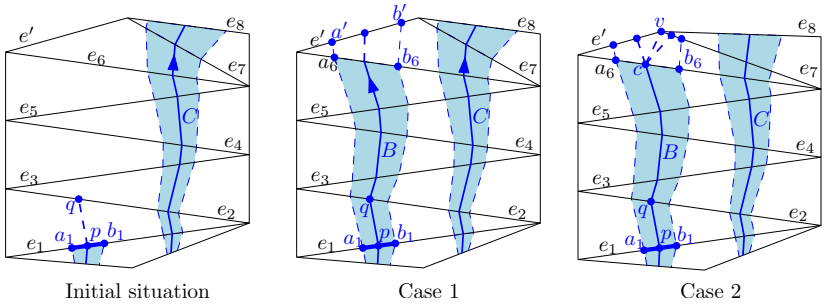


Figure 13 Result after calling Construct Tunnel once in Cases 1 and 2.

▶ **Lemma 8.** *Let N denote the total number, over all beams D constructed during the course of the algorithm, of nodes in the trees $T_\ell(D)$ or $T_r(D)$. The procedure* Construct Tunnel *runs in $O(\log N)$ time.*

We now describe the procedure Construct Tunnel.

Case 1. In this case, we assume that B follows the left side of C, until B and C split at some edge e_i. That is, the combinatorial structure of B is $\alpha.(e_1, \ldots, e_i, e')$ where $e' \neq e_{i+1}$, and α is the sequence of edges crossed by B before it reaches e_1. (See Figure 13.) So the goal is to identify this index i, and to update the data structures accordingly.

In order to simplify the presentation, we first assume that C did not bound any tunnel before we started to expand B on its left side. Let a_i and b_i denote the ith vertex along the left- and right-turning paths of B, starting from a_1 and b_1, respectively. We now want to find the last index i such that a_i and b_i lie on e_i. Our data structures $T_\ell(C)$ and $T_r(C)$ allows us to find it in $O(\log N)$ time as follows. Without loss of generality, we only consider a_i and use $T_\ell(C)$.

We find the last index i by traversing $T_\ell(C)$ from the leaf recording (e_1, e_2) towards the root, and going then going down towards the leaf recording (e_i, e_{i+1}). (Essentially, we are doing exponential search.) The left-turn function (e_1, e_2), together with the coordinates of a_1, allow us to determine whether $a_2 \in e_2$—more precisely, we check whether a_1 is in the domain of $f_{(e_1,e_2)}$. If not, then we are done. Otherwise, we move to the parent of the node recording (e_1, e_2). If (e_1, e_2) was a left child, then its parent records (e_1, e_2, e_3) or (e_1, e_2, e_3, e_4), so we use the corresponding left-turn function to determine whether $a_3 \in e_3$ or $a_4 \in e_4$. If (e_1, e_2) is a right child, then we move to the node at the same level and immediately to the right of its parent. This node records the sequence $(e_2, \ldots, e_j)$ where $j \leq 6$, and we can determine in constant time from the position of a_2 computed earlier whether a_j lies on e_j. We repeat this process $O(\log N)$ time, until it fails. Then we know that (e_i, e_{i+1}) is stored at a descendent of the current node, and we find it by traversing the tree downwards.

After finding this last index i, we cut from $T_\ell(C)$ the nodes corresponding to the sequence $(e_1, \ldots, e_i)$ and append them to $T_\ell(B)$. This can be done by performing two split and two join operations, which takes $O(\log N)$ time using Red-Black trees [8]. We then insert a single node into $T_\ell(C)$ that records the functions $f_{(e_1,\ldots,e_6)}$ and $g_{(e_1,\ldots,e_6)}$, whose coefficients we can also compute in $O(\log N)$ time using our data structure. Then we make a copy of this node and append it to $T_r(B)$. Finally, we record in each of these two nodes a pointer to the other. So overall, we have updated the data structures for B and C in $O(\log N)$ time.

The procedure above still applies when beam C bounds one or several tunnels on its left side which are then split by B. The nodes of $T_\ell(C)$ corresponding to these tunnels are moved

to $T_\ell(B)$ in the same way as the other nodes, which represent single edges of the skeleton. The cross-pointers between these nodes do not need to be updated. Thus, the data structure can be updated in $O(\log N)$ time.

Case 2. In this case, we assume that B follows the left side of C, until B reaches an edge e_i and branches. This case is similar to Case 1, except for the termination condition. In Case 1, the beam B was extended until it quit following C from the left. In Case 2, we extend B until it branches, and hence the left- and right-turning paths of B must be separated by a vertex v of the subdivision $\mathcal{S}$. (See Figure 13.) This vertex v can be found in $O(\log N)$ time using the data structures $T_\ell(C)$ and $T_\ell(B)$, which allow us to trace the left- and right-turning paths from a_1 and b_1. After v has been found, we know that B stops at the edge e_i opposite from v, and we update the data structures in the same way as in Case 1.

Case 3. In this case, we assume that B follows the left side of C, until B is interrupted because its next tentative edge gets pruned. So we assume that this edge $\overline{p_1q_1}$ has combinatorial structure (e_1, e_2), and it gets pruned due to an edge $\overline{p_2q_2}$ of Ske. We denote by e' the third edge of the face bounded by e_1 and e_2.

First assume that $p_2 \in e_1$ and $q_2 \in e_2$. Then we can prove that q_1 and q_2 must be at the midpoint of an edge of $\mathcal{S}$. We will show in the full version of this paper how to identify this case in constant time by augmenting the subdivision $\mathcal{S}$.

Now assume that $\overline{p_2q_2} \subset e_2$. In this case, $\overline{p_2q_2}$ is an edge of $\mathcal{S}$, so it can be identified in $O(\log N)$ time as in Case 1.

Suppose that $p_2 \in e_2$ and $q_2 \in e_1$. Then C forms a reversed tunnel with the beam D containing $\overline{p_2q_2}$. This tunnel is narrower along e_1, and since $\overline{p_1q_1}$ follows parallel to B, it cannot cross $\overline{p_2q_2}$, a contradiction.

The case where $p_2 \in e_1$ and $q_2 \in e'$ is similar to the case where $p_2 \in e_1$ and $q_2 \in e_2$. The case where $p_2 \in e'$ and $q_2 \in e_2$ is similar to the case where $p_2 \in e_1$ and $q_2 \in e_2$. The case where $p_2 \in e'$ and $q_2 \in e_1$ is similar to the case where $p_2 \in e_2$ and $q_2 \in e_1$. The case where $p_2 \in e_2$ and $q_2 \in e'$ is similar to the case where $p_2 \in e_2$ and $q_2 \in e_1$.

Case 4. In this case, we assume that B follows the left side of C, until we reach the terminal node of C. Our data structure allows us to identify this case in $O(\log N)$, by checking that the new section of B has the same combinatorial structure as C until we reach the end of C. If the terminal node is a sink, then B enters an infinite spiral and thus we stop extending it.

Case 5. The beams B and C are equal, and hence we start a spiral. So B has a combinatorial structure of the form $\alpha.\beta$, where $\beta = (e_1, e_2, \ldots, e_p)$, before the current call to Construct Tunnel. If the spiral is finite, then we are going to extend it into $\alpha.\beta^k.(e_1, \ldots, e_i)$, where $k \geq 0$ is the number of full turns made by the spiral, and i is the number of edges in the last (partial) turn.

Using our data structure, we compute f_σ and g_σ, which are linear functions. So $(f_\sigma)^k(a_1)$ and $(f_\sigma)^k(b_1)$ are geometric progressions, whose expression can be determined in constant time. If $x_1(a_1)$ is not in the domain of $(f_\sigma)^k$ for some k, then the spiral is finite, and we can find in constant time the first such index k, which is the number of full turns of the spiral. Similarly, we can find whether $x_1(b_1)$ leaves $(g_\sigma)^k$ for some k.

If the spiral is finite, then we append a node to $T_r(B)$ and $T_\ell(C)$ corresponding to the sequence $\beta^k.(e_1, \ldots, e_i)$ of the spiral. If the spiral is infinite, we stop propagating B, and append a sink node at the end of $T_\ell(B)$ and $T_r(B)$.

Case 6. In this case, we assume that B and C form a two-way tunnel, starting from $\overline{pq}$. For any node of $T_\ell(C)$ or $T_r(C)$ that corresponds to a sequence σ, we know the description of the left- and right-turn functions f_σ and g_σ. So we can also get in constant time their inverses f_σ^{-1} and g_σ^{-1}. Therefore, we can follow beam C backwards in the same way as in Cases 1 to 4, and Case 6 can be handled in the same way.

6.3 Main result

The main result of this paper is the theorem below. Due to space limitation, its proof is omitted.

▶ **Theorem 9.** *The vertices that are reachable from s are nodes of the skeleton computed by Algorithm 2. Therefore, we can compute all the reachable vertices in $O(n \log n)$ time.*

References

1 Michael Ben-Or. Lower bounds for algebraic computation trees. In *Proc. 15th ACM Symposium on Theory of Computing*, pages 80–86, 1983. doi:10.1145/800061.808735.

2 Siu-Wing Cheng and Jiongxin Jin. Approximate shortest descending paths. In *Proc. 24th ACM-SIAM Symposium on Discrete Algorithms*, pages 144–155, 2013.

3 Siu-Wing Cheng, Hyeon-Suk Na, Antoine Vigneron, and Yajun Wang. Approximate shortest paths in anisotropic regions. *SIAM Journal on Computing*, 38(3):802–824, 2008.

4 Mark de Berg, Herman J. Haverkort, and Constantinos P. Tsirogiannis. Implicit flow routing on terrains with applications to surface networks and drainage structures. In *Proc. 22nd ACM-SIAM Symposium on Discrete Algorithms*, pages 285–296, 2011.

5 Mark de Berg and Marc J. van Kreveld. Trekking in the alps without freezing or getting tired. *Algorithmica*, 18(3):306–323, 1997. doi:10.1007/PL00009159.

6 John H. Reif and Zheng Sun. Movement planning in the presence of flows. *Algorithmica*, 39(2):127–153, 2004. doi:10.1007/s00453-003-1079-5.

7 Zheng Sun and John H. Reif. On finding energy-minimizing paths on terrains. *IEEE Transactions on Robotics*, 21(1):102–114, 2005. doi:10.1109/TRO.2004.837232.

8 Robert Endre Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.