Simple Wriggling is Hard unless You Are a Fat Hippo

Irina Kostitsyna* Valentin Polishchuk†

Abstract

We prove that it is NP-hard to decide whether two points in a polygonal domain with holes can be connected by a wire. This implies that finding any approximation to the shortest path for a long snake amidst polygonal obstacles is NP-hard. On the positive side, we show that snake's problem is "length-tractable": if the snake is "fat", i.e., its length/width ratio is small, the shortest path can be computed in polynomial time.

1 Introduction

The most basic problem in VLSI and printed circuit board design is to connect two given points, s and t, by a shortest "thick" path avoiding a set of polygonal obstacles in the plane. The quarter-of-a-century-old approach to the problem is to inflate the obstacles by half the path width, and search for the shortest s-t path amidst the inflated obstacles [9]. The found path, when inflated, is the shortest thick s-t path.

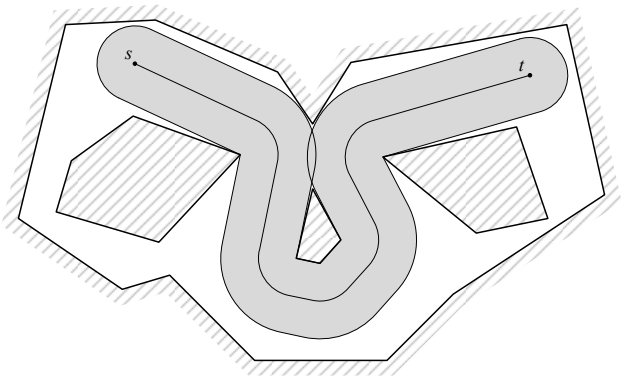

Figure 1: Only a self-overlapping thick path exists.

It went almost unnoticed that the thick path built by the above procedure may self-overlap (Fig. 1): apart from our recent work on thick paths [4], we only found one mention of the possibility of the overlap — Fig. 4 in [15]. (In a different context, Bereg and Kirkpatrick [8, Fig. 2] also noted that Minkowski sum of a disk and a path may be not simply-connected.) When the path represents a thick *wire* connecting terminals on a VLSI chip or on a circuit board, self-overlap is undesirable as the wire must retain its width throughout. Thus, the objective in the basic wire routing problem should be to find the shortest *non-selfoverlapping* thick path.

The problem shows up in other places as well. For instance, one may be interested in the optimal conveyor belt design: the belt is a non-selfoverlapping thick path. Our particular motivation comes from air traffic management where thick paths represent lanes for air traffic. Lane thickness equals to the minimum lateral separation standard, so that aircraft following different lanes stay sufficiently far apart to allow for errors in positioning and navigation. If an airlane self-overlaps, two aircraft following the lane may come too close to each other; thus it is desirable to find lanes without self-overlaps.

*CS Dept, Stony Brook University, Stony Brook, NY 11794, USA.
†Helsinki Institute for Information Technology, CS Dept, University of Helsinki, FI-00014, Finland.

1.1 Our contributions

We prove a surprisingly strong ***negative result*** (Section 4): it is NP-hard even to *decide* whether there exists (possibly, arbitrarily long) s-t wire; this implies that no approximation to the shortest wire can be found in polynomial time (unless P=NP).

Short Snakes Our intractability result means that in general it is NP-hard for a snake to wriggle its way amidst polygonal obstacles (assuming the snake is uncomfortable with squeezing itself). The good news for snakes is that in our hardness proof the sought wire is considerably long; i.e., the hardness of path finding applies only to *long* snakes. Our ***positive result*** (Section 3) is that for a bounded-length snake, the shortest path can be found in polynomial time (assuming real RAM and the ability to solve constant-size differential equations in constant time) by a Dijkstra-like traversal of the domain.

1.2 Related work

In VLSI numerous extensions and generalizations of the basic problem were considered. These include routing multiple paths, on several levels, and with different constraints and objectives. It is impossible to survey all literature on the subject; we will only mention the books [20, 21].

In robotics thick paths were studied as routes for a circular robot. In this context, path self-overlap poses no problem as even a self-overlapping path may be traversed by the robot; that is, in contrast to VLSI, robotics research should not care about finding non-selfoverlapping paths. In [9], Chew gave an efficient algorithm for finding a shortest thick path in a polygonal domain. In a sense, our algorithm for shortest path for a short snake (Section 3) may be viewed as an extension of Chew's.

Motion planning for an object with few degrees of freedom may be approached with the *cell decomposition* techniques [16, 17]. Closest to our bounded-length snake problem is the work on path planning for a *segment* (rod) [5, 6, 18, 23]. Short snakes are also relevant to more recent applications of motor protein motion [10, 24].

2 Snake Anatomy and Physiology

In this section we introduce the notation and formulate our problem.

Let P be an n-vertex polygonal domain with holes, i.e., a connected subset of the plane bounded by straight-line segments; the holes of P are called *obstacles*. For a planar set $\mathcal{S}$ let $\mathrm{bd}\mathcal{S}$ denote the boundary of $\mathcal{S}$, and for $r > 0$ let $< \mathcal{S} >^{(r)}$ denote the Minkowski sum of $\mathcal{S}$ with the radius-r open disk centered at the origin. Let $P^r = P \setminus < \mathrm{bd}P >^{(r)}$ be P offset by r inside. The boundary of P^r consists of straight-line segments and arcs of circles of radius r centered on vertices of P. We call such (maximal) arcs *r-slides*.

Let π be a path within P^1; let $|\pi|$ denote its length. A *thick path* Π is the Minkowski sum $\Pi = < \pi >^{(1)}$. The path π is called the *reference path* of Π; the *length* of Π is $|\pi|$.

A *snake* is a non-selfoverlapping thick path, i.e., a path which is a simply-connected region of the plane. The reference path of the snake is its *spine* (Fig. 2). One of the endpoints of the spine is the snake's *mouth* m. The snake is a "rope" that "pulls itself by the head": imagine that there are little legs (or a wheel, for a toy snake) located at m, by means of which the snake moves. The friction between the snake's body and the ground is high: any point p of the spine will move only when the path from the mouth to p is a "pulled-taut string", i.e., is a *locally shortest* path. That is, the snake always stays pulled taut against the obstacles (or itself).

The input to our problem is the domain P, two points $s, f \in P^1$ – the "start" and the "food", a number $L > 0$ – the length of the snake, and the initial direction of the snake at s. The goal is to find a path for the snake such that the snake's mouth starts at s and ends at f; the constraints are that the snake remains pulled taut and non-selfoverlapping throughout the motion. The objective is to minimize the distance traveled by the mouth, or equivalently, assuming constant speed of motion, the time until the snake reaches the food.

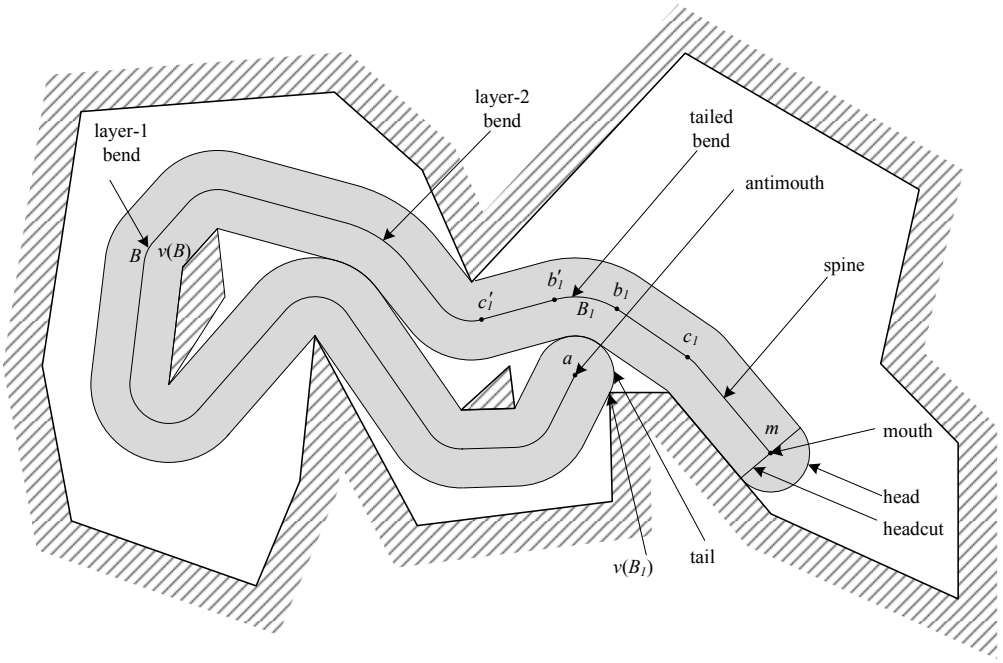

Figure 2: A snake. Mouth, head, headcut, antimouth, tail, and bends are described in Section 3. A layer-2 tailed bend B_1 does not have a point connected to $v(B_1)$ with a length-2 segment fully lying within the snake.

Remark 1. To simplify the formulation, we did not specify the initial configuration of the snake; a pedantic view could be to assume that the snake slithers in from a Riemann sheet glued along the diameter of $< s >^{(1)}$ that is perpendicular to the initial direction of the snake's motion. Our results, both positive and negative, remain valid even if the initial configuration of the snake is part of the input.

Remark 2. For simplicity, we will assume that f is at distance 1 from some vertex p of P and that the snake arrives at the slide of $< p >^{(1)}$ tangentially; this will allow us to discover f "automatically", with no extra effort, in the course of our algorithm in Section 3. Without the assumption, we would have to drop tangents from f onto the track of the snake's head while pulling the snake through P; as soon as a tangent is obstacle-free, the snake can move to f directly along the tangent (as we show, the trajectory of the head consists of a polynomial number of pieces; thus, dropping the tangents would still keep the running time of the algorithm polynomial).

Remark 3. It is not true that all points of the spine necessarily follow the same path.

Remark 4. Whoever guesses that the above model of a snake was developed just for FUN, is right. Nevertheless, the proposed problem formulation may be relevant also in more serious circumstances. It models, e.g., the path of a rope being pulled by its frontpoint through a polygonal domain. If it is a fire hose or a tube delivering life-saving medicine [3, 11], minimizing the time to reach a certain point seems like a natural objective (more important than, say, the work spent on pulling the tube). For another application, consider a chain of robots moving amidst obstacles. Each robot, except for the leader of the chain, has very simple program of following its predecessor – just keeping the distance to it. Then the robots form (approximately) a pulled taut thick string.

Remark 5. It is possible to come up with problem instances in which no path for a pulled-taut snake exists, while a path for a snake that is not required to be taut, does.

3 Shortest Path for a Fat Hippo

When the snake is relaxed and its spine is a straight-line segment, the snake is the Minkowski sum of the length-L segment and the unit disk. Such sums are known as *hippodromes* [1, 2, 7, 13, 22], or *hippos* for short. We say that a hippo is *fat* if its length is constant: $L = O(1)$. In this section we show that for a fat hippo our problem can be solved in polynomial time.

Overview of the approach

Our algorithm is a Dijkstra-style search in an implicitly defined graph $\mathcal{G}$ (Fig. 3): neither the nodes nor the edges of the graph are known in advance. Instead, $\mathcal{G}$ is built incrementally, by propagating the labels from the node v with the smallest temporary label (as in standard Dijkstra). The labels are propagated to nodes in the "L-visibility" region of v, which is what the snake "sees" while it slithers for distance L starting from v. In order to discover the nodes in the region, we pull the snake from v for length L along every possible combinatorial type of path; by a packing argument, there is only a small number of the types. The edges of $\mathcal{G}$ correspond to bitangents between the paths and 1-slides. We prove that the algorithm is polynomial-time by observing that the snake must travel for at least $2\pi - 2$ before it "touches" itself with its head; this implies that the snake never "covers" any point of $\mathrm{bd}P$ with more than $O(L/\pi) = O(1)$ "layers", and hence there is a polynomial number of relevant bitangents.

In the reminder of the section we elaborate on the algorithm's details.

3.1 A Bit More Anatomy

Consider the unit disk $< m >^{(1)}$. The part of the boundary of $< m >^{(1)}$ that is also the boundary of the snake is the unit semicircle whose diameter is perpendicular to the spine at m; we call the part the *head* and the diameter the *headcut*. The endpoint of the spine that is not the mouth is called the *antimouth* a. The part of the boundary of $< a >^{(1)}$ that is also the boundary of the snake is called the *tail*. Refer to Fig. 2.

3.2 Freeze!

Let us have a closer look at the structure of the pulled-taut snake at any moment of time. Some pieces of the spine are straight-line segments. The segments are bitangents between the other, non-segment pieces supported by vertices of P, possibly via several "layers" of the snake. We call each such (maximal) piece B a *bend*; we denote the vertex that supports B by $v(B)$, and say that v is *responsible* for B.

Snake layers We show that each vertex v is responsible for $O(1)$ bends.

Definition 3.1. A bend B belongs to *layer* 1 if there exists a point $b \in B$ such that $|bv(B)| = 1$. Recursively, B is *layer*-$(k+1)$ bend if it is not layer-k and there exists a point $b \in B$ such that for some point b' at layer k, we have $|b'b| = 2$ and $b'b$ fully lies within the (closure of the) snake (Fig. 2).

Let K be the maximum index of a layer.

Lemma 3.2. $K \leq \lceil L/(2\pi - 2) \rceil$.

Proof. Let b_k, b_{k+1} be points on layers $k, k+1$ such that $|b_k b_{k+1}| = 2$. Let $\pi(b_k b_{k+1})$ be the part of the spine between b_k and b_{k+1}. The Minkowski sum $< \pi(b_k b_{k+1}) >^{(1)}$ encloses at least one obstacle, h (or else the snake is not pulled taut). The perimeter of the inflated obstacle $< h >^{(1)}$ is at least 2π, thus $|\pi(b_k b_{k+1})| + |b_{k+1} b_k| \geq 2\pi$. But $|b_{k+1} b_k| = 2$, hence $|\pi(b_k b_{k+1})| \geq 2\pi - 2$. □

As a corollary from the lemma, we have that life is very simple for a fat enough hippo:

Corollary 3.3. *For $\alpha \geq 2\pi - 2$, an α-fat hippo can follow a shortest thick path without self-overlap.*

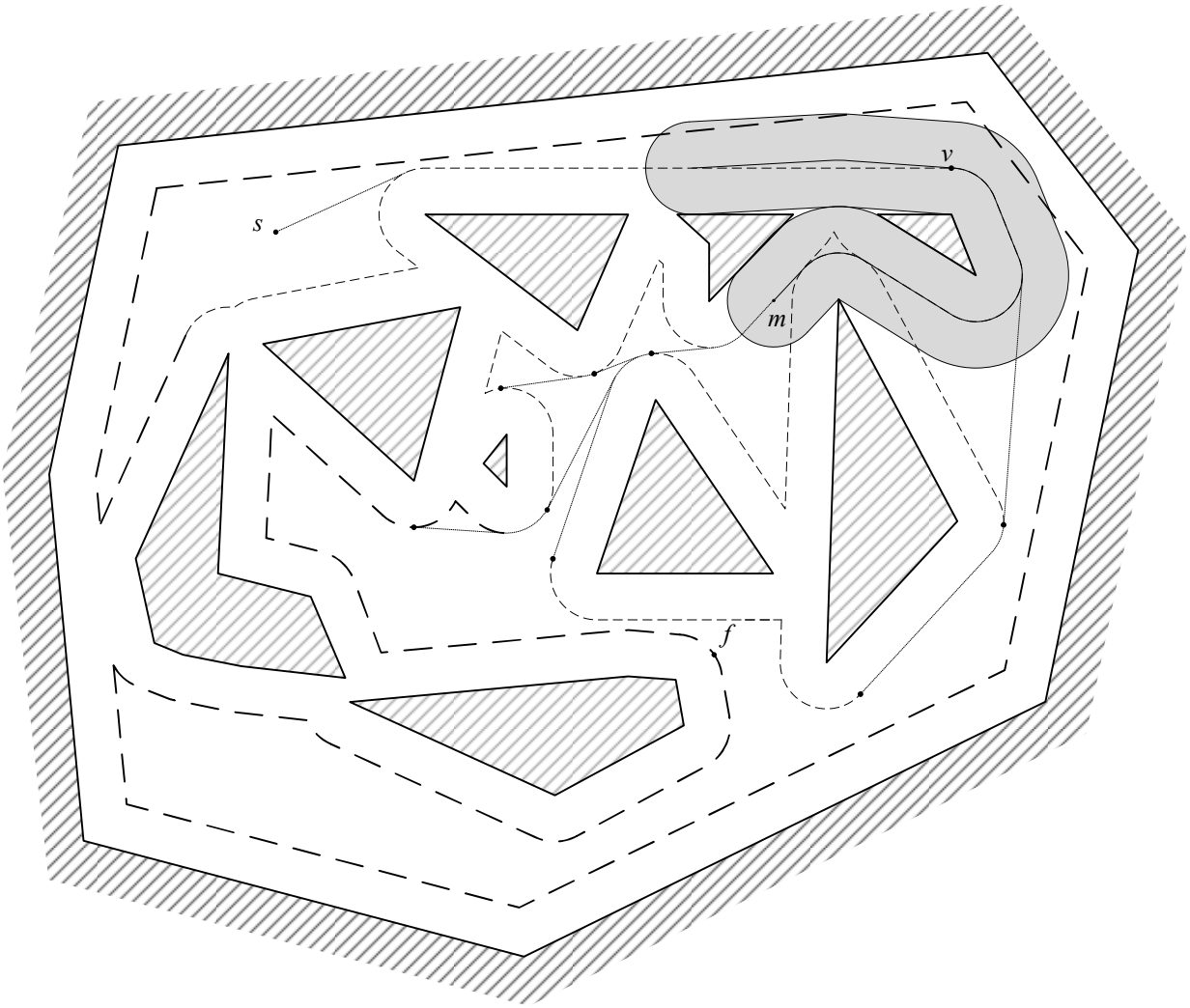

Figure 3: v is a node of $\mathcal{G}$; pull the snake from v for distance L along "every possible" path. Endpoints of visibility edges (some of them are shown with hollow circles) become nodes of $\mathcal{G}$.

Tailed and headed bends One may wonder why we did not opt for a simpler definition of layer-k bend B as one having a point b such that $|bv(B)| = 2k-1$ and $bv(B)$ lies fully within the (closure of the) snake. The reason are two special kinds of bends which make the snake's portrait more complicated than in the case of an infinite-length snake. Specifically, we say that a bend B at layer k is *tailed* (resp. *headed*) if $< B >^{(2k-3)}$ touches the tail (resp. head). The simpler definition may not work for such bends (Fig. 2).

Any bend that is not tailed or headed is an arc of a circle (actually, it is part of a slide). A tailed or headed bend B is a different shape – string pulled taut against a ball touching $v(B)$.

Snake configuration The snake can be reconstructed in linear time as soon as the following is specified: (1) list of the vertices responsible for the bends; (2) for each bend, its layer; (3) for a headed bend – the vertex or the edge of P in contact with the head, and the slope of the headcut; (4) similar information for a tailed bend; (5) positions of the mouth m and antimouth a. We will call (1)–(5) the *configuration* of the snake.

Lemma 3.4. *The list (1) contains $O(n)$ vertices.*

Proof. Each vertex may be responsible for up to K bends, hence the total number of bends is $O(Kn) = O(n)$. (In fact, since the part of the snake between consecutive bends leaning on one vertex, encloses at least one obstacle, every bend can be charged either to a vertex or to an obstacle; thus the number of bends is actually $O(n+h) = O(n)$, where h is the number of obstacles.) □

3.3 Move!

Suppose that we are given the configuration $\mathcal{C}$ of the snake at some time; let π be the spine when the snake is in $\mathcal{C}$ and let $m_{\mathcal{C}}$ be the position of the mouth in $\mathcal{C}$. Suppose we are also given a path γ for the mouth starting at $m_{\mathcal{C}}$. Assume γ has the following properties: (1) it is consistent with $\mathcal{C}$ in that the tangent to γ at $m_{\mathcal{C}}$ coincides with the tangent to π at $m_{\mathcal{C}}$; (2) it has a polynomial number of pieces, each of constant description complexity; (3) $|\gamma| \leq L$. In what follows we assume that every path γ has these properties. We claim that in polynomial time we can check whether γ is a feasible path for the mouth, i.e., whether the snake stays obstacle-free when m is pulled along γ (say, at unit speed).

First we note that it is enough to check only whether the mouth moves feasibly. Indeed, the first time that the snake (possibly) becomes infeasible as m follows γ, the mouth is necessarily a part of "certificate of infeasibility". This is so because the only way that the snake experiences "side pressure" is due to appearance of a headed bend. Other than that, any piece $\mathcal{P}$ of the snake is merely pulled by the preceding piece, $\mathcal{P}'$: either $\mathcal{P}$ exactly follows the same path that just was feasible for $\mathcal{P}'$, or $\mathcal{P}$ is a tailed bend B. Of course, the bend morphs as the time passes, but only becomes "more feasible" with time, i.e., the free space around B increases – B is pushed only by the tail, and the tail "moves away" with time.

Let now γ' be a piece of γ. If we know how each piece $\mathcal{P}$ of the snake changes with time, we can test whether m stays feasible while following γ', by checking the feasibility against each $\mathcal{P}$ in turn. That is, while neither the configuration of the snake nor the piece γ' of γ changes, the feasibility test can be done piece-versus-piece in constant time (assuming real RAM). Observe that overall there is only a polynomial number of configuration changes. Indeed, the configuration may change only due to one of the following *events*: (1) the tail starts to follow another feature of P, (2) a headed bend appears or changes its combinatorial structure, (3) a tailed bend disappears or changes its combinatorial structure. But each of the events (1)–(3) may happen only once per vertex-bend-layer triple; thus, by Lemmas 3.2 and 3.4 there is only a polynomial number of events.

Tracking the configuration changes is easy *given* the way each piece changes with time. For event (1), we only have to know how the tail speed changes with time between consecutive events (the tail speed is not necessarily constant, we elaborate on it in the next paragraphs). For event (2) we check what is the first time that the head collides with a piece or when a headed bend hits a vertex; all this can be done in polynomial time as there is only a linear number of candidate collisions. Event (3) is similar. The next event time is then the minimum of the event times over all the pieces.

It remains to show how to determine the way each piece changes. Here the crucial role is played by the headed and tailed bends; again, it is the finiteness of the snake length that makes things involved. Let $B_1 = b_1 b_1'$ be the first such bend counting from m (Fig. 2). Assume that B_1 is a layer-k tailed bend; the situation with a headed bend is actually simpler (because the mouth speed is constant). Let $c_1 b_1$ and $b_1' c_1'$ be the pieces adjacent to B_1 – both are bitangents (straight-line segments, possibly of 0 length) to B_1 and adjacent bends. Every point of the spine between m and c_1 moves at speed 1, and none of the bends before B_1 changes with time.

To figure out what happens after c_1, we have to solve a constant-size differential equation that describes the "propagation" of speed of motion of the spine. Specifically, let $u(\tau)$ denote the speed at which the antimouth moves at time τ. Knowing $u(\tau)$ and knowing the initial position of the antimouth, we can write the antimouth position as a function of time, and hence we know $< a >^{(2k-2)}$ as a function of time. The points b_1 and b_1' are points of tangency to $< a >^{(2k-2)}$ from c_1 and c_1'; thus knowing $< a >^{(2k-2)}$ we know how the length $|c_1 b_1| + |\pi(b_1 b_1')| + |b_1' c_1'|$ changes with time. Knowing that, and recalling that at c_1 the spine moves with unit speed, we can write what the spine speed at c_1' is as a function of τ.

Now, the spine speed does not change between c_1' and the next point, c_2, that is the start of the tangent to the next headed or tailed bend, B_2. We perform at B_2 the same operations as above, and get the speed of motion of the spine past the bend B_2. Continuing in this fashion, in the end, after going though all bends, we obtain some expression, $\mathcal{E}(u(\tau))$ for the spine speed after the last bend.

Finally, to close the loop, we solve the equation

$$u(\tau) = \mathcal{E}(u(\tau)) \tag{1}$$

Since there is only a constant number of layers (Lemma 3.2), there is only a constant number of the headed and tailed bends, and hence the expression $\mathcal{E}$ is a sum of a constant number of terms (each containing $u(\tau)$ and $\int u(\tau) d\tau$), each of constant description complexity. In our computation model, we can solve the equation for $u(\tau)$ in constant time. Substituting $u(\tau)$ back into the formulae for the different bends, we obtain the spine as a function of time, as desired.

3.4 See!

Assume that at some point the snake is in configuration $\mathcal{C}$. What happens next, as the snake follows the optimal path to f? Local optimality conditions imply that it will "wiggle around the obstacles" for some time and then "shoot" towards a vertex of P. We formalize this below.

Final piece of anatomy The *eye* of the snake is collocated with the mouth. The snake can *see* a point $x \in P^1$ if the segment mx lies fully within P^1 and is tangent to the spine at m. Recall that $P^1 = P \setminus < \mathrm{bd}P >^{(1)}$ where $< \mathrm{bd}P >^{(1)}$ is the Minkowski sum of $\mathrm{bd}P$ with *open* unit disk; hence mx can go along the boundary of P^1 (with x being, e.g., an endpoint of a slide; see Fig. 3). The snake itself is transparent for its eye: we do not forbid mx to intersect the snake.

Definition 3.5. A point p on $\mathrm{bd}(P^1)$ is *L-visible* from $m_{\mathcal{C}}$ if there exists a path γ for the mouth ending in a point m^* such that m^* sees p and m^*p is tangent to $\mathrm{bd}(P^1)$ at p. We say that $m_{\mathcal{C}}p$ is an *L-visibility edge*, or an *L-edge* for short. We say that γ is the *wiggle segment* of $m_{\mathcal{C}}p$, and that m^*p is the *visibility segment* of the edge. (We remind that we assume γ enjoys the properties listed in the beginning of Section 3.3: tangent at $m_{\mathcal{C}}$ consistent with $\mathcal{C}$, polynomial-size description, length $\leq L$.)

To be on a (locally) optimal path, the mouth would like to follow an L-edge also past m^*. This may not be feasible due to a conflict with the snake itself. However, such conflicts can be discovered "on-the-fly", as the mouth attempts to move along m^*p. Specifically, try to move the mouth along m^*p, as described in Section 3.3. If during the motion, the head collides with the snake, note what kind of bend the head collides with. If this bend B is not tailed, adjust the path for the mouth so that it is tangent to $< B >^{(2)}$, and follow the adjusted path. If by the time the mouth reaches $< B >^{(2)}$, the bend B is "gone", i.e., the head "misses" the snake, we know that we are dealing with a tailed bend (or with the tail itself). We identify the time

and place of contact with the bend by solving a differential equation similar to (1): assuming the speed of the tail is $u(\tau)$, we know how the bitangent between m and the corresponding ball centered at a changes; in particular, we know its length $l(\tau)$ as a function of time. The time τ^* when the head hits the tail is then the solution to the equation $\tau^* = l(\tau^*)$. After solving the equation we know the configuration of the snake at τ^*, and continue moving the mouth to p around the tail.

The above procedure essentially "develops" the path γ piece-by-piece. This is consistent with Section 3.3 where we described how to pull the snake along a *given* path γ for the mouth: the pulling was done piece-by-piece, which means that it can be performed even if γ is not given in advance but instead is revealed piece after piece. More importantly, using the procedure, we can develop *all* L-edges incident to $m_{\mathcal{C}}$, piece-by-piece, in a BFS manner. We describe this below.

Let us start the development by considering the visibility segment $m'm_{\mathcal{C}}$ of the L-edge that has led the mouth to $m_{\mathcal{C}}$. The segment is tangent to a slide $S \ni m_{\mathcal{C}}$. The slide S down from $m_{\mathcal{C}}$ (i.e., in the direction consistent with $m'm_{\mathcal{C}}$) is the first (potential) piece of a new L-edge. We extend bitangents from the piece to all other 1-slides and to all pieces of the spine inflated by 2. These bitangents become the next potential pieces for the L-edges. After the bitangents, the next potential pieces are the slides and the spine pieces at which the bitangents end. We continue in this way (possibly adjusting the pieces of a particular edge to account for snake self-interaction) until for each edge its wiggle segment reaches length L. (Refer to Figure 3 where some of the bitangents, discovered while developing edges incident to v, are shown with dotted lines.)

We now bound the total number of bitangents extended while developing all L-edges from $m_{\mathcal{C}}$. All our bitangents are between slides (up to layer K) and pieces of the wiggle segments. There are $O(n)$ slides and $O(n)$ pieces; there are $O(1)$ bitangents between any two slides, and $O(1)$ bitangents between a slide and a piece. Thus, the total number of *distinct* bitangents extended during the development is $O(n^2)$. It is possible though, that a single bitangent has to be extended more than once (Fig. 4). Nevertheless, every time the bitangent is discovered, it is reached by a new homotopy type path from $m_{\mathcal{C}}$ – this is because for any homotopy type there is a unique shortest path of the given type. It follows that the number of times the bitangents are extended (and hence the total time spent on growing L-edges from $m_{\mathcal{C}}$) is $O(n^2 \cdot X)$, where X is the maximum number of homotopy types of length-L paths from $m_{\mathcal{C}}$ to a bitangent.

To bound X, consider the set $\mathcal{I} = < m_{\mathcal{C}} >^{(L)} \cap P^1$ of points in P^1 within distance L from $m_{\mathcal{C}}$. X depends only on the number H of *holes* in $\mathcal{I}$, not on the number of vertices; more precisely, there exists a (possibly exponential) function f such that $X \leq f(H)$. Since the area of a hole in P^1 is at least π and the area of $\mathcal{I}$ is $\pi L^2 = O(1)$, we have that $H \leq L^2 = O(1)$, and hence $X \leq f(H)$ is also constant.

Lemma 3.6. *L-edges incident to $m_{\mathcal{C}}$ can be developed in time $O(n^2)$.*

3.5 Label!

We are ready now to traverse the "L-visibility" graph $\mathcal{G}$ of P, searching only the relevant part of $\mathcal{G}$ and not building the whole graph explicitly. The label of each node in $\mathcal{G}$ consists of two parts: the *distance label* (storing the distance from s) and the *configuration label* (storing the snake configuration at which the node was reached). That is, a node may have several labels – one per configuration. Applying the arguments in Lemma 3.6 "in reverse" (i.e., starting from a node of $\mathcal{G}$ and counting the number of different ways in which the node may be reached), we obtain:

Lemma 3.7. *The number of different snake configurations that can reach any node of $\mathcal{G}$ is $O(n^2)$.*

Thus the total number of labels of any node is $O(n^2)$.

Remark 6. We could differentiate between two types of bitangents in L-edges: "short" – those that are parts of the wiggle segments (and hence are actually traversed by the snake in the vicinity of $m_{\mathcal{C}}$), and "long" – the visibility segments. It is the number of long bitangents which is quadratic; the number of short segments could be shown to be constant similarly to the number of the homotopy types. With the refined argument we would bound the number of configurations that can reach any node of $\mathcal{G}$ by a constant. Still, for our purposes, any polynomial bound is enough.

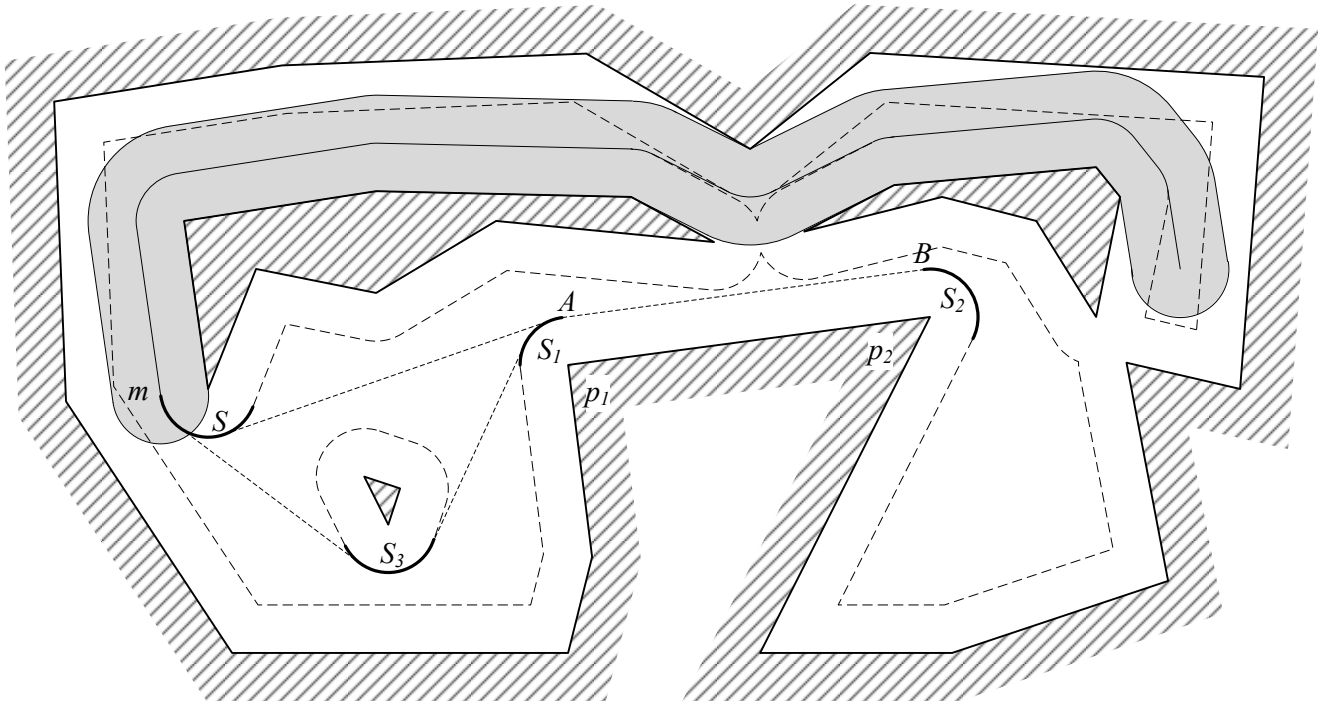

Figure 4: Bitangent AB between slides $S_1 \subset \mathrm{bd}(< p_1 >^{(1)})$ and $S_2 \subset \mathrm{bd}(< p_2 >^{(1)})$, where p_1, p_2 are vertices of P, is extended after reaching S_1 directly from the slide S, and also after reaching S_1 via the slide S_3. Note that in the first case we may not be able to continue moving the mouth because it is possible that the snake has not moved away yet to open the passage.

The labeling starts from $m_{\mathcal{C}} = s$, and assigning distance label 0 to s. The algorithm grows the graph $\mathcal{G}$ whose edges are the discovered L-edges and whose nodes are the endpoints of the L-edges. Note that by the definition of L-visibility (Definition 3.5), all nodes of $\mathcal{G}$ reside on slides. At a generic step, take the node with the smallest (temporary) distance label, make the label permanent, and construct L-edges from the node. The endpoints of the edges join $\mathcal{G}$ and get their (temporary) distance labels and configuration labels. In addition, each already existing node over which the mouth passes, gets its distance label corresponding to the same configuration label carried by the mouth updated if the new distance label is smaller than the node's current label. The search stops when f is reached.

We now prove that the algorithm terminates in a polynomial number of steps. As noted in Section 3.4, the bitangents explored by the algorithm are between slides up to layer K and pieces of the wiggle segments. These latter pieces are of two types: (1) slides up to layer K themselves and (2) curves that are obtained as the head rolls over the tail, possibly padded by up to K layers of the snake. Call a bitangent *hanging* if it goes between such a curve and a slide; since we do not know in advance the locations of the curves, there is in principle an uncountable number of possible hanging bitangents. Call a bitangent *true* if it is between two slides; there are $O(n^2)$ of true bitangents.

At every step of the algorithm, at least one true bitangent is discovered. Indeed, extension of a hanging bitangent means that the snake touches itself, which implies that it encloses at least one obstacle (Lemma 3.2). Thus, apart from the vertex touched by the tail, there are at least 2 more vertices of P that are responsible for the snake's bends (the snake cannot enclose an obstacle unless it makes at least 3 "turns"). The snake's spine between two consecutive such vertices follows a true bitangent (note that by our definition, a true bitangent does not have to go between two different holes in P^1; e.g., a straight-line edge of the boundary of P^1 is also a true bitangent).

Any of the $O(n^2)$ true bitangents may be discovered only $O(n^2)$ times — once per the configuration of the snake reaching the endpoint bitangent. Thus overall there are $O(n^4)$ steps, and we have our main positive result:

Theorem 3.8. *Shortest path for a fat hippo can be computed in polynomial time.*

4 Being a Long Snake is Hard

In this section we prove that if the snake length is not bounded, deciding existence of a path for the snake is NP-hard. Specifically, our problem is: Given polygonal domain P and points s, f, find a thick non-selfoverlapping s-f path (i.e., a thick s-f wire).

We show the problem's hardness by a reduction from planar 3SAT. Recall that the *graph* of a 3SAT instance has nodes for all variables and clauses, and two types of edges: (1) a cycle $\mathbb{C}$ through all variables, and (2) edges connecting every clause to its three variables (Fig. 5). *Planar 3SAT* is a restriction of 3SAT to instances whose graph is planar; Lichtenstein [19] proved that planar 3SAT is NP-hard. To show the hardness of wire routing, we start from an instance I of planar 3SAT; we identify the instance with its (planar) graph, and the variables and clauses with the points in the plane into which they are embedded.

Augmenting I

The cycle $\mathbb{C}$ splits the plane into two parts; each clause belongs to exactly one part. We say that the clauses, edges, etc. inside (resp. outside) $\mathbb{C}$ are *inner* (resp. *outer*).

Focus on the outer clauses. Define parent-child relationship between the clauses as follows. The clauses that belong to the outer face of the graph I are *orphans* – they have no parents. Now imagine removing an orphan C, together with the edges that connect C to the variables. Any clause C' that now (after C's removal) belongs to the outer face of I is a *child* of C (and C is the *parent* of C'). Recursively, any clause C'' is the parent for all clauses that appear on the outer face of I after removal of C'' (and all its ancestors).

We now augment I with new edges. For any parent, the children are angularly sorted around the parent. We connect the first and the last child to the parent; if a parent has only one child, connect the parent and

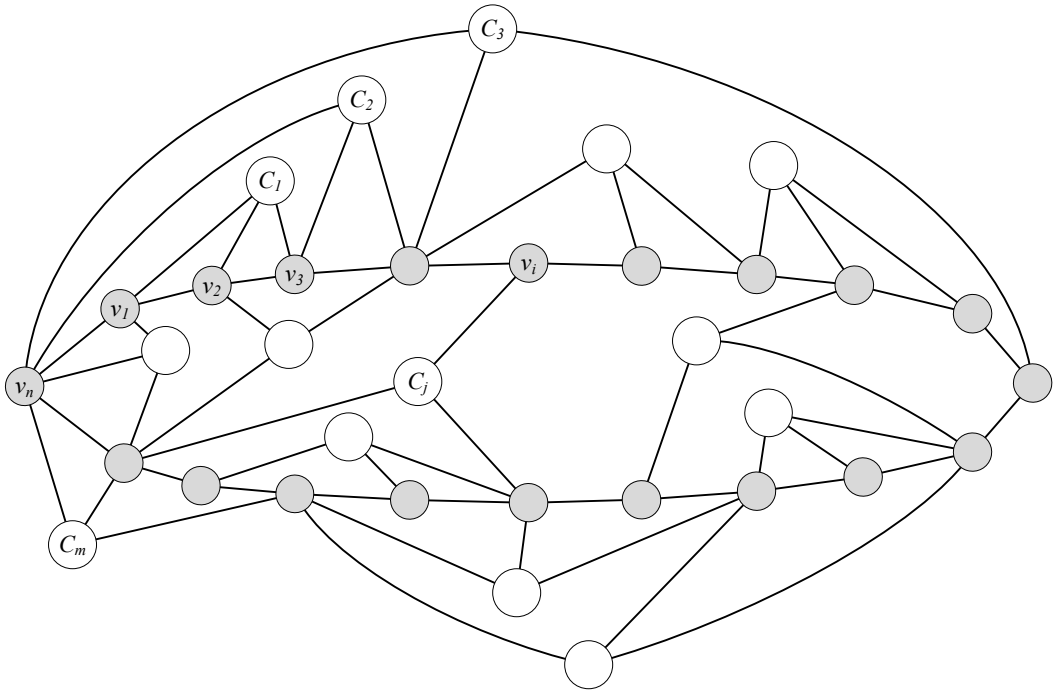

Figure 5: (The graph of) an instance I of 3SAT. The variables are shaded circles, the clauses are hollow circles.

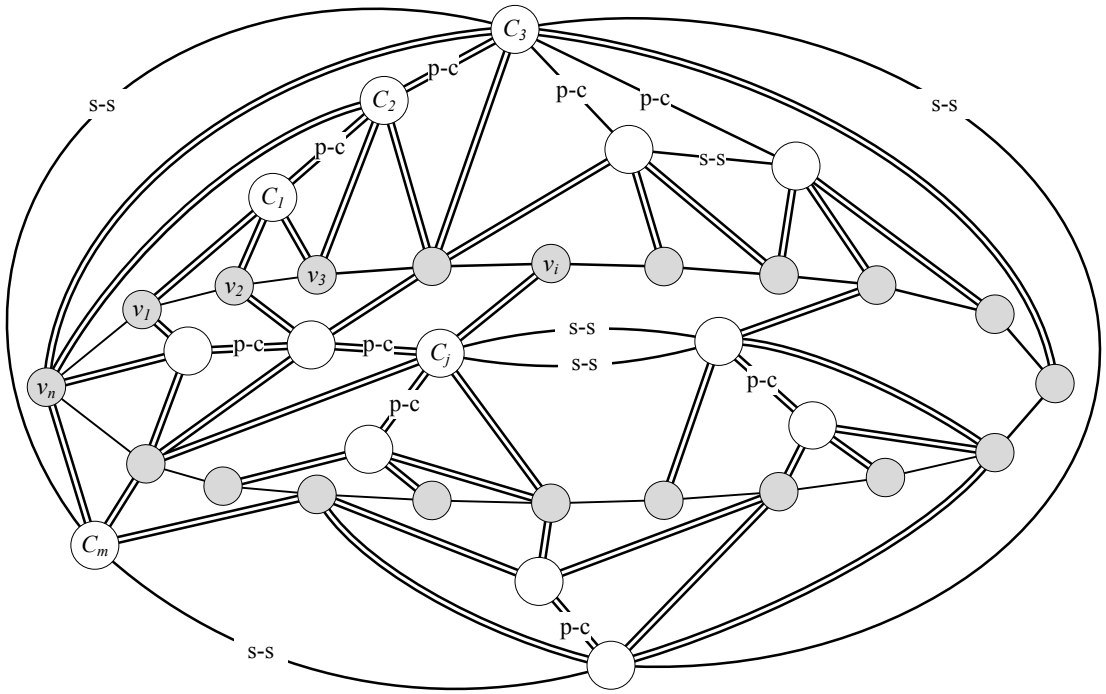

Figure 6: I augmented with parent-child edges (some of them are marked p-c), sibling edges (some marked s-s), and with variable-clause edges duplicated. The clause C_j is one of the two clauses incident to the face F (that plays the role of the outer face when augmenting the graph inside $\mathbb{C}$).

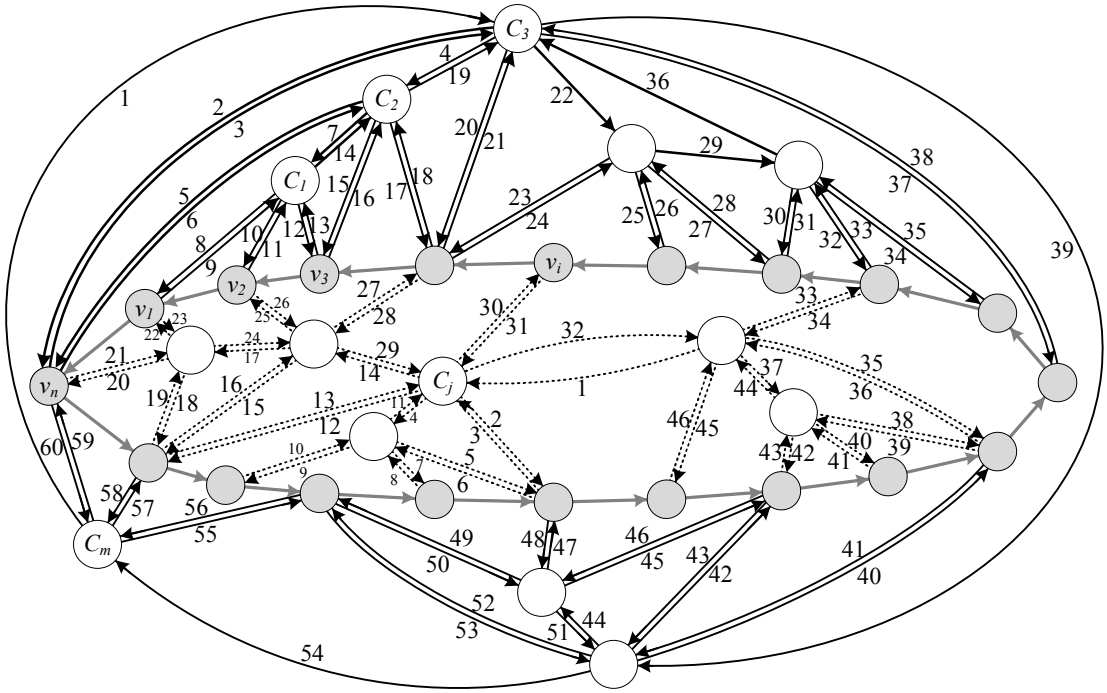

Figure 7: The closed walks $\mathcal{W}_{out}$, $\mathcal{W}_{in}$; the numbers indicate the order in which edges are traversed by the walks.

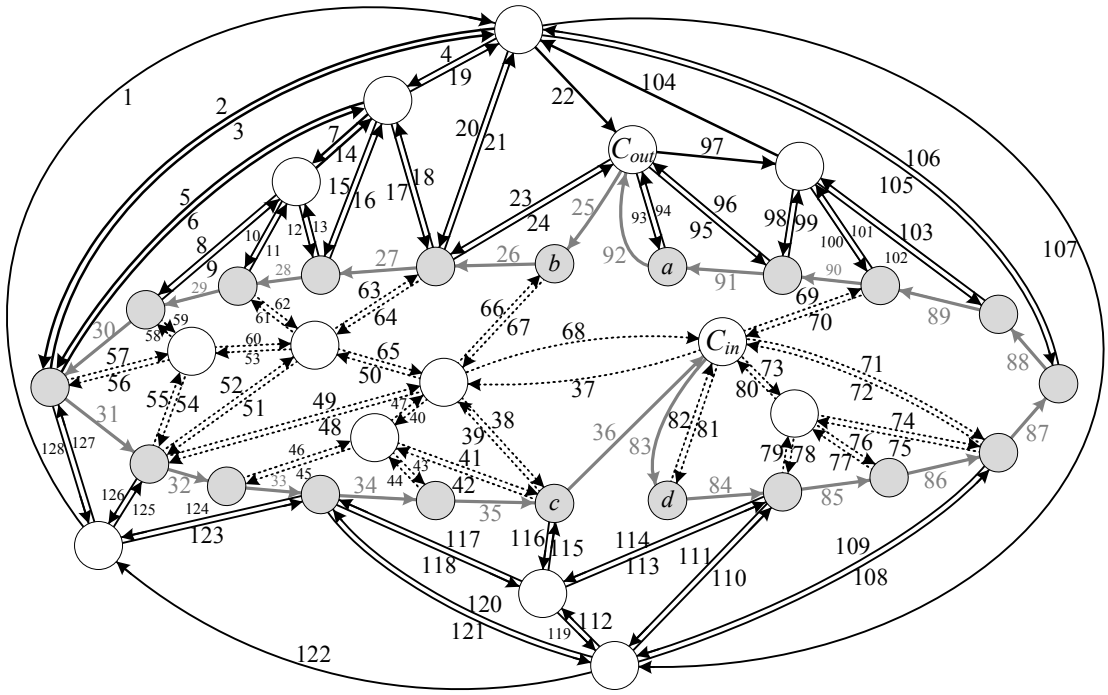

Figure 8: The closed walk $\mathcal{W}$; the numbers indicate the order in which edges are traversed by the walk.

the child by 2 parallel edges (parallel in the graph-theoretic sense, in the embedding they are not parallel). Add also edges between siblings; in particular, connect orphans with a cycle, in the order in which they appear on the outer face of I (Fig. 6). Finally, add a parallel edge for each clause-variable edge (so that every clause is connected to each of its variables with 2 parallel edges).

Similar augmentation is done for the inner part of $\mathbb{C}$: (1) Choose an arbitrary inner face F; the clauses incident to F are orphans – they are siblings, so connect them in a cycle with sibling-sibling edges in the order as they appear around F. (2) Imagine removing the orphans one-by-one; every clause that appears on F after an orphan is removed, has the orphan as its parent – add parent-child and sibling-sibling edges. Recurse.

The walks $\mathcal{W}_{out}$,$\mathcal{W}_{in}$

Let $\mathcal{W}_{out}$ be the closed walk that goes through the outer clauses and all variables in the DFS manner, with preference to go right (as seen from a clause) and down (i.e., towards $\mathbb{C}$). Specifically, at the "top level", the walk contains the cycle through all orphans. In addition, at each orphan C (and in general, at each clause) the walk is the DFS traversal of the subtree of C: it goes to the rightmost (as seen from C) variable of C, then goes back to C (using the parallel edge), and then – either to the next variable of C or to the rightmost child C' of C (whichever is more to the right). At C', the walk recurses down to the rightmost variable of C', then goes back to C', and then again – either to the next variable of C' or to the rightmost child of C', etc. The walk follows the edge to the sibling or to the parent of a clause C'' after all children and variables of C'' have been visited. In particular, the variables of childless clauses are just visited one-by-one, from right to left. Refer to Fig. 7.

We do an analogous augmentation of the subgraph of I inside the cycle $\mathbb{C}$. Specifically, we choose some face F of I inside $\mathbb{C}$, and let F play the role of the outer face: The orphan clauses are those on the boundary of F; the children are defined recursively as the clauses that appear on F after deletion of parents. As before, we duplicate variable-clause edges. Let $\mathcal{W}_{in}$ be closed walk analogous to $\mathcal{W}_{out}$: $\mathcal{W}_{in}$ goes through the orphans, recursing to variables and children in the DFS manner, with the preference to go left (as seen from a clause) and towards $\mathbb{C}$. Refer to Fig. 7.

The walk $\mathcal{W}$

We now splice the walks $\mathcal{W}_{out}$, $\mathcal{W}_{in}$, and the cycle $\mathbb{C}$ into a single closed walk $\mathcal{W}$ through I. Let ab be an arbitrary edge of $\mathbb{C}$, and let C_{out} be an outer clause that belongs to the face of I that has ab on the boundary. Remove ab from $\mathbb{C}$, and add edges aC_{out}, bC_{out} (Fig. 8). Similarly, remove an edge cd from $\mathbb{C}$, and add edges cC_{in}, dC_{in} to an inner clause C_{in}.

The walk $\mathcal{W}$ starts from following $\mathcal{W}_{out}$ until reaching C_{out}, at which point it uses the edge $C_{out}b$. (This may not necessarily be the first time the walk visits C_{out}; the walk uses $C_{out}b$ so that the usage is consistent with the ordering of edges around C_{out} – refer to Fig. 8, where C_{out} is reached first by edge 22, while $C_{out}b$ is only 25th.) From b, the walk follows the cycle $\mathbb{C}$ up to c, where it uses the edge cC_{in} to enter inside $\mathbb{C}$. From C_{in}, the walk $\mathcal{W}$ follows the walk $\mathcal{W}_{in}$ all the way around back to C_{in}. It then uses $C_{in}d$ to get back to $\mathbb{C}$, upon which it traverses $\mathbb{C}$ from d to a. At a, the walk uses aC_{in} and follows the rest of $\mathcal{W}_{out}$ from there.

From 3SAT to wire routing

As the last step of the reduction, we convert I to an instance of finding a thick wire. For that, we thicken the edges of I, turning them into channels of width 2. We replace variables and clauses with gadgets shown in Figs. 9 and 10 resp. The connections (channels) between variables and clauses (Fig. 11) ensure that whenever a channel from a clause to a variable is used by the wire, the variable satisfies the clause.

We cut (the channel corresponding to) one of the edges of $\mathbb{C}$, and place the points s, f on the opposite sides of the cut. This turns the closed walk $\mathcal{W}$ into an s-f path. By our construction, the only s-f wire in the instance is one that follows the walk $\mathcal{W}$ in I, possibly, omitting some clause-variable edges (channels). Indeed,

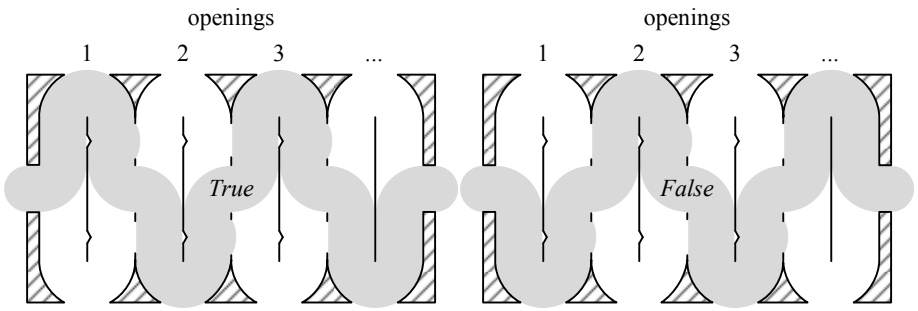

Figure 9: A variable gadget can be traversed in one of the two ways, setting the truth assignment. Depending on the way, the path bulges out of the odd (when the variable is set to True) or even (when it is False) openings at the top of the gadget, and resp. even/odd openings at the bottom.

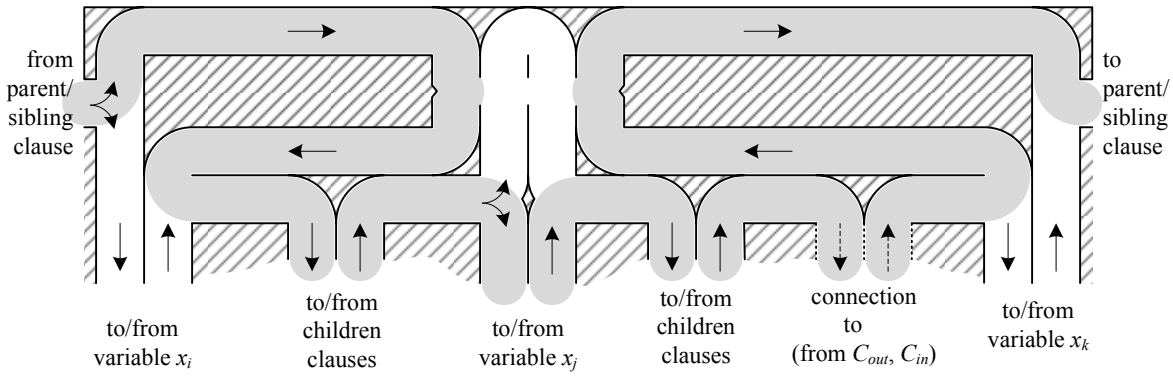

Figure 10: When a clause gadget is traversed, from left to right, one of the channels leading to variables must be used. Otherwise, 3 subpaths go through the top of the gadget leading to a self-overlap.

neither channels nor gadgets have any leakage – the wire must follow them through. The only flexibility that the wire has is (1) how to traverse the variable gadgets, and (2) which clause-variable channels (not) to use. But we know, by clause gadget construction (see Fig. 10), that at least one channel from every clause must be used; moreover, when it is used, the variable must satisfy the clause (see Fig. 11). Thus, the s-f wire exists if and only if I is satisfiable.

5 Conclusion

We showed that for a snake to stay simple, it must grow fat (or else, learn to squeeze). We mention few open problems here:

To minimize snake's squeeze factor, it would be interesting to find a path that minimizes maximum self-overlap of a snake. It is easy to see that a snake of thickness 1/2 can follow (without self-overlap) the shortest (possibly self-overlapping) path for a thickness-1 snake. Is 1/2 best possible? Note that in our hardness proof, a non-zero amount of overlap is enforced; what is the largest self-overlap for which the problem remains hard?

Minimizing the path length for *one* point of the snake is similar to finding "d_1-optimal" motion of a rod [5]. Other objectives are possible: e.g., distance traveled by another point (not the mouth) or the average distance traveled (d_2-, ..., d_∞-optimality, etc.). Does the situation with the snake mimic that for the rod: minimizing motion of any point is NP-hard, except for a rod endpoint?

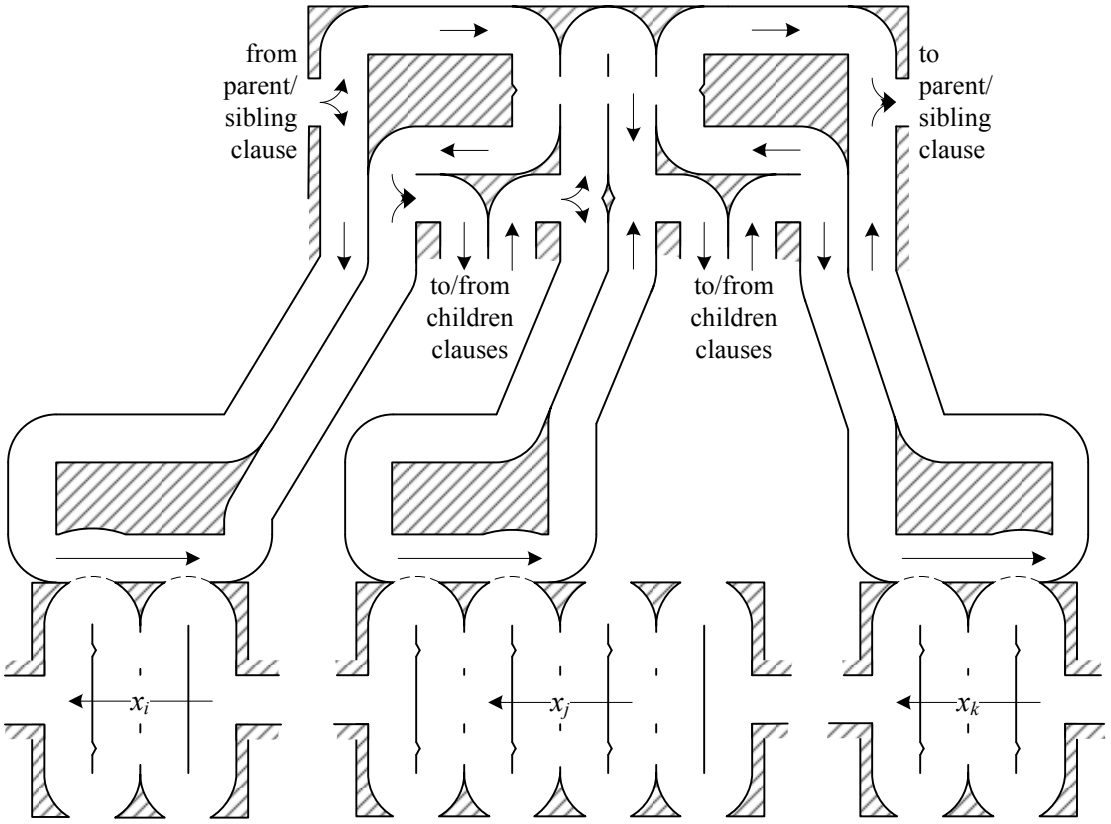

Figure 11: If a variable does not satisfy a clause the channel between them cannot be used by non-selfoverlapping path.

What is the hardness of computing paths for a short snake that is not required to be pulled taut? It seems that the answer to this question leads to an interesting research direction of "snake packing": Given a polygonal domain, can one layout a length-L snake in it? The problem is NP-hard by a reduction from Hamiltonicity of grid graphs; what about, say, simple polygons?

Can a shortest *rectilinear* wire be computed efficiently?

Acknowledgements We thank Estie Arkin, David Kirkpatrick, Joe Mitchell and Jukka Suomela for discussions. VP is supported by the Academy of Finland grant 138520.

References

[1] P. K. Agarwal, A. Efrat, and M. Sharir. Vertical decomposition of shallow levels in 3-dimensional arrangements and its applications. *SIAM Journal on Computing*, 29(3):912–953, 2000.

[2] P. K. Agarwal and M. Sharir. Efficient algorithms for geometric optimization. *ACM Computing Surveys*, 30:412–458, 1998.

[3] R. Alterovitz, M. S. Branicky, and K. Y. Goldberg. Motion planning under uncertainty for image-guided medical needle steering. *International Journal of Robotic Research*, 27(11-12):1361–1374, 2008.

[4] E. M. Arkin, J. S. B. Mitchell, and V. Polishchuk. Maximum thick paths in static and dynamic environments. *Computational Geometry Theory and Applications*, 43(3):279–294, 2010.

[5] T. Asano, D. Kirkpatrick, and C. K. Yap. d_1-optimal motion for a rod. In *Proceedings of the 12th Annual ACM Symposium on Computational Geometry*, pages 252–263, 1996.

[6] T. Asano, D. Kirkpatrick, and C. K. Yap. Minimizing the trace length of a rod endpoint in the presence of polygonal obstacles is NP-hard. In *Proceeding of Canadian Conference on Computational Geometry*, pages 10–13, 2003.

[7] J. Barcia, J. Diaz-Banez, F. Gomez, and I. Ventura. The anchored Voronoi diagram: static and dynamic versions and applications. In *19th European Workshop on Computational Geometry*, 2003.

[8] S. Bereg and D. Kirkpatrick. Curvature-bounded traversals of narrow corridors. In *Proceedings of the 21st Annual Symposium on Computational geometry*, pages 278–287, 2005.

[9] L. P. Chew. Planning the shortest path for a disc in $O(n^2 \log n)$ time. In *Proceedings of the 1st Annual Symposium on Computational Geometry*, pages 214–220, 1985.

[10] D. Chowdhury. Molecular motors: Design, mechanism, and control. *Computing in Science and Engineering*, 10:70–77, 2008.

[11] A. F. Cook IV, C. Wenk, O. Daescu, S. Bitner, Y. K. Cheung, and A. Kurdia. Visiting a sequence of points with a bevel-tip needle. In *Proceedings of the 9th Latin American Theoretical Informatics Symposium*, 2010.

[12] D. P. Dobkin and D. L. Souvaine. Computational geometry in a curved world. *Algorithmica*, 5:421–457, 1990.

[13] A. Efrat and M. Sharir. A near-linear algorithm for the planar segment center problem. *Discrete & Computational Geometry*, 16:239–257, 1996.

[14] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1999.

[15] T. Hu, A. Kahng, and G. Robins. Optimal robust path planning in general environments. *IEEE Transactions on Robotics and Automation*, 9:775–784, 1993.

[16] J.-C. Latombe. *Robot Motion Planning*. Kluwer, Boston, 1991.

[17] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[18] J. Y. Lee and H. Choset. Sensor-based planning for a rod-shaped robot in three dimensions: Piecewise retracts of R3 x S2. *International Journal of Robotic Research*, 24(5):343–383, 2005.

[19] D. Lichtenstein. Planar formulae and their uses. *SIAM Journal on Computing*, 11(2):329–343, 1982.

[20] F. M. Maley. *Single-Layer Wire Routing and Compaction*. MIT Press, 1990.

[21] K. McEvoy and J. V. Tucker, editors. *Theoretical Foundations of VLSI Design*. Cambridge University Press, New York, NY, USA, 1991.

[22] J. Pach and G. Tardos. Forbidden patterns and unit distances. In *Proceedings of the 21st Annual Symposium on Computational Geometry*, pages 1–9, 2005.

[23] S. Sifrony and M. Sharir. A new efficient motion-planning algorithm for a rod in two-dimensional polygonal space. *Algorithmica*, 2:367–402, 1987.

[24] C. Veigel, L. M. Coluccio, J. D. Jontes, J. C. Sparrow, R. A. Milligan, and J. Molloy. The motor protein myosin-I produces its working stroke in two steps. *Nature*, 398:530–533, 1999.