Routing on the Visibility Graph*

Prosenjit Bose[1], Matias Korman[2], André van Renssen[3,4], and Sander Verdonschot[1]

1 **School of Computer Science, Carleton University, Ottawa, Canada**
`jit@scs.carleton.ca`, `sander@cg.scs.carleton.ca`
2 **Tohoku University, Sendai, Japan**
`mati@dais.is.tohoku.ac.jp`
3 **National Institute of Informatics, Tokyo, Japan**
`andre@nii.ac.jp`
4 **JST, ERATO, Kawarabayashi Large Graph Project**

Abstract

We consider the problem of routing on a network in the presence of line segment constraints (i.e., obstacles that edges in our network are not allowed to cross). Let P be a set of n points in the plane and let S be a set of non-crossing line segments whose endpoints are in P. We present two deterministic 1-local $O(1)$-memory routing algorithms that are guaranteed to find a path of at most linear size between any pair of vertices of the *visibility graph* of P with respect to a set of constraints S (i.e., the algorithms never look beyond the direct neighbours of the current location and store only a constant amount of information). Contrary to *all* existing deterministic local routing algorithms, our routing algorithms do not route on a plane subgraph of the visibility graph.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, G.2.2 Graph Theory

Keywords and phrases Routing, constraints, visibility graph, Θ-graph

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2017.18

1 Introduction

Routing is a fundamental problem in networking. The goal is to find a path from a source vertex to a destination vertex in the network. When the whole network is known to the routing algorithm, there exist many algorithms to find paths. The problem is more challenging when the only information available is the location of the current vertex, its neighbours and a constant amount of additional information (such as the source and destination vertex). This is often referred to as *local* routing (or k-local for some constant k, when the k-neighbourhood is considered). In our setting, we assume that the network is a graph embedded in the plane, with edges being straight line segments connecting pairs of vertices, weighted by the Euclidean distance between their endpoints. Algorithms routing on such networks are referred to as *geometric* routing algorithms (see [7] and [8] for surveys of the area).

Deterministic routing algorithms that guarantee delivery in these networks typically route on plane subgraphs of the complete Euclidean graph. This means that of the potentially quadratic number of edges available to the routing algorithm, only a linear number are ever

* P. B. is supported in part by NSERC. M. K. was partially supported by MEXT KAKENHI Nos. 15H02665, and 17K12635. A. v. R. was supported by JST ERATO Grant Number JPMJER1201, Japan. S. V. is supported in part by NSERC and the Carleton-Fields Postdoctoral Award.

© Prosenjit Bose, Matias Korman, André van Renssen and Sander Verdonschot;
licensed under Creative Commons License CC-BY
28th International Symposium on Algorithms and Computation (ISAAC 2017).
Editors: Yoshio Okamoto and Takeshi Tokuyama; Article No. 18; pp. 18:1–18:12

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

considered. This forces these algorithms to use paths that are much longer than necessary. In this paper, we present the first deterministic local routing algorithm that considers more edges by not restricting its choices to a plane subgraph.

Moreover, we study routing algorithms in a more general setting. In certain cases, some edges of a network may not be usable if for example there is a large obstacle blocking direct communication between two nodes. We model this impossibility via a set S of non-intersecting *line segment constraints* whose endpoints are vertices of the network. Given a set P of n points in the plane and a set S of non-intersecting line segment constraints, we say that two vertices u and v can *see each other* provided that either the line segment uv does not properly intersect any constraint in S or uv is itself a constraint in S. If two vertices u and v can see each other, the line segment uv is referred to as a *visibility edge*. The *visibility graph* of P with respect to a set of constraints S, denoted $Vis(P, S)$, has P as vertex set and all visibility edges as edge set. In other words, $Vis(P, S)$ is the complete graph on P minus all edges that properly intersect one or more constraints in S.

Although this setting has been studied extensively in the context of motion planning amid obstacles ([5, 6, 1, 4]), there has not been much work on routing in this setting. Bose *et al.* [2] showed that it is possible to route locally and 2-competitively between any two visible vertices in the constrained Θ_6-graph. Additionally, an 18-competitive routing algorithm between any two visible vertices in the constrained half-Θ_6-graph was provided. In the same paper it was shown that no deterministic local routing algorithm is $o(\sqrt{n})$-competitive between all pairs of vertices of the constrained Θ_6-graph, regardless of the amount of memory it is allowed to use. Recently, the authors presented a non-competitive 1-local $O(1)$-memory routing algorithm to route on the visibility graph by determining locally the edges of the constrained half-Θ_6-graph [3], a plane subgraph of $Vis(P, S)$.

We present two deterministic 1-local $O(1)$-memory routing algorithms on $Vis(P, S)$. The first algorithm locally computes a non-plane subgraph of the visibility graph (the constrained Θ_6-graph) and routes on it. We then modify this algorithm to obtain a routing algorithm that routes directly on the visibility graph. To the best of our knowledge, this is the first local routing algorithm does not compute a plane subgraph of the visibility graph.

2 Preliminaries

The Θ_m-graph plays an important role in our routing strategy. We begin by defining it. Define a *cone* C to be the region in the plane between two rays originating from a vertex referred to as the *apex* of the cone. When constructing a (constrained) Θ_m-graph, for each vertex u consider the rays originating from u with the angle between consecutive rays being $2\pi/m$. Each pair of consecutive rays defines a cone. The cones are oriented such that the bisector of some cone coincides with the vertical ray emanating from u that lies above u. Let this cone be C_0 of u and number the cones in clockwise order around u (see Fig. 1). The cones around the other vertices have the same orientation as the ones around u. We write C_i^u to indicate the i-th cone of a vertex u, or C_i if u is clear from the context. For ease of exposition, we only consider point sets in general position: no two points lie on a line parallel to one of the rays that define the cones, no two points lie on a line perpendicular to the bisector of a cone, and no three points are collinear. The main implication of this assumption is that no point lies on a cone boundary.

Let vertex u be an endpoint of a constraint c (if any) and let v be the other endpoint and let cone C_i^u be the cone that contains it. The lines through all such constraints c split C_i^u into several *subcones* (see Fig. 2). We use $C_{i,j}^u$ to denote the j-th subcone of C_i^u (again,

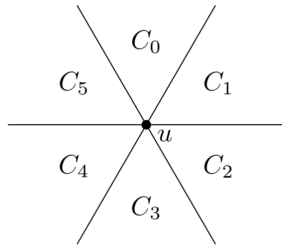

Figure 1 The cones with apex u in the Θ_6-graph. All points of S have exactly six cones.

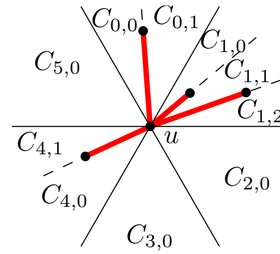

Figure 2 The subcones with apex u in the constrained Θ_6-graph (constraints denoted as red thick segments).

numbered in clockwise order). When a constraint $c = (u, v)$ splits a cone of u into two subcones, we define v to lie in both of these subcones. We consider a cone that is not split to be a single subcone.

We now introduce the *constrained* Θ_m-graph: for each subcone $C_{i,j}$ of each vertex u, add an edge from u to the closest vertex in that subcone that can see u, where distance is measured along the bisector of the original cone (*not the subcone*). More formally, we add an edge between two vertices u and v if v can see u, $v \in C^u_{i,j}$, and for all points $w \in C^u_{i,j}$ that can see u, $|uv'| \leq |uw'|$, where v' and w' denote the projection of v and w on the bisector of C^u_i and $|xy|$ denotes the length of the line segment between two points x and y. Note that our general position assumption implies that each vertex adds at most one edge per subcone.

We now define our routing model. Formally, a routing algorithm A is a deterministic 1-local, $O(1)$-memory routing algorithm, if the vertex to which a message is forwarded from the current vertex s is a function of s, t, $N(s)$, and M, where t is the destination vertex, $N(s)$ is the set of vertices adjacent to s and set of constraints incident to s and M is a memory of constant size, stored with the message. We consider a unit of memory to consist of a $\log_2 n$ bit integer or a point in P. Our model assumes that the only information stored at each vertex of the graph is $N(s)$.

▶ **Lemma 1.** [1] *Let u, v, and w be three arbitrary points in the plane such that uw and vw are visibility edges and w is not the endpoint of a constraint intersecting the interior of triangle uvw. Then there exists a convex chain of visibility edges from u to v in triangle uvw, such that the polygon defined by uw, wv and the convex chain is empty and does not contain any constraints.*

If u and v do not see each other, the above lemma proves the existence of a convex path between them. We use this property repeatedly in our routing algorithm.

3 Routing in the Constrained Θ_6-Graph

Prior to describing our routing strategy for the entire visibility graph, we first provide one for the constrained Θ_6-graph. Note that the Θ_6-graph is not necessarily plane. In this section, we assume that we are given the constrained Θ_6-graph explicitly. In the next section, we show how to use this algorithm to route on the visibility graph.

If there are no constraints, there exists a simple local routing algorithm that works on all Θ-graphs with at least 4 cones. This routing algorithm, which we call Θ-routing, always follows the edge to the closest vertex in the cone that contains the destination. In the constrained setting, this algorithm follows the edge to the closest vertex in the *subcone* that

contains the destination. Unfortunately, this approach does not necessarily succeed in the constrained setting due to two issues. First, a key factor of convergence in the unconstrained Θ-routing algorithm is that each step gets us closer to the destination (as long as we have at least 6 cones). Unfortunately, this property need not hold in the constrained setting (see Figure 3a).

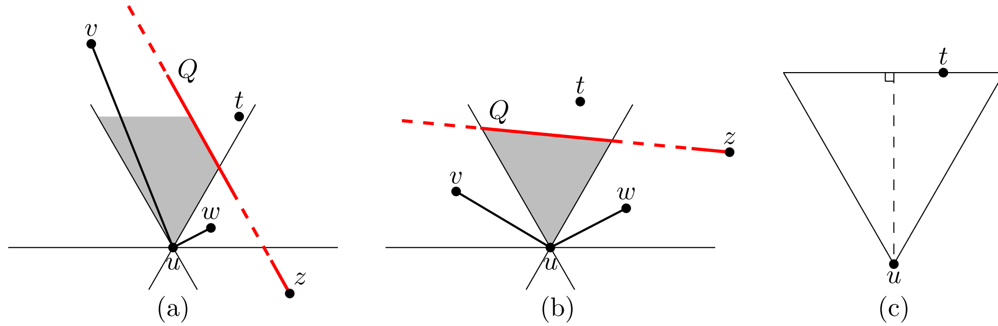

Figure 3 (a) The situation in which Θ-routing follows an edge to v and ends up further away from the destination. (b) The situation where the Θ-routing algorithm cannot follow any edges at u, since the destination t lies behind a constraint. (c) The canonical triangle of u.

A second, more important problem is that the cone containing the destination need not contain any visible vertices. This happens when a constraint is directly blocking visibility (see Figure 3b). In this case, the Θ-routing algorithm will get stuck, since it cannot follow any edge in that cone.

The first problem can be easily fixed: given a vertex u and the destination t, we define the *canonical triangle* of u with respect to t, denoted $\triangle_{ut}$, as the triangle with apex u, bounded by the cone boundaries of the cone of u that contains t and the line through t perpendicular to the bisector of the cone (see Figure 3c). If the edge of u that lies in that cone ends outside the canonical triangle, we say it is *invalid* and we ignore it. By ignoring invalid edges we make sure that any edge we follow leads to a vertex that is closer to t.

To solve the second problem, the routing algorithm needs to find a path even when an obstacle is blocking visibility to the destination (either blocking all visibility from u in the cone of t or because the edge in that cone is invalid). In this case the algorithm enters the *obstacle avoidance phase*, routing differently until an endpoint of the blocking constraint is reached.

Intuitively, our algorithm uses the Θ-routing algorithm until it gets stuck, at which point it switches to the obstacle avoidance phase in order to get around the constraint blocking its visibility to t. After this phase ends, the algorithm switches back to the Θ-routing algorithm. This process is repeated until t is reached. A more precise description follows in Section 3.2.

3.1 Obstacle Avoidance Phase

We now describe the obstacle avoidance phase. The algorithm enters this phase when routing from source s to destination t, and reaches a vertex u that does not have any valid edges in the cone that contains t. This can only happen if a constraint Q is blocking visibility (if many of them exist, let Q be the one whose intersection with segment $\overline{ut}$ is closest to u). The goal of this phase is to reach the right endpoint of Q, which we denote as z. The main difficulty with this phase is that the algorithm does not know where z is, since Q is not incident on u. In order to overcome this difficulty, the algorithm exploits several geometric

properties arising from the unique symmetries present in the constrained Θ_6-graph, some of which are outlined in the proof of Lemma 2.

Without loss of generality, t lies in C_0^u. We first describe the case where u has no edges in C_0. The general case, where u may have invalid edges in C_0, will be considered afterwards. In this first case, the algorithm proceeds as follows. At a current vertex m, the algorithm considers one of two candidate edges to follow (see Figure 4). The first is the edge to the closest visible vertex v in the subcone of C_2^m that shares a boundary with C_1^m. The second edge is the edge from m to the vertex w in C_1^m that minimizes the angle α between $\overline{mw}$ and the right boundary of C_0^m. If v lies in C_4^w and m is not the endpoint of a constraint that intersects the interior of triangle mvw, the algorithm follows the edge to v. Otherwise, it follows the edge to w. In the proof of Lemma 2, we show that at least one of v or w exists. If one of the two vertices v or w does not exist, the algorithm follows the edge that does exist. The obstacle avoidance phase ends when the algorithm reaches the endpoint of a constraint that intersects $\overline{ut}$. In order to recognize this, the algorithm stores u when the phase begins.

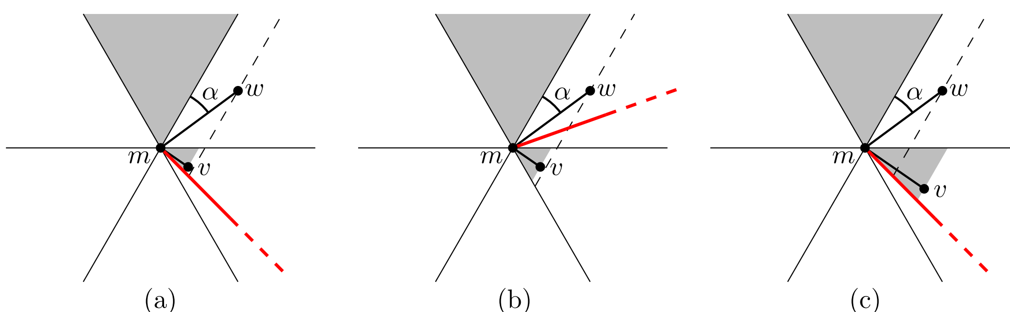

Figure 4 Routing from a vertex m. (a) Follow the edge to v, since v lies in C_4^w. (b) Follow the edge to w, since m is the endpoint of a constraint that intersects mvw. (c) Follow the edge to w, since v lies outside of C_4^w.

▶ **Lemma 2.** *When u has no edges in the cone containing the destination t, the obstacle avoidance phase initiated by u reaches the right endpoint z of the closest constraint Q blocking visibility to t.*

Proof. Without loss of generality, let t lie in C_0^u. Since u has no edges in C_0, the closest constraint Q must intersect both boundaries of C_0^u. This implies that z is either in C_1^u or C_2^u. We maintain the invariant that each intermediate vertex m has no edges in C_0^m and that the intersection of the right boundary of C_0^m and Q is closer to z than in the previous step. We first show that there always exists either a w in C_1^m or a v in C_2^m. This implies that our algorithm eventually reaches z since there are a finite number of points in P.

As a consequence of our invariant, z must either lie in C_1^m or C_2^m. Since m has no edges in C_0, we have that Q is the closest constraint to m in C_0^m. Thus, any point x on $Q \cap C_0^m$ is visible from both m and z. Hence, we can apply Lemma 1 to the triangle mxz and obtain a convex chain of visibility edges from m to z. In particular, this implies that m can see a vertex in $C_1 \cup C_2$, and therefore has an edge in $C_1 \cup C_2$. What remains to be shown is that the invariant is maintained after every step of the algorithm. We note that for any vertex in $C_1^m \cup C_2^m$ the intersection of the right boundary of its cone C_0 is closer to z than that of m. Thus, it remains to show that C_0 of this next vertex contains no edges. We consider the following two cases.

The algorithm follows the edge to v. If the algorithm follows the edge to v, recall that v lies in C_4^w and m is not the endpoint of a constraint that intersects the interior of triangle mvw. In particular, this means that w lies outside of C_0^v. Since v is the closest visible vertex in the subcone of C_2^m that shares a boundary with C_1^m, the part of C_0^v below the horizontal line through m must be empty of points visible to v (see Figure 5a).

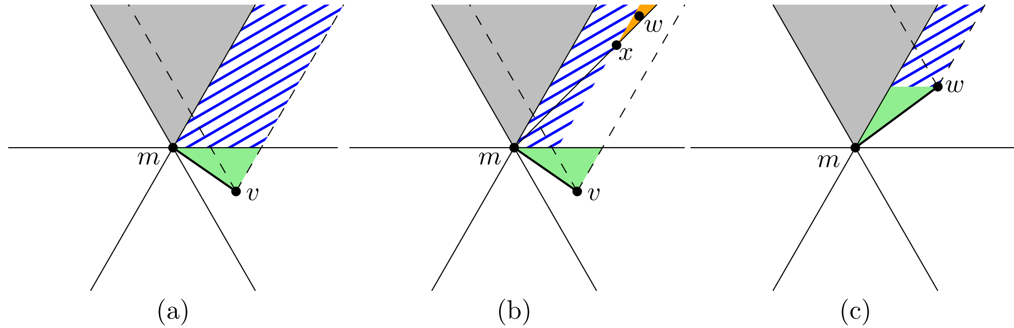

Figure 5 (a) If m routes to v, the union of green and blue regions must be empty of points. (b) An illustration of the proof: if the region is not empty, we find a point x that must have an edge with m that we would have followed instead of v. (c) Routing from m to w.

By the invariant, $C_0^m \cap C_0^v$ is empty of visible points. What remains to be shown is that there are no points visible to v in $C_0^v \setminus C_0^m$ above the horizontal line through m. If this region is not empty, we sweep the region using the right boundary of C_0^m. Let x be the first vertex hit by this sweep that is visible to m. This implies that the $\triangle_{xm}$ is empty of points visible to x since it is contained in the union of C_0^m (which is empty), the swept part of C_1^m, and a portion of C_2^m that must also be empty by our choice of v. This implies that there is an edge from x to m. This means that w must exist. By construction, $\overline{mw}$ forms the smallest angle with the right boundary of C_0^m. This means that $x \in \triangle_{mw}$. Furthermore, since mw and mv are visibility edges, Lemma 1 implies the existence of a vertex visible to w in $\triangle_{wm}$. This contradicts the existence of the edge mw. Thus, C_0^v is empty of vertices visible to m. Suppose that there was a vertex y visible to v in C_0^v, then since vy and vm are visibility edges, Lemma 1 implies the existence of a vertex visible to m in C_0^v, which is a contradiction.

The algorithm follows the edge to w. As in the previous case, we consider the part below the horizontal line through w and the part above (solid green and dashed blue regions in Figure 5c, respectively). The former region must be empty or the edge mw would not be present: any point visible to m in this region prevents m from creating an edge to w and vice versa. An argument similar to the one for v, showing that the region above the horizontal boundary of C_1 is empty, also proves that the region above the horizontal line through w is empty. Thus, C_0^w must be empty of points visible to w.

◀

We now consider the general case, where u may have invalid edges in C_0 (see Figure 6a). In this case, when u initiates the obstacle avoidance phase, we either reach z or a vertex m that has no edges in C_1 and C_2 (see Figure 6b). This latter case can only occur when z lies in C_3^m. Note that this implies that Q intersects both boundaries of C_1^m. Therefore, we initiate a new obstacle avoidance phase from m where C_1 plays the role of C_0. By Lemma 2, the second invocation of the obstacle avoidance phase must reach z.

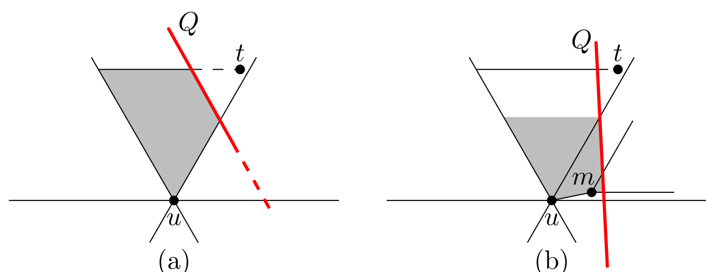

Figure 6 (a) When Q does not fully block the visibility of C_0, we maintain the invariant that the visible portion of the canonical triangle (gray region) must be empty along our routing. (b) The situation where we restart the obstacle avoidance algorithm at m.

▶ **Lemma 3.** *When u has no valid edges in the cone containing the destination t, the general obstacle avoidance phase initiated by u reaches the right endpoint z of the closest constraint Q blocking visibility to t.*

We note that the above proof relies heavily on the fact that we have exactly 6 cones (and thus we are in the constrained Θ_6-graph). We have a specific example in which the routing strategy described above would fail for 14 cones (for some node, no edge will keep an invariant zone empty). Thus, a different obstacle avoidance method is needed when the number of cones is not 6.

3.2 Global Routing Strategy

We now have all the pieces in place to describe our routing strategy. Our routing strategy alternates between three phases: while not blocked by an obstacle, we use the classic Θ-routing algorithm. If the current vertex has no valid edges in the cone containing the destination, it must be blocked by a constraint Q. In this case, we enter the obstacle avoidance phase to reach the right endpoint of Q. Once we reach this endpoint, we check which of the two endpoints of Q is closer to the destination. If the closest point to destination is the other endpoint of Q, we enter the *alternative endpoint phase* to reach it. Note that the two endpoints of Q can see each other, so we can route between them using the strategy introduced in [2]. Once we have reached the endpoint of Q that is closest to the destination, we resume classic Θ-routing. We call this alternation between the three phases the *constrained Θ_6-routing strategy.*

3.3 Convergence

We now show that our routing algorithm always reaches the destination. First we give a proof of convergence which greatly overestimates the number of steps needed to reach the destination, but it turns out that first showing that the algorithm always reaches the destination simplifies the proof of bounding the number of steps.

▶ **Lemma 4.** *The constrained Θ_6-routing strategy always reaches the destination within a finite number of steps.*

Proof. By construction, each edge followed during the Θ-routing phase gets closer to the destination. Hence, each Θ-routing phase can consist of at most n steps. Similarly, an obstacle avoidance phase performs at most n steps, since each step brings the boundary of

cone C_0 closer to the endpoint we are routing to. At the end of an obstacle avoidance phase, we may need an alternative endpoint phase which visits each vertex at most once [2].

Thus, in order to show termination it remains to bound the number of alternations between phases. Each invocation of an obstacle avoidance phase is tied to a single constraint Q. We bound the number of times Q can trigger an obstacle avoidance phase. Let z be the endpoint of Q that is closest to t. In order for Q to trigger another obstacle avoidance phase the routing path needs to first reach a vertex v such that v and t are in different halfplanes (with respect to the line containing Q). Since the routing path cannot cross the constraint Q itself, in the routing path between z and v we reach a vertex that is further from t than z is.

Since Θ-routing only gets closer to t, we must perform at least one obstacle avoidance phase with a different constraint Q'. Since an obstacle avoidance phase (together with the possible alternative endpoint phase) always ends at the endpoint z' of Q' that is closest to t, this implies that z' is further away from t than z. Let $Q_1, \ldots, Q_k$ be all the constraints sorted by decreasing distance of their closest endpoint to t. Let z_i be the endpoint of Q_i closest to t. Notice that Q_1 cannot invoke more than one obstacle avoidance phase since there are no constraints whose closest endpoint z_i is further from t than z_1. In general, this ordering implies that Q_i cannot invoke an obstacle avoidance phase more than 2^{i-1} times. Therefore, when there are k constraints, there can be at most $2^k - 1$ invocations of an obstacle avoidance phase. Since we take at most $3n$ steps between two obstacle avoidance steps, the total number of steps is upper bounded by $O(n \cdot 2^k)$. ◀

Note that the above reasoning shows that a single constraint could be visited many times, however, a simple argument shows that each constraint invokes at most one obstacle avoidance phase.

▶ **Lemma 5.** *Let Q be a constraint and let z be the endpoint of Q that is closest to t. Vertex z can be visited as the final vertex of at most one obstacle avoidance or alternative endpoint phase.*

Proof. When we reach z at the end of an obstacle avoidance or alternative endpoint phase, since the constrained Θ_6-routing strategy is memoryless at the end of this phase, it follows the same edge from z every time we reach it. This implies that z cannot be visited twice using an obstacle avoidance or alternative endpoint phase, since otherwise the path would cycle indefinitely, contradicting Lemma 4. ◀

This immediately gives a linear bound on the number of phase changes, implying a quadratic bound on the number of steps. We now use a more detailed analysis of the circumstances in which a vertex may be visited to tighten this further to $O(n)$.

▶ **Lemma 6.** *The constrained Θ_6-routing strategy always reaches the destination in $O(n)$ steps.*

Proof. Consider any vertex v and consider how we reached it. We will show that overall, no vertex is visited too many times.

1) **v is reached during a Θ-routing phase.** Since the routing strategy in this phase is memoryless, we would make the same routing step from v every time we reach it. In particular, this would imply that v cannot be visited twice using a Θ-routing phase (otherwise, the path would cycle indefinitely, contradicting with Lemma 4). Hence, we conclude that v is visited once during a Θ-routing phase during the whole routing algorithm.

2) **v is reached during an avoidance phase of constraint Q.** We consider two subcases:

2.1) v is not an endpoint of Q. Let u be the vertex that initiated the avoidance phase and first consider the case in which Q completely blocks visibility of u in the cone containing t (see Figure 3b). In this situation, the same cone remains empty for all vertices along the path (including v). Note that there can be at most three constraints that fully block visibility of v in some cone. Thus, if v is visited more than three times as part of an obstacle avoidance path, two of them share the same cone. Both of these times, the obstacle avoidance and alternative endpoint phases would end up at z, the endpoint of Q closest to t, contradicting Lemma 5. Thus, we conclude that v can be reached this way at most three times.
It is possible that Q did not block the visibility in the cone completely (i.e., we initiated the obstacle avoidance phase because the edge was invalid, see Figure 3a). This situation is very similar to the case in which visibility was completely blocked. The only difference is that the choice of the edge we follow at v depends on the cone that contained t when we started this obstacle avoidance phase as well as on whether or not v has edges in the two adjacent cones. We again conclude that if v is visited more than a constant number of times in this way, the algorithm would route to the same neighbour of v, eventually ending at the same endpoint of Q and contradicting Lemma 5.

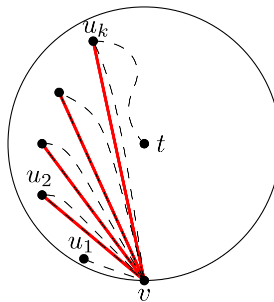

Figure 7 A vertex v can be visited $\Omega(n)$ times as the endpoint not closest to t. This implies that v is the endpoint of many constraints and in all of them it is further away from t than the other endpoint u_2, ..., u_k. For clarity, the disk centred at t passing through v is drawn (as solid black), and a possible routing path that visits v multiple times is also shown (in dashed black).

2.2) v is an endpoint of Q. As argued in Lemma 5, v can only be visited once during the whole execution of the algorithm if it is the endpoint that is closest to t. Similarly, if v is the endpoint that is furthest away from t, we know the algorithm enters the alternative endpoint phase and routes to the opposite endpoint of Q. Note that v could be visited several times this way (see Figure 7). However, notice that v can never be visited twice because of the same constraint Q, as this would imply that we visit the same closest endpoint twice as well, contradicting Lemma 5. Thus, during the entire execution of the algorithm, we can visit at most $3n - 6$ vertices as the endpoint of a constraint that is not closest to t.

3) v is reached during an alternate endpoint phase. Every time a vertex is part of a path in the alternate endpoint phase, Lemma 3 of [2] shows that at least one of its cones is empty.

Hence, excluding case 2.2, each vertex is visited a constant number times. Since case 2.2

adds at most $3n-6$ visited vertices during the entire execution of the algorithm, this implies that a total of $O(n)$ steps are executed as claimed. ◀

▶ **Theorem 7.** *There exists a 1-local $O(1)$-memory routing algorithm for the constrained Θ_6-graph that reaches the destination in $O(n)$ steps.*

Proof. The algorithm is 1-local by construction, since we consider only information about vertices the current vertex is connected to. The Θ-routing phase does not require any memory. The obstacle avoidance phase and alternative endpoint phase store a single vertex each and this information is discarded when the phase ends. Hence, the algorithm requires $O(1)$ memory. Lemma 6 shows that the algorithm terminates in $O(n)$ steps. ◀

4 Routing on the Visibility Graph

We now return our attention to our main goal: routing on the visibility graph. Since in the previous section we presented a routing algorithm for the constrained Θ_6-graph, we first show that we can use this algorithm to route on the visibility graph as well. Afterwards, we also describe how to modify the constrained Θ_6-routing algorithm to route on the visibility graph directly without locally determining the edges of the constrained Θ_6-graph.

We note that, unfortunately, the length of the paths resulting from these two approaches need not be related to the length of the shortest path in the visibility graph. Since we cannot determine locally which endpoint of a constraint is closest to t, the routing algorithms may follow a path to an endpoint arbitrarily far away, preventing us from being competitive.

4.1 Using the Constrained Θ_6-Graph

In order to use the constrained Θ_6-routing algorithm from the previous section, we need to determine locally at a vertex which of its visibility edges are part of the constrained Θ_6-graph. Since it is easy to locally determine at a vertex u if a vertex v is the closest vertex in one of its subcones, we focus on the situation where this is not the case and we thus have to determine at u if it is the closest vertex in one of the subcones of v. Let the *constrained canonical triangle* of v be $\triangle_{vu}$ clipped using the constraints intersecting the boundary of the canonical triangle with one endpoint at u (see Figure 8). Note that we can determine the constrained canonical triangle of v locally at u.

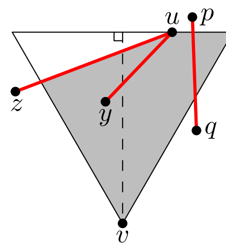

Figure 8 The constrained canonical triangle of v (gray). Constraint uz is used to clip the triangle. Constraint uy does not clip the triangle, since it does not cross the triangle boundary. Constraint pq does not clip the triangle, since it has no endpoint at u.

▶ **Lemma 8.** *Let u and v be two vertices such that v is not the closest vertex to u in any subcone of u. Edge uv is part of the constrained Θ_6-graph if and only if u does not have any visible vertices in the constrained canonical triangle of v.*

Proof. We first note that we can consider the subcone of v that contains u to be the full cone: If the constraint defining the subcone ends in the constrained canonical triangle, Lemma 1 implies that it also contains a vertex visible to u, correctly implying that uv is not an edge. If the constraint does not end in the constrained canonical triangle, the part of the constrained canonical triangle outside the subcone is not visible to u and hence it does not influence the decision at u.

It is easy to see that if u has any visible vertices in the constrained canonical triangle of v, uv is not an edge of the constrained Θ_6-graph: Consider the vertex x such that the smaller angle of ux and uv is minimized. Since the angle is minimized, u is not the endpoint of any constraints intersecting triangle uvx, so we can apply Lemma 1 to uvx. This gives us a vertex inside the constrained canonical triangle that is visible to v. Hence, u is not the closest visible vertex to v and thus uv is not an edge of the constrained Θ_6-graph.

Next we show that if u has no visible vertices in the constrained canonical triangle of v, uv is an edge of the constrained Θ_6-graph. We prove this by contradiction, so assume that uv is not an edge of the constrained Θ_6-graph. This implies that there exists a vertex x in the subcone of v that contains u that is closer to v than u is. Hence, x lies in the constrained canonical triangle. Applying Lemma 1 to uvx gives us a vertex inside the constrained canonical triangle that is visible to u, contradicting that u has no visible vertices in this region. ◀

4.2 Routing Directly on the Visibility Graph

In order to route directly on the visibility graph, instead of at each vertex computing the local neighbourhood in the constrained Θ_6-graph, the constrained Θ_6-routing algorithm needs to be modified. We do this in such a way that the vertices do not need to store any fixed cone orientations.

When a vertex s wants to send a message, it picks an arbitrary cone orientation and stores it in the message it sends. We note that a vertex can pick a different orientation of the cones for each message that it sends and this only requires a constant amount of storage. Since the orientation is stored in the message, vertices do not need to agree on a fixed orientation in advance, as every vertex along the routing path can extract the orientation from the message and use that for its decisions.

Like in the constrained Θ_6-routing algorithm, routing directly on the visibility graph works in three phases: Θ-routing, obstacle avoidance, and alternative endpoint. During the Θ-routing phase a vertex u simply sends the message to the closest vertex in the cone that contains t, again limiting the edges it is allowed to follow to the edges that end in $\triangle_{ut}$.

During the obstacle avoidance phase, we start by routing to either endpoint of the constraint blocking visibility to t. Since we are routing on the visibility graph, Lemma 1 tells us that there is a convex chain of visibility edges to these endpoints. Hence, in order to reach an endpoint of the constraint, we follow one of these convex chains. In order to determine the next edge on the chain at an intermediate vertex m, the message needs to store the predecessor of m on the chain and whether the path should continue to the next clockwise or counter-clockwise edge of m. The next edge along the convex chain at m is the edge that minimizes the angle with the line through m and the predecessor of m in the stored direction.

When we arrive at an endpoint of a constraint, we can determine the location of the other endpoint, since they are connected in the visibility graph. Using this information, we can determine if this constraint is the one that caused the obstacle avoidance phase by checking if it blocks visibility of u to t. If this is the case, we also determine which of the two

endpoints is closer to t. If we are not yet at the endpoint closest to t, we start the alternative endpoint phase, which is now simplified to following the edge in the visibility graph to the other endpoint of the constraint.

▶ **Theorem 9.** *There exists a 1-local $O(1)$-memory routing algorithm for the visibility graph that reaches the destination in $O(n)$ steps.*

Proof. We first note that locality follows from the fact that we only need to consider the neighbours of the current vertex in each of the steps. The memory bound follows from the fact that we need to store only the orientation of the cones in the message, as well as the starting vertex of the obstacle avoidance phase.

It remains to bound the number of steps. This algorithm has properties similar to those of the constrained Θ_6-routing algorithm. First, the Θ-routing phase always gets closer to the destination and thus cannot repeat vertices. This implies that Lemma 4 also holds for this routing algorithm. This in turn implies that a vertex can be the closest endpoint of an obstacle avoidance or alternative endpoint phase at most once. Next, since the obstacle avoidance path is convex, this implies that this path visits a subset of the vertices visited by the obstacle avoidance phase of the constrained Θ_6-routing algorithm. Finally, the alternative endpoint phase consists of at most a single edge, hence this phase too is a subpath of its constrained Θ_6-routing counterpart. Hence, when we compare the path of this routing algorithm to the constrained Θ_6-routing path that uses the same cone orientation, the routing path on the visibility graph is a subpath of the constrained Θ_6-routing path. Hence, it takes at most $O(n)$ steps. ◀

5 Conclusion

We presented the first 1-local $O(1)$-memory routing algorithms for the visibility graph that do not require the computation of a planar subgraph. Unfortunately, our algorithms do not give guarantees on the length of the routing path, only on the number of edges used. Hence, designing an algorithm that is competitive with respect to the shortest path remains open.

Acknowledgements
We thank Luis Barba, Sangsub Kim, and Maria Saumell for fruitful discussions.

References

1 Prosenjit Bose, Rolf Fagerberg, André van Renssen, and Sander Verdonschot. On plane constrained bounded-degree spanners. In *LATIN*, volume 7256 of *LNCS*, pages 85–96, 2012.
2 Prosenjit Bose, Rolf Fagerberg, André van Renssen, and Sander Verdonschot. Competitive local routing with constraints. *Journal of Computational Geometry*, 8(1):125–152, 2017.
3 Prosenjit Bose, Matias Korman, André van Renssen, and Sander Verdonschot. Constrained routing between non-visible vertices. In *COCOON*, 2017.
4 Prosenjit Bose and André van Renssen. Upper bounds on the spanning ratio of constrained theta-graphs. In *LATIN*, volume 8392 of *LNCS*, pages 108–119, 2014.
5 Ken Clarkson. Approximation algorithms for shortest path motion planning. In *STOC*, pages 56–65, 1987.
6 Gautam Das. The visibility graph contains a bounded-degree spanner. In *CCCG*, pages 70–75, 1997.
7 Sudip Misra, Subhas Chandra Misra, and Isaac Woungang. *Guide to Wireless Sensor Networks*. Springer, 2009.
8 Harald Räcke. Survey on oblivious routing strategies. In *Mathematical Theory and Computational Practice*, volume 5635 of *LNCS*, pages 419–429, 2009.