Tight Linear Lower Memory Bound for Local Routing in Planar Digraphs

Maia Fraser*

Abstract

Local *geometric* routing algorithms with logarithmic memory are in widespread use in modern MANETs (mobile ad hoc networks). Formally, these are algorithms executed by an agent traveling from node to node in a geometric graph, using only *local geometric information* at each node, leaving no traces and carrying $O(\log n)$ bits of memory. For the case of undirected graphs, theoretical and practical aspects of such algorithms have been extensively developed since Face Routing (FR) was proposed in 1999 for planar embedded graphs. By contrast, the corresponding problem in geometric digraphs has been relatively little studied. In CCCG'08, the author and co-authors showed a lower bound of $\Omega(n)$ bits on the memory of local geometric routing algorithms for planar embedded digraphs.

The purpose of the present paper is to show this lower bound is tight under two models: either node identifiers (possibly coordinates) are known to come from an $O(n)$ size space and no marks may be left, or else $O(\log n)$ bits suffice to identify a node (as assumed in FR) and pebbles may be left. We describe an analog to FR which under either model guarantees delivery in polynomial time in strongly connected planar embedded digraphs. In the first model, memory of $O(n)$ bits is transported, in the second $O(\log n)$ bits are transported and $O(n)$ pebbles are left. By contrast, for non-geometric algorithms of the second model a tight lower memory bound of $\Omega(n \log n)$ bits follows from work of Ilcinkas and Fraigniaud with an exponential runtime algorithm. Our work thus provides a first example confirming that geometry simplifies routing in digraphs.

1 Introduction

Face Routing (FR), proposed in a CCCG'99 paper by Kranakis, Singh and Urrutia, was the first routing algorithm to use geometric and purely local information. It guarantees delivery in time $O(n)$ for planar embedded (undirected) graphs of size n, while transporting only $O(\log n)$ bits memory[1]. It opened a new era in routing at a time when both wireless ad hoc networks (where global connectivity information is not available) and global positioning devices (which provide real-time position information) were becoming commonplace. The idea is simple: to get from node s to node t the algorithm walks one-by-one around the faces which meet the segment st, until t is reached. While only applying to graphs with no edge-crossings, the simplicity and robustness of the algorithm made it canonical and it was modified and extended in many ways so that today most practically occurring MANETs may be handled by some derivative of FR, and it remains at the base of some of the most commonly used algorithms in this class today.

By contrast very little is known for the case of geometric *directed graphs* (digraphs). In certain special cases of planar embedded digraphs, local geometric routing algorithms with logarithmic memory are known to exist: in Eulerian graphs (in which in- and out-degrees coincide and are constant) and in outerplanar graphs (in which a single face contains all vertices) [2]. But these are very restricted classes and even within the class of planar embedded digraphs there is no hope of a logarithmic memory algorithm like FR: in CCCG'08 we showed [5] there exist planar embedded digraphs in which correct local geometric routing *requires* $\Omega(n)$ *bits memory*. In that paper, we considered only algorithms which do not leave traces, however our proof can be adapted to cover algorithms which do leave traces, in which case the *required memory* is the combined number of pebbles and transported bits (see Section 3.1).

The aim of the present paper is theoretical[2]. We show the linear lower memory bound is tight under two models: either node identifiers (possibly coordinates) come from an $O(n)$ size space and no marks are left or else $O(\log n)$ bits identify a node (as assumed in FR) and marks (pebbles) may be left. We describe an analog of FR which correctly routes in all strongly connected planar embedded digraphs under these models. In the first model, memory of $O(n)$ bits is transported and no marks are left; in the second, $O(\log n)$ bits are transported and $O(n)$ pebbles are left.

We remark that a significant amount of theoretical work does exist on routing in *non-geometric* digraphs

*Department of Computer Science, University of Chicago, maia@cs.uchicago.edu

[1] the minimal memory for a routing algorithm, if it by definition carries at least the destination ID.

[2] We concluded in [5] that uni-directional links should be avoided in MANETs. In fact, there is another practical issue. Unless we allow multicasting, node v will not send to node u unless it knows of u's existence and given a uni-directional arc from v to u this is impossible to achieve directly by communication between the two nodes: nodes will not be locally aware of their downstream neighbours.

under other models. In particular the problem of directed st-connectivity is usually posed assuming a JAG model (see for example the survey article [1]). JAG's are automata working in a team and able to teleport (jump) to teammates. This is very different from our setting. An instance of our agent may not be teleported but must be *transmitted* and this is by definition only possible to direct neighbours. Our second model, however, is the geometric analog of a model considered by Ilcinkas and Fraignaud in [3]. They assume an agent which cannot jump but is able to leave pebbles at nodes and transport some memory. They show such an agent needs $\Omega(n \log d)$ bits of memory to explore a digraph with maximum out-degree d but *no vertex labeling*, even if it can use a linear amount of pebbles. Since no geometric information is present, by the adversary argument this is also a lower bound on routing (the destination would be marked instead of specified by ID). They give such an algorithm with exponential runtime. By contrast, our algorithm which has access to geometry (and hence node ID's) can successfully route in polynomial time leaving $O(n)$ pebbles and transporting $O(\log n)$ bits.

2 Directed Face Routing Algorithm

Let G be a directed graph of size n embedded in the plane. The main strategy of the algorithm is the same as that of FR:

Input: source s and destination t
Procedure:

1. Traverse anti-clockwise the face which is entered by the segment st at s.
2. If t is visited then STOP.
3. Else if a node s' such that $d(s', t) < d(s, t)$ is visited then $s \leftarrow s'$.
4. Go to step 1.

The process of traversing a face, however, is much more arduous than in the undirected case.

3 Results

Assuming either of the two models defined above, we will show:

Theorem 1 *There is a local geometric algorithm,* Directed Face Traversal*, which uses $O(n)$ bits memory and traverses a given face F of a strongly connected planar embedded digraph G.*

This then implies:

Corollary 2 *Directed FR is a local geometric algorithm transporting $O(n)$ bits memory which guarantees delivery in any strongly connected planar embedded digraph G.*

Indeed, using Directed Face Traversal, Directed FR is guaranteed to reach t by the usual argument: since G is embedded in the plane, when the segment st enters a face F it must meet the boundary of F again either to exit F or to arrive at t; thus, one of the conditions in steps 2. and 3. is guaranteed to hold and either we will stop at t or the next iteration begins with a strictly reduced distance $d(s, t)$. Moreover, this local geometric algorithm transports only $O(\log n)$ bits memory of its own (to record coordinates of s and t) besides the memory of Directed Face Traversal so the over all memory requirement remains $O(n)$ bits.

Observation 1 *For simplicity, we will describe the algorithm in terms of the second model - leaving $O(1)$ pebbles per node. In the case of the first model[3], we may associate $O(1)$ bits to each node using an array of total size $O(n)$ bits, and so by transporting this memory we can simulate the algorithm written for the second model.*

As an additional result, in Section 3.1, we briefly address the extension of the lower bound from [5] to the second model. Sections 4 and 5 are then devoted to describing Directed Face Traversal and proving Theorem 1.

3.1 Lower bound

In this section we show that the lower bound of [5], which we established for agents which do not leave pebbles, also applies to agents leaving pebbles: for such agents, the sum of the bits transported and pebbles left must be $\Omega(n)$.

Our proof in [5] was based on a simulator of local geometry for a special class $\mathcal{C}$ of graphs K_{x^n}, $n \in \mathbb{N}$, where K_{x^n} is defined using the initial n-bit substring x^n of a Kolmogorov random bit string x. Such graphs were called *kinked embedded locks*. We refer the reader to [5] for more detail. The important aspect, for our purposes is that each graph K_{x^n} has a left-most vertex u_0 and a right-most vertex w (both on the x-axis) and any routing or traversal algorithm $\mathcal{A}$ will have to at some time T travel exactly the path $u_0, u_1, \ldots, u_n, w$.

[3] We make two remarks. First, the assumption of $O(\log n)$ identifiers is also made in standard FR when coordinates are used as identifiers. Indeed, if nodes were arranged on the real line with exponentially increasing spacing (e.g. distance 2^k between k'th and $k+1$'st nodes) then coordinates with $\Omega(n)$ bits would be needed in order to distinguish nodes; such an example is usually ruled out. Second, if either node ID's or coordinate pairs come naturally from key spaces which are large they could be hashed first; we assume we start with (possibly virtual) node ID's which are integers in a range $[0, Cn]$ for some constant C.

The argument in [5] shows that given input the state of $\mathcal{A}$ at time T, one can simulate geometry and make use of $\mathcal{A}$ on the simulated geometric graph to output the string x^n. Moreover, the total memory overhead for this process is only $O(\log n)$. This is the memory needed to run the simulator and co-ordinate tasks and it excludes the memory $M(n)$ transported by $\mathcal{A}$ and also the input (which is a state of $\mathcal{A}$ and so also bounded above by $M(n)$). The Kolmogorov randomness of x implies at least n bits of memory for any algorithm outputing x^n and so one concludes that transported memory of $\mathcal{A}$ must be $\Omega(n)$ bits. We will use a similar argument for slightly different graphs.

To define these, choose any integer constant $c > 1$ and consider kinked embedded locks K_{x^c}, $K_{(x^c)'}$, $K_{(x^c)''}$, ... formed from the initial c-bit substring x^c of x, the subsequent c-bit substring $(x^c)'$, and so on. Then form a new graph, denoted $SK_n(x)$, defined as the series composition of the n graphs $K_{x^c}, K_{(x^c)'}, \ldots, K_{(x^c)^{(n-1)}}$ (with u_0 and w as terminals in each). Let π_i denote the path in $SK_n(x)$ which was labeled $u_0, u_1, \ldots, u_n, w$ in $K_{(x^c)^{(i)}}$. The size of $SK_n(x)$ is a linear function of n. Denote it $L(n)$. Now, define the class $\mathcal{C}^* = \{SK_n(x) \colon n \in \mathbb{N}\}$. These are bounded-degree planar digraphs and we will prove:

Theorem 3 *Any deterministic traversal or routing algorithm which is correct on graphs of the class $\mathcal{C}^*$, is allowed to leave pebbles and uses only local information must satisfy $P(n) + M(n) \in \Omega(n)$ where $P(n)$ is the number of pebbles left and $M(n)$ the transported memory in the graph $SK_n(x)$ of size $\Theta(n)$ (even if the algorithm uses geometric information).*

Proof. If $\mathcal{A}$ is correct for $\mathcal{C}^*$ then it must, at some time T, travel the concatenated path $\rho = \pi_0\pi_1 \cdots \pi_{n-1}$ in $SK_n(x)$. We will assume only one kind of pebble can be left (and at most one is left on each vertex). The argument for a constant number of pebble types is essentially the same[4]. Consider the n subgraphs $K_{(x^c)^{(i)}}$, $i = 0, \ldots, n-1$ and suppose there are pebbles on exactly $k = k(n)$ of these at time T (so $P(n) \geq k(n)$). Let R be a bitstring of length ck which is the concatenation of k strings of bitstrings of length c indicating – for those k values of i such that $K_{(x^c)^{(i)}}$ has pebbles – on which of the vertices of each π_i there is a pebble. Additionally form a single n-bit string S whose i'th bit is 1 if $K_{(x^c)^{(i)}}$ has pebbles and 0 if $K_{(x^c)^{(i)}}$ does not (so that in total S has k bits set to 1).

Now, define an algorithm which takes as input the state of $\mathcal{A}$ at time T (as in [5]) *but also* the strings R and S. If the i'th bit of S is 0, this algorithm should simulate geometry using the output of $\mathcal{A}$ and thus traverse a simulated π_i (exactly as in [5]) but if the i'th bit of S is 1, it should read from R in order to simulate geometry with pebbles and thus traverse a simulated π_i. Just as in [5], in this way it will travel a virtual version of ρ and can be used to output the string x^N, where $N = nc$. Let $W(N)$ denote an upper bound on the working memory of such a process. As in [5] we have $W(N) \in O(\log N)$.

Fix some $\epsilon \in (0, 1 - 1/c)$ and let R be a sufficiently large integer that $W(N) < \epsilon N$ for all $N > R$. By Kolmogorov randomness of x we have:

$$\begin{aligned} 2M(n) + ck + n + W(N) &> N \\ 2M(n) + ck + n &> N - W(N) \\ &> (1-\epsilon)N \\ 2M(n) + ck &> \left(1 - \epsilon - \frac{1}{c}\right) N \end{aligned}$$

Thus,

$$\begin{aligned} cM(n) + ck &> \left(1 - \epsilon - \frac{1}{c}\right) N \\ M(n) + k &> \beta n \\ P(n) + M(n) &> \beta n \end{aligned}$$

where $\beta = 1 - \epsilon - \frac{1}{c}$. This is positive by our choice of ϵ, and so we conclude that $P(n) + M(n) \in \Omega(n)$. □

We remark that alternatively, one may consider the same class of graphs as in [5], namely kinked embedded locks of various sizes (for a given Kolmogorov random x) and record the position of pebbles (along the up and down simulated vertices) by two sequences of inter-pebble distances n_i and m_i. A similar argument to that of [5] shows that either this record or $M(n)$ must be $\Omega(n)$. Using the inequality of arithmetic and geometric means one can show that the former implies $P(n) \in \Omega(n)$, yielding a result analogous to Theorem 3 (but for kinked embedded locks).

4 Terminology

First we fix some standard terminology for embedded digraphs. By *edge* we mean an edge of the undirected graph G' obtained by forgetting the directions of arcs of G. By *face* we mean a connected component of the complement of G' in the plane. A *walk* will refer to a walk of G'; i.e., a sequence of vertices $v_0, v_i, \ldots, v_k$ such that either $v_{i-1}v_i$ or v_iv_{i-1} is an arc of G for each $i : 1 \leq i \leq k$. The walk is said to be *directed* if arc orientation is respected, i.e. each $v_{i-1}v_i$ is actually an arc of G. A walk is closed if $v_0 = v_k$. The vertices of a walk may in principle coincide; we say the walk is *simple* if they do not. The fact the digraph G is *strongly connected* means that for any two nodes, s and t, there exists a directed walk starting at s and ending at t. Every face has a unique *boundary walk* starting at a vertex v on the boundary of the face, namely the closed walk consisting of the nodes

[4] There would be several bitstrings like R.

which are encountered when following the face boundary in its canonical boundary orientation, starting from v. This orientation is *to the left* for an observer standing within the face. Boundary walks are in general not directed walks.

Since G is embedded there is a natural cyclical ordering of edges at each vertex: draw a small circle around v, travel around it in the clockwise direction and cyclically order the edges incident at v by the cyclical order in which they are encountered. Suppose uv is an incoming arc at v then we denote by $\mathrm{succ}(uv)$ the next outgoing arc in this cyclical ordering. Given a set M of marked vertices one may analogously define $\mathrm{succ}_M(uv)$ as the next outgoing arc whose head belongs to M; we assume succ_M returns NULL if there are no such outgoing arcs.

This allows one to define a walk starting from a given arc a by iteratively taking $\mathrm{succ}(a)$. If a walk $v_0, \ldots, v_k$ is thus defined by succ_M where M is the vertex set of the walk, we say the walk is *left-free*, since the condition is equivalent to there being no edges of the walk incident on the left side of the walk. Boundary walks are always left-free (since there are no edges at all between uv and vw)[5]. Simple walks are certainly left-free.

We now introduce two notions which will simplify our discussion of face traversal.

Definition 1 *A closed walk $v_0, v_1, \ldots, v_k = v_0$ is said to* **self-cross** *if some edge $v_{i-1}v_i$ is incident at $v_k = v_i$ and lies strictly to one side of the walk at v_k, while v_iv_{i+1} lies strictly to the other. When a closed walk β is not self-crossing we define $\mathcal{C}(\beta)$ to be the collection of* **faces it encloses***: those faces from which a curve may be traced which first meets β on the left and not the right side of β.*

This is illustrated in Figure 1.

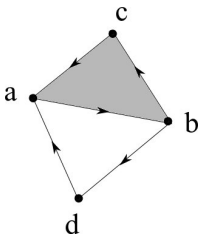

Figure 1: The shaded face is enclosed by the walk a, b, c, a. On the other hand, the walk a, b, d, a encloses a collection of two faces: the shaded one and the outer face.

Suppose β is non self-crossing and F is a face *not* in $\mathcal{C}(\beta)$ but *adjacent* to an edge $v_{i-1}v_i$ of β. It lies necessarily on the right of β and when one replaces the sub-walk $v_{i-1}v_i$ of β with the remainder of the boundary walk of F (also a walk from v_{i-1} to v_i) one obtains a new non self-crossing walk β' which encloses one more face than did β. For example, in Figure 1, the unshaded triangle is a face not enclosed by a, b, c, a, and so using it we can produce the walk a, d, b, c, a which encloses both triangular faces.

Definition 2 *Given a directed walk α define its* **outer shell** *$\mathrm{shell}(\alpha)$ to be the (non self-crossing) closed directed walk which includes α as a sub-walk and encloses a minimal number of faces.*

To see this is well-defined, suppose there are two non-identical closed directed walks $u, v, w_1, \ldots, w_k = u$ and $u, v, w'_1, \ldots, w'_k = u$ both enclosing a minimal number M of faces. Starting from v let the last vertex where they coincide be w_ℓ, so $w_i = w'_i$ for $i \leq \ell$ and $w_{\ell+1} \neq w'_{\ell+1}$. Now let $w_m = w'_p$ be the first vertices which coincide for $m, p > \ell$. We then have two directed walks from $w_\ell = w'_\ell$ to $w_m = w'_p$ which meet only at their endpoints: $w_\ell, w_{\ell+1}, \ldots, w_m$ and $w'_\ell, w'_{\ell+1}, \ldots, w'_p$. By replacing one sub-walk with the other we can obtain a closed directed walk which includes u, v and encloses fewer than M faces, a contradiction.

5 Directed Face Traversal Algorithm

Idea The idea of this algorithm is to traverse the face boundary β until an opposing arc a is encountered, say at vertex v, and then to traverse the outer shell σ of the arc a (or the outer shell of a longer directed walk ending in a). This shell traversal cannot in general be done in one iteration, rather our algorithm will on the i'th iteration produce a non self-crossing directed closed walk δ_i such that $\mathcal{C}(\delta_i) \subset \mathcal{C}(\sigma)$ and such that $\mathcal{C}(\delta_i) \subsetneq \mathcal{C}(\delta_{i+1})$. Eventually therefore we must have $\delta = \sigma$. Suppose this walk starts and ends at the head of a. We now use, as an exit from this area, the outgoing arc uw whose tail (on σ) most closely preceeds v (in the order on σ) such that there are no outgoing arcs on the right of σ between uw at u and the incoming arc at v and thus we know that this part of σ (which includes a) is part of the face boundary β which we seek. And then we continue. This process must terminate since we always increase the visited portion of β.

5.1 Internal memory

Under the first model, the algorithm stores three binary arrays of size $O(n)$:

$$\text{INNER[], OUTER[], and MID[],}$$

all initialized to zero. Under the second model it uses pebbles of types INNER, OUTER, MID. In

[5]For this to hold, it is important that the cyclical ordering be done in clockwise fashion when one considers boundary walks with canonical boundary orientation.

both models, it also records three node identifiers: JOIN, NEW_EXIT, and EXIT, each of size $O(\log n)$ as well as the coordinates of s and t. We will assume there is a simple logspace subroutine which verifies if a given edge crosses the segment st.

In addition, Directed Face Traversal records its state which is one of the following: WALK_BDRY, BACKTRACK, FIND_EXIT, GO_EXIT. Finally there is a Boolean flag TENTATIVE, initially set to false.

5.2 State diagram

The state diagram for the algorithm is cyclical:

$$\begin{aligned}\text{WALK\_BDRY} &\to \text{BACKTRACK}\\ &\to \text{FIND\_EXIT}\\ &\to \text{GO\_EXIT} \quad \to \text{WALK\_BDRY}\end{aligned}$$

5.3 Invariant

In the Appendix we prove that the following property is an invariant, always true upon entering a new state.

Path property:

1. The nodes marked OUTER form a *directed* simple walk, τ, from s to EXIT.
2. The nodes marked INNER form a left-free walk from s to JOIN which passes through EXIT. We denote by η the sub-walk up to EXIT and by λ the sub-walk after EXIT.
3. The nodes marked INNER and MID can be used to form a *directed* walk from EXIT to s (take MID when INNER cannot be taken).
4. τ concatenated with $-\eta$ (which goes from EXIT to s) is non self-crossing and all nodes marked MID belong to faces enclosed by this concatenated curve.

This is illustrated in Figure 2.

Explanation Borrowing the notation used to describe the *Idea* of the algorithm at the start of Section 5, the point of the Path property is that it allows the current δ_i to be defined by pebbles; namely, δ_i is 'almost' traversed by starting at s and following τ then $-\eta$ back to s. The reason for the word 'almost' is that such a walk will in general not be a directed walk: although τ is directed so that one can indeed follow it in this direction, $-\eta$ may not be. The small walks marked MID are detours which allow one to get back to s from EXIT (i.e. each piece of MID by-passes an unsuitably directed part of $-\eta$). Thus a more accurate statement is that by following OUTER till EXIT and then INNER+MID back to s one traverses δ_i.

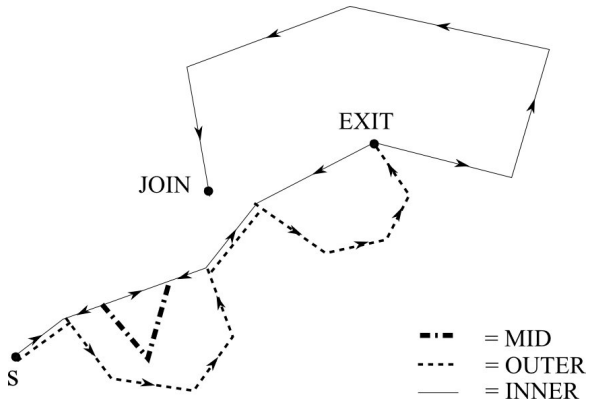

Figure 2: Path property

The condition, in the Path property, that pebbles form a walk (which can be followed) is to be interpreted using the following fact.

Lemma 4 *Any left-free walk β may be uniquely followed (from some vertex u onward) by a memory-less (local) geometric algorithm as long as all its vertices are marked by pebbles, no adjacent non-β vertices are marked, and the algorithm is given a starting arc at u.*

This holds because if we start at a pebble-marked node u and are given the first arc uv to follow, then we may iteratively follow the path defined by succ_M where M is the set of pebble-marked vertices.

Now, assume we have established the Path property as an invariant. We will *upon entry into each state* be able to follow τ (via OUTER), λ (via INNER) or almost $-\eta$ (via INNER+MID). In our pseudocode we write `follow` to indicate we are following one of these walks appealing to these principles. We now describe the main loop of Directed Face Traversal.

5.4 Main loop

We describe the algorithm in **high-level pseudocode**, suitable for proving correctness. In the interests of clarity we will refer to the curves η, τ and λ from the Path property, as well as a vertex called PRIOR which is defined below. **All of these walks as well as PRIOR can however be determined locally.** In the Appendix we give lower-level pseudocode which does this and implements the same actions as specified below.

It remains to define the high-level variable PRIOR. Assuming the path property, the walk along INNER from EXIT to JOIN meets either:

1. the walk INNER only, and on the left side of that walk, in a section bypassed by MID (a) or not (b).
2. the walk OUTER only, and on the right side of that walk

3. a vertex of both walks, either on the left or right side (cases (a) and (b) resp.).

In all cases, we define PRIOR to be the vertex belonging to *both walks* which most closely preceeds (or equals) the vertex[6] JOIN (in the walk that was met).

This is illustrated in Figure 3. The convention is the same as in Figure 2: the dotted directed walk represents OUTER and it ends at EXIT. After that point the walk along INNER continues from EXIT to JOIN, where it meets either INNER, or OUTER, or both. This is indicated by a ray with its arrow pointing to the place of return. The possible cases are illustrated and labeled as above: 1ab, 2, 3ab.

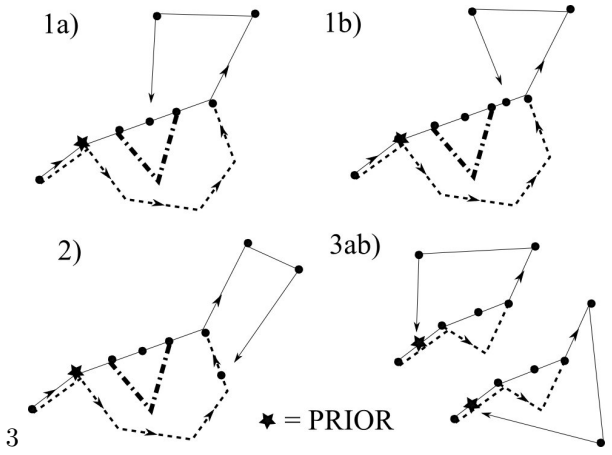

Figure 3: PRIOR is shown for the various cases

We now make two remarks. They concern respectively the setting of PRIOR (in the state BACKTRACK) and the use of PRIOR (in the state FIND_EXIT).

Observation 2 *It is possible for the agent to deduce which case has occurred by walking provisionally to the left once JOIN occurs. A case by case analysis shows the logic involved. The key point is that the orientation of arcs along INNER between intersections with OUTER or MID is prescribed and so contradictions must eventually arise if the provisional assumption is wrong. We assume that PRIOR can be found, i.e. the agent can walk to this vertex and determine that it is the vertex PRIOR. Moreover, at that moment it knows which way τ is oriented.*

Observation 3 *Once the agent is at PRIOR, by walking along τ and then $-\eta$ (using detours provided by MID), always keeping track of the last outgoing arc on the right until PRIOR is reached again, the location of a new exit can be found. By strong connectivity some outgoing arc must exist and this method of choosing it will ensure there are no outgoing arcs between it and PRIOR on the walk just described.*

The actions to perform in the main loop depend on the state. If at any point the following implicit exit condition becomes valid the main loop is exited and the algorithm places the agent at whichever endpoint u or v is closer to t and returns.

Exit condition: the arc uv crosses the line segment st.
Initialization: Set EXIT and JOIN to s and let $a = sv$ be the first arc which exits s to the right of the line segment st.
Main loop:

- WALK_BDRY:
  1. Use succ() starting along the arc a (at EXIT) to build a left-free walk, marking all its vertices with INNER. Stop when this walk first meets an existing INNER or OUTER vertex and let JOIN be the intersection vertex.
  2. Change state to BACKTRACK.
- BACKTRACK:
  1. `Follow` INNER+MID or OUTER and thereby determine and reach the vertex PRIOR (see Observation 2).
  2. Change state to FIND_EXIT.
- FIND_EXIT:
  1. In cases 1, 2, 3a, walk along τ then λ, always recording the last outgoing arc encountered on the right (see Observation 3). Let NEW_EXIT be the vertex where this outgoing arc is found (it will be PRIOR itself if no later outgoing arc was found). In case 3b let NEW_EXIT be PRIOR. When NEW_EXIT = PRIOR some additional special handling is required (see Appendix).
  2. Change state to GO_EXIT.
- GO_EXIT:
  1. Return to NEW_EXIT correcting all markers so the Path property will be preserved[7] upon setting EXIT $\leftarrow$ NEW_EXIT. Set EXIT $\leftarrow$ NEW_EXIT and also JOIN $\leftarrow$ NEW_EXIT. Set a to succ(b) at EXIT, where b is the outgoing arc of the new $-\eta$ at EXIT.
  2. Change state to WALK_BDRY.

[6] In the first case, where JOIN is a repeated vertex of INNER, we refer to its first occurrence.

[7] We want to move EXIT to a new vertex, NEW_EXIT, along the combined walk τ with $-\eta$ and thus we will be changing the breakpoint between τ and $-\eta$. To retain the Path property the pebbles for INNER, MID and OUTER must be adjusted accordingly. Details are given in the Appendix.

5.5 Correctness

Proof. Let F be the face determined by sv from the initialization. We will prove that Directed Face Traversal traverses F. We assume that the Path property holds upon entry to all states.

Consider the first v_{i-1}, v_i on the boundary walk β of F which is *not an arc*. Then $v_i v_{i-1}$ is an arc. We claim the agent will eventually reach v_i. Let α be a maximal directed sub-walk of $-\beta$ which ends with v_i, v_{i-1}. Let $\sigma = \text{shell}(\alpha)$ be the outer shell of α.

Let μ be the concatentation of τ with $-\eta$. We prove by induction (on the iteration), that at the start of the main loop we have $\mathcal{C}(\mu) \subsetneq \mathcal{C}(\sigma)$ as long as the agent has not yet arrived v_i. To see this, assume μ is as at the start of the k'th iteration and it satisfies this condition. We claim its new value at the start of the $k+1$'st iteration will also do so. We consider the new λ which will be formed during the WALK_BDRY state. Suppose the agent will not arrive at v_i by the start of the $k+1$'st iteration. Let $F_1, F_2, F_3, \ldots$ be the faces encountered on the left when following the new λ and $a_1, a_2, a_3, \ldots$ the corresponding arcs which are incoming to λ on the left. If the first vertex of this walk after EXIT does *not* belong to a face in $\mathcal{C}(\sigma)$ then since $\mathcal{C}(\mu) \subset \mathcal{C}(\sigma)$, the walk μ must coincide with σ at EXIT. But following μ back to v_{i-1} we do not meet any outgoing arcs on the right. Thus μ and σ must coincide all the way back to v_{i-1} and so μ passes through v_i, a contradiction. Thus the first vertex past EXIT on the new λ does indeed belong to a face in $\mathcal{C}(\sigma)$. And so σ encloses F_1. But given the orientation of a_1, this implies σ encloses the outer shell of a_1 and hence encloses F_2, and so on. This continues until we reach the end of the new λ (at the new JOIN) because on the left side of λ there are only incoming arcs. Thus at the start of the $k+1$'st iteration we will have $\mathcal{C}(\mu) \subset \mathcal{C}(\sigma)$ and this inclusion must be strict, otherwise μ would equal σ and the agent would return to v_i.

Moreover, we have also shown that $\mathcal{C}(\mu)$ increases on each such iteration. This cannot go on forever so eventually the agent must reach v_i. This means it will advance along β past a_1. We repeat the above argument using (in place of τ and η) τ_v and η_v, the sub-walks of τ and η starting at v, where v is the next value of EXIT after the agent has returned to v_i (we know EXIT belongs to both walks). This process continues and since the agent thus advances by at least one edge along β every time, eventually it must traverse all of β. □

6 Conclusions

We have shown that the lower linear bound on memory for local routing algorithms that was proved in [5] is tight. Our algorithm mimics Face Routing but involves a much more involved face traversal procedure. Nevertheless the linear memory requirement of the algorithm is a significant reduction in the $\Omega(n \log n)$ memory required for local routing in *non-geometric* digraphs proven by Ilcinkas and Fraigniaud [3], where they also provide algorithms with exponential runtime or else polynomial runtime and higher memory. Our geometric algorithm's worst case runtime (hop-length) is polynomial. Indeed on a given iteration of Directed Face Traversal, each arc may be traversed at most $O(1)$ times and there are $O(n)$ iterations for a given face (since the node EXIT will be different on each) and $O(n)$ faces in total.

7 Acknowledgements

The author thanks the referees for their very useful comments and suggestions, in particular for seeking clarification on the models used and prompting Figure 1.

References

[1] G. Barnes, J. Edmonds. Time-space lower bounds for directed st-connectivity on graph automata models. *SIAM J. on Computing*, 27(4):1190–1202,1998

[2] E. Chavez, S. Dobrev, E. Kranakis, J. Opatrny, L. Stacho, J. Urrutia. Route discovery with constant memory in oriented planar geometric networks. *Networks*, 48(1):7-15, 2006.

[3] P. Fraigniaud and D. Ilcinkas. Digraphs exploration with little memory. *21st Symposium on Theoretical Aspects of Computer Science (STACS04)*, LNCS 2296:246-257, 2004.

[4] E. Kranakis, H. Singh, J. Urrutia. Compass routing in geometric graphs. *11th Canadian Conference on Computational Geometry (CCCG'99)*, 51-54, 1999.

[5] M. Fraser, E. Kranakis, J. Urrutia. Memory requirements for local geometric routing and traversal in digraphs. *20th Canadian Conference on Computational Geometry (CCCG'08)*, 2008.

Appendix

Lower-level pseudocode for parts of Directed Face Traversal. Suppose we are at vertex u and given next vertex v.

- WALK_BDRY:
  1. Do:
     (a) Let $vw = \text{succ}_+(uv)$.
     (b) Add an INNER pebble to u
     (c) $u \leftarrow v$
     (d) $v \leftarrow w$

     while u is neither INNER nor OUTER.
  2. JOIN $\leftarrow u$
  3. Change state to BACKTRACK.
- BACKTRACK:
  1. If u is both INNER and OUTER: let v be the vertex to the left along OUTER, set the flag TENTATIVE[8], go to step 4.
  2. If u is only INNER: attempt to `follow` INNER+MID to the right until a vertex that is both INNER and OUTER is found, but if this fails (assert: we have been travelling on INNER and not seen MID) then `follow` INNER back to MID and let the arc leading along MID be a, then `follow` INNER+MID in the direction of a until a vertex that is both INNER and OUTER is found, then let u be the new current vertex, and v the vertex to the left along OUTER, go to step 4.
  3. If u is only OUTER: `follow` OUTER to the right until a vertex that is also INNER is found, let a be the arc to the left along INNER, `follow` INNER+MID along a until OUTER is reached, then let u be new current vertex, and v the vertex to the left along OUTER, go to step 4.
  4. Set PRIOR $\leftarrow u$.
  5. Change state to FIND_EXIT.
- FIND_EXIT:
  1. (assert: u = PRIOR and this is a node of both INNER and OUTER)
  2. if not TENTATIVE then set NEW_EXIT $\leftarrow$ PRIOR and
     - `follow` OUTER along v until possibly EXIT is reached, after which `follow` INNER; at each node u during this walk if there are outgoing arcs on the right then NEW_EXIT $\leftarrow u$. When JOIN is reached, stop.
     - If NEW_EXIT = PRIOR then return to PRIOR and go to step 5
     - Otherwise go to step 4.
  3. if TENTATIVE then `follow` OUTER until either
     - OUTER diverges from INNER on the wrong (left) side, then
       * Set NEW_EXIT $\leftarrow u$, go to step 5.
     - or OUTER will diverge from INNER on the correct (right) side, then reset TENTATIVE to false, set v to previous node and go to step 2.
     - or OUTER and INNER have not diverged but there is an outgoing arc uv on the left
       * Set NEW_EXIT $\leftarrow u$, go to step 4.
     - or OUTER and INNER have not diverged but there is currently an outgoing arc on the right then reset TENTATIVE to false, set v to previous node and go to step 2.
  4. Change state to GO_EXIT.
  5. (special handling when forced to exit along an arc which is part of $-\eta$):
     - `follow` INNER+MID along v until OUTER returns to INNER on the left; record the vertex at which it diverged as TMP_JOIN and the vertex at which it returns as TMP_PRIOR. Imitate the 'if not TENTATIVE' case (with its flow control) but using TMP_JOIN and TMP_PRIOR in place of JOIN and PRIOR respectively.
- GO_EXIT:
  1. If not at NEW_EXIT then (assert: we are at JOIN) return to PRIOR by following INNER+MID.
  2. Depending on whether we are at NEW_EXIT or PRIOR follow the existing paths and change pebbles accordingly so the Path property will be preserved (see paragraph below).
  3. Set EXIT $\leftarrow$ NEW_EXIT and also JOIN $\leftarrow$ NEW_EXIT. Set a to succ(b) at EXIT, where b is the outgoing arc of the new $-\eta$ at EXIT.
  4. Change state to WALK_BDRY.

In all of the above, when following a walk as indicated, **node updating** is assumed to occur as in WALK_BDRY so that node u is always the current one and node v the next one.

Path property is an invariant The way in which markers must be corrected (i.e. pebbles placed and removed) during state GO_EXIT is not specified in the above pseudocode. We claim it can be done in such a way that the Path property will hold true upon entry into each new state. Note that it holds (trivially) at initialization. Suppose it holds at the start of a run of the main loop.

Pebbles are only changed in the states WALK_BDRY and GO_EXIT, so the Path property is trivially preserved during BACKTRACK and FIND_EXIT. In WALK_BDRY, INNER pebbles are added but only until reaching either OUTER or INNER which occurs at JOIN; thus, the Path property is preserved during that state as well.

To analyze the state GO_EXIT,and see that pebbles can be changed suitably, note that we begin this state at the

[8] At this point we do not know on which side we have reached the common INNER/OUTER path; we tentatively assume from the left and wait for contradictions.

vertex JOIN or (in case 3b only) at NEW_EXIT. If not the last case, we will follow INNER+MID back to PRIOR.

Consider the cases for PRIOR shown in Figure 3. In cases 1 and 2, it is possible, starting from PRIOR, to follow OUTER either till EXIT and then INNER till JOIN, or OUTER directly to JOIN, while at some time passing NEW_EXIT. From JOIN, it is then possible to follow INNER+MID to PRIOR. In case 3a, none of this applies as PRIOR = JOIN.

To maintain the Path property in cases 1 and 2, it thus suffices to ensure that during the walk from PRIOR – up until JOIN – we leave or remove pebbles so that all vertices up to NEW_EXIT are labeled OUTER, and all vertices after this are labeled INNER. From JOIN back to PRIOR, we must then change pebbles so that INNER+MID will define a directed walk from NEW_EXIT back to s (it suffices to deal with the portion from JOIN until PRIOR since the portion up until JOIN was just handled). In case 1 this will mean severing a piece of the old INNER which forms a chord on the new INNER (this can be done by removing a pebble at each end), and in case 1a) additionally changing some INNER to MID. In case 2 this will mean converting some OUTER to MID.

In case 3a, we will need to change OUTER to INNER as we follow OUTER from NEW_EXIT to EXIT, then we will need to return to JOIN along INNER+MID, severing this walk from the other marked walks (by removing a pebble at each end).

Finally, consider case 3b (where we entered the state GO_EXIT at vertex NEW_EXIT). By removing OUTER pebbles from the part of τ after NEW_EXIT, we will have a new τ which indeed starts at s and ends at EXIT (upon setting EXIT $\leftarrow$ NEW_EXIT). Conditions 2, 3 of the Path property will also be satisfied if we convert the portion of τ between NEW_EXIT and JOIN to INNER and the portion between JOIN and EXIT to MID.