Finding Short Right-Hand-on-the-Wall Walks in Graphs

Stefan Dobrev[1], Jesper Jansson[2],
Kunihiko Sadakane[4], and Wing-Kin Sung[3]

[1] SITE, University of Ottawa, Canada
sdobrev@site.uottawa.ca
[2] Department of Computer Science and Information Systems,
The University of Hong Kong, Hong Kong
jjansson@cs.hku.hk
[3] School of Computing, National University of Singapore
ksung@comp.nus.edu.sg
[4] Department of Computer Science and Communication Engineering,
Kyushu University, Japan
sada@csce.kyushu-u.ac.jp

Abstract. We consider the problem of *perpetual traversal* by a single agent in an anonymous undirected graph G. Our requirements are: (1) deterministic algorithm, (2) each node is visited within $O(n)$ moves, (3) the agent uses no memory, it can use only the label of the link via which it arrived to the current node, (4) no marking of the underlying graph is allowed and (5) no additional information is stored in the graph (e.g. routing tables, spanning tree) except the ability to distinguish between the incident edges (called *Local Orientation*).

This problem is unsolvable, as has been proven in [9, 28] even for much less restrictive setting. Our approach is to somewhat relax the requirement (5). We fix the following traversal algorithm: *"Start by taking the edge with the smallest label. Afterwards, whenever you come to a node, continue by taking the successor edge (in the local orientation) to the edge via which you arrived"* and ask whether it is for every undirected graph possible to assign the local orientations in such a way that the resulting perpetual traversal visits every node in $O(n)$ moves.

We give a positive answer to this question, by showing how to construct such local orientations. This leads to an extremely simple, memoryless, yet efficient traversal algorithm.

1 Introduction

The problem of searching and exploring an unknown environment is a fundamental problem with applications ranging from robot navigation to searching the WWW. As such, it has been extensively studied under many different assumptions about the environment. Typically, either a geometric setting has been assumed (see e.g. [7, 11, 29]) or the environment is modeled as a graph with moves permitted only along the edges.

A. Pelc and M. Raynal (Eds.): SIROCCO 2005, LNCS 3499, pp. 127–139, 2005.

The graph setting has been extensively investigated [1, 3, 6, 12, 13, 14, 16, 20, 25] under many different assumptions (directed vs undirected graphs, anonymous nodes vs nodes with distinct identities) and goals (different variants of the exploration, focusing on time complexity, minimizing the memory requirements).

An important aspect of any solution is its memory requirements – both in the exploring agent(s) and in the network environment itself (can the agent mark the nodes, what is the nature of the marks and how many can be used?). Since the desire is to have simple and cost efficient agents, and there can be many of them independently operating in the network, it is of practical importance to limit both the local memory of the agents, and their ability to mark the network.

An extreme case of minimizing memory requirements is to limit the agents memory to a constant number of bits. This can be modeled as exploring a graph using finite automata and has been intensively studied in 70's [9, 24, 26, 28]. The strongest result is due to Rollik [28]: No finite group of finite automata can cooperatively explore all cubic planar graphs (see [21] for more recent results). This means that we either have to allow the agents to use more memory, resort to randomization or provide some structural information that restricts the set of graphs we have to traverse.

If we do not place strict restrictions on the local memory, single pebble is sufficient to explore the graph [5], even for anonymous directed graphs [1].

Another possibility is to drop the determinism requirement – it is known that a random walk of length $O(n^3)$ would, with high probability, visit every node [22]. Trying to regain determinism led into research of derandomized random walks, searching a sequence (called Universal Traversal Sequence) of edge labels that would traverse all graphs in a given class. While many interesting results have been achieved [2, 4, 23, 27], the memory requirements are not always clear and the traversal times are rather high, especially with respect to our goal of $O(n)$ time.

Research most closely related to our result considers using structural information to improve the time and/or memory complexity of graph traversal. First results (concerning exploration of a labyrinth using a compass) are due to Blum and Kozen [8]. Later, Flocchini et al [18] introduced a more general notion of Sense of Direction and proved that traversal can be performed using $O(n)$ messages/agent moves [17]. Fraigniaud et al [19] have shown that interval routing scheme can be used to achieve the same. In fact, given a spanning tree, the graph can be traversed using $O(n)$ moves. Pelc and Panaite [25] studied the impact of having a map of the graph on the efficiency of graph exploration/traversal. All these solutions, though, use quite a lot of memory – either in the network (routing tables in [19], remembering the spanning tree) or in the agent (storing Sense of Direction and remembering visited nodes in [17], remembering the network map in [25]).

[1] The graph must be strongly connected and an upper bound on the number of nodes must be known. The time complexity, while polynomial, is quite high, though.

In this paper we propose to use the capabilities already present in the system – namely the ability to distinguish the links incident to a node – to store the information allowing efficient traversal: A common requirement in point-to-point networks is that the nodes can distinguish between incident edges (often called *Local Orientation*). This is normally done by giving the edges incident to a node v numbers $1, 2, \ldots, d_v$, where d_v is the degree v. Usually, there is no assumption on how is this edge ordering chosen. In this paper we propose to choose the local orientations in a very specific way. Whether this costs us any memory depends on how are the lower level communication layers implemented, but it is quite conceivable that if done at the time of network construction/initialization, it comes essentially for free.

We fix the following traversal algorithm: *"Start by taking the edge labelled 1. Afterwards, whenever you come to a node, continue by taking the successor edge (in the local orientation) to the edge via which you arrived"* and ask whether it is for every undirected graph possible to to choose the local orientations in such a way that the resulting perpetual traversal visits every node in $O(n)$ moves.

We give a positive answer to this question, by showing how to construct such local orientations. This leads to an extremely simple, memoryless, yet efficient traversal algorithm.

The paper is organized as follows: In Section 2 we introduce the notation used and give basic definitions and properties. Section 3 contains the main result of the paper. In Section 4 we discuss how to adapt to dynamically changing networks. Section 5 contains open questions, as well as a brief comparison to Sense of Direction.

2 Preliminaries

Let G be a simple, connected, undirected graph. Let d_v denote the degree of vertex v in G. We assume that each vertex can distinguish edges incident to it, by having assigned unique label to each incident edge[2]. This labeling (denoted π_v and called *local orientation*) and defines cyclical ordering of the edges incident to v. Let $succ_v(e)$ denote the successor of e in π_v. Denote by $\boldsymbol{G}$ the symmetric directed graph obtained from G by replacing each undirected edge by two directed edges in opposite direction. For each directed edge $\boldsymbol{e} = \{u, v\}$ we define the *underlying* edge to be the undirected edge (u, v).

We want to find a (short, possibly non-simple) cycle $\boldsymbol{C}$ in $\boldsymbol{G}$ containing all vertices of $\boldsymbol{G}$ and satisfying the right-hand rule: If $e_1 = \{u, v\}$ and $e_2 = \{v, w\}$ are two successive edges of $\boldsymbol{C}$ then $e_2 = succ_v(e_1)$. Since the local orientations can be rotated so that the underlying edge with label 1 is used in outgoing direction at every vertex, the algorithm *"Start by using edge labelled* 1 *and then follow the successor edges"* traverses exactly $\boldsymbol{C}$. We call such a cycle a *witness cycle* for G.

[2] More precisely, to endpoints of the edges incident to v. Each edge gets two labels, one at each endpoint. Note that these two labels can be different and unrelated.

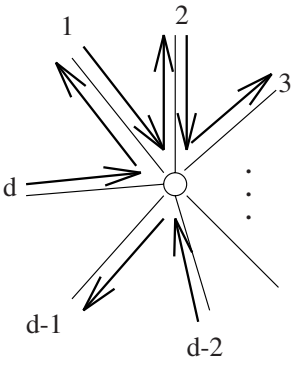

Fig. 1. Ordering two bidirectional, two incoming and two outgoing underlying edges

Let $\boldsymbol{H}$ be a subgraph of $\boldsymbol{G}$ containing all its vertices. For a vertex v, denote by b_v, i_v and o_v the number underlying edges incident to v used by $\boldsymbol{H}$ in both directions, only incoming and only outgoing, respectively. Let d'_v denote the number of underlying edges used by $\boldsymbol{H}$, i.e. $d'_v = b_v + i_v + o_v$.

The overall idea is to find a graph $\boldsymbol{H}$ containing all the edges of a witness cycle $\boldsymbol{C}$ and then to figure out how to label the edges of $\boldsymbol{H}$ to obtain single witness cycle. The following definition captures the right-hand traversal property that must be satisfied at each vertex of a witness cycle:

Definition 1. *We say that a vertex v is* RH-traversable *if there exists a local orientation π_v in v such that for each directed edge of $\boldsymbol{H}$ incoming to v via an underlying edge e there exists an outgoing edge in $\boldsymbol{H}$ leaving v via the underlying edge that succeeds e in π_v.*

We call such ordering a witness *ordering for v.*

If $b_v = d_v$, the vertex v is said to be *saturated*. The following Lemma characterizes the necessary local conditions for existence of the witness ordering:

Lemma 1. *v is RH-traversable if and only if v is saturated or $i_v = o_v > 0$.*

Proof. The if *direction:* If v is saturated, any ordering of the underlying edges will do. In the second case, choose any ordering in which bidirectional edges are labelled $1, 2, \ldots, d_v$, forming one compact block followed by an outgoing edge. All remaining unidirectional edges are placed as a block preceding the bidirectional block; the edges of this unidirectional block alternate directions, with the last edge preceding the bidirectional block being incoming – see Figure 1.

The only if *direction:* First of all, note that since the successor function is injective, i_v must be equal to e_v. Furthermore, note that if a bidirectional underlying edge is followed by an unused underlying edge, the RH-traversal property is violated. Finally, if v is not saturated and there are no unidirectional underlying edges, there must be such bidirectional edge. □

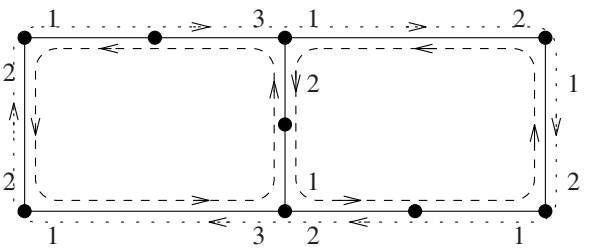

Fig. 2. Each vertex is RH-traversable, but the witness orderings of the vertices define several cycles and no cycle spans the whole graph

Note that if a witness cycle exists, each vertex is RH-traversable. The converse is not necessarily true, as the witness orderings of the vertices might define several cycles, none of which span the whole graph – see Figure 2.

3 Main Result

First note that if G is Hamiltonian, a Hamiltonian cycle can be chosen as witness cycle and we get $\boldsymbol{C}$ of length n. (The RH-traversability is trivially satisfied as each node is visited only once.) Therefore, for the rest of the paper, we assume G is not Hamiltonian.

The main idea of our approach is to

- first find a subgraph $\boldsymbol{H}$ such that each vertex is RH-traversable and
- then figure out how to connect the edges of $\boldsymbol{H}$ to form a single witness cycle $\boldsymbol{C}$.

One obvious possibility is to set $\boldsymbol{H} = \boldsymbol{G}$, i.e. use all edges bidirectionally. From Lemma 1 we know that each vertex can be made RH-traversable. Moreover, since each node of $\boldsymbol{G}$ is of even degree, $\boldsymbol{G}$ has an Eulerian cycle. However, it is not immediately clear how to choose the local edge orderings to satisfy RH-traversability and simultaneously result in a single cycle. Another problem is that the resulting cycle would be of length $O(|E|)$, not $O(n)$.

This forces us to take the following more refined approach:

1. Construct a directed graph $\boldsymbol{H}$ such that
   (a) The undirected graph H' induced by the bidirectional edges of $\boldsymbol{H}$ is connected and contains all vertices of $\boldsymbol{G}$.
   (b) Each vertex v of $\boldsymbol{H}$ is either saturated or has exactly two unidirectional underlying edges, one incoming and one outgoing.
   (c) $\boldsymbol{H}$ contains $O(n)$ edges.
2. For each vertex of $\boldsymbol{H}$ define a witness ordering. (These orderings define one or more cycles in $\boldsymbol{H}$.)
3. Locally modify the orderings in some nodes in order to merge these cycles into one supercycle $\boldsymbol{C}$ containing all vertices, while maintaining RH-traversability.

The first property of $\boldsymbol{H}$ ensures that if we connect all vertices of $\boldsymbol{H}$ into a single cycle $\boldsymbol{C}$, it will span the whole graph. The second property ensures that each vertex is RH-traversable and the third guarantees that $\boldsymbol{C}$ is of size $O(n)$. Note again that the second property does not guarantee by itself that the witness orderings of the vertices define single cycle spanning $\boldsymbol{H}$ (see Figure 2) – we really need to do the third step.

3.1 Constructing Subgraph $\boldsymbol{H}$

The following algorithm constructs the graph $\boldsymbol{H}$:

Algorithm CONSTRUCT $\boldsymbol{H}$:

1: $\boldsymbol{H} \leftarrow \emptyset$; {*Unless stated otherwise, when an edge is added to* $\boldsymbol{H}$*, it is added bidirectionally*}.
2: **repeat**
3: Find in G a cycle C_i such that it does not contain two consecutive vertices that are both in $\boldsymbol{H}$.
4: Add C_i to $\boldsymbol{H}$.
5: **until** no such cycle C_i can be found.
6: Add to $\boldsymbol{H}$ all vertices of G not yet in $\boldsymbol{H}$, together with all their incident edges.
7: If $\boldsymbol{H}$ is not connected, add some bridging edges to make it connected. (Note that because of the previous step, there is no need to add vertices.)
8: Let $\tilde{G} = (\tilde{V}, \tilde{E})$ be the graph induced by all yet unused underlying edges.
9: **repeat**
10: **repeat**
11: Find a vertex v of degree 1 in $\tilde{G}$.
12: Add to $\boldsymbol{H}$ the edge incident to v in $\tilde{G}$; remove v from $\tilde{G}$.
13: **until** There is no vertex of degree 1 in $\tilde{G}$
14: Find in $\tilde{G}$ a cycle C'_i.
15: Add C'_i to $\boldsymbol{H}$ *unidirectionally* (in arbitrary direction) and remove all the vertices of C'_i from $\tilde{G}$.
16: **until** no such cycle C'_i can be found.
17: Add the remaining edges of $\tilde{G}$ (a forest, possibly empty) to $\boldsymbol{H}$.

Lemma 2. $\boldsymbol{H}$ *uses at most* $5n$ *underlying edges.*

Proof. We prove the lemma by observing the following facts:

- **Fact 1.** *The number of edges added on lines 2... 5 is at most* $2n-3k_1$*, where* k_1 *is the number of resulting connected components.*

  Proof: We charge the cost of each cycle to its new vertices. Since at least half of the vertices of the cycle are new, each one of them is charged at most 2 edges. In addition, the vertices of the first cycle (of size at least 3) in each component are charged 1 edge each (as they are all new).
- **Fact 2.** *The number of edges added on line 6 is at most* $2n-2$.

Proof: Let us divide those edges into E_1 – the edges between nodes added in line 6 and E_2 – the edges between the new and the "old" (added before line 6) nodes. The graph induced by edges in E_1 does not contain cycle, otherwise a cycle of all-new vertices would have been found in line 3. Similarly, the graph induced by the edges in E_2 does not contain cycle, otherwise that cycle (containing exactly half new vertices) would have been found in line 3.

– **Fact 3.** *The number of edges added on line 7 is at most $k_2 < k_1$, where k_2 is the number of connected components after line 6.*

Proof: Straightforward.

– **Fact 4.** *The number of edges added on lines 9...17 is at most n.*

Proof: For each edge that is added to $\boldsymbol{H}$ one vertex is removed from $\tilde{G}$. □

Let us call the cycles added in line 15 *relief* cycles.

Lemma 3. *$\boldsymbol{H}$ is connected, contains all vertices of G and each vertex is either saturated or it lies on exactly one relief cycle.*

Proof. The first two properties follow by construction from lines 6 and 7. If node v does not lie on a relief cycle, then either it has never been in $\tilde{G}$ or it was removed from there in lines 10...13 or 17. Either case can happen only if v is (or becomes) saturated. v cannot be in more then once relief cycle, because it is removed from $\tilde{G}$ when its relief cycle is added to $\boldsymbol{H}$. □

Lemma 4. *The complexity of Algorithm* Construct $\boldsymbol{H}$ *is $O(n^3)$.*

Proof. We will show how to implement line 3 (finding a cycle that does not contain consecutive old vertices) in time $O(n^2)$. This straightforwardly results in $O(n^3)$ time for the loop on lines 2..5.

The loop on lines 9..16 can be executed only $O(n)$ times, and the statements in its body can easily be implemented in $O(n^2)$ time. As the remaining steps can be easily implemented in $O(n^2)$ time, the overall complexity would be $O(n^3)$.

The line 3 can be implemented in $O(n^2)$ in the following way: Define graph $G' = (V', E')$ as follows: (1) V' contains all old vertices (the vertices $in\boldsymbol{H}$) and one vertex for each connected component of the graph induced by the new vertices. (2) An edge $(u, v) \in E'$ where u is an old vertex and v corresponds to a connected component of new vertices if and only if there is an edge in G connecting u to a vertex from the connected component corresponding to v.

Note that a cycle in G' defines a cycle in G satisfying the requirements of line 3. Since $O(n^2)$ time is sufficient for constructing G' as well as for finding a cycle in it, line 3 can be implemented in time $O(n^2)$. □

3.2 Constructing Witness Cycle C

Once we have $\boldsymbol{H}$, the local ordering of underlying edges in each vertex is initialized according to the construction from the proof of Lemma 1. We know (from Lemmas 3 and 1) that such witness ordering exists for each vertex v; however, we

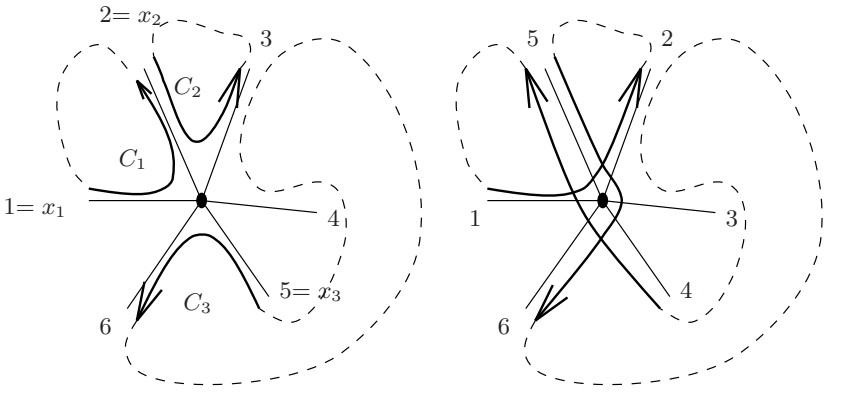

Fig. 3. Applying rule *Merge3*

may not get a single cycle spanning all vertices (see Figure 2). In the next step, we combine the resulting cycles until we get one such cycle, while maintaining RH-traversability. To achieve that, we use the following rules:

Rule Merge3: Let v be a node incident to at least three different cycles C_1, C_2 and C_3. Let x_1, x_2 and x_3 be underlying edges in v containing incoming edges for cycles C_1, C_2 and C_3, respectively (x_1, x_2 and x_3 can be unidirectional or bidirectional). The ordering of the edges in v which makes the successor of x_2 become the successor of x_1, successor of x_3 become the successor of x_2 and successor of x_1 become successor of x_3 and keeps the relative order of the remaining edges the same (see Figure 3) connects the cycles C_1, C_2 and C_3 into one cycle, while remaining a witness ordering for v (because the original ordering was).

Rule EatSmall: Fix an arbitrary ordering γ of the cycles. Let C_1 be the smallest non-simple cycle in this ordering and let v be a vertex appearing in C_1 at least twice which is also incident to a different cycle C_2 such that $\gamma(C_1) < \gamma(C_2)$. Let x and y be underlying edges containing incoming edges of C_1 and C_2 in v, respectively; let z be the underlying edge containing the incoming edge by which C_1 returns to v after leaving via the successor of x. If z is successor of y, choose a different x. Modify the ordering of the edges in v as follows: (1) the successor of x becomes the new successor of y, (2) the old successor of y becomes the new successor of z, (3) the old successor of z becomes the new successor of x and (4) the order of the remaining edges does not change – see Figure 4.

Lemma 5. *Applying the rule* EatSmall *results in transfer of one loop of edges from cycle C_1 to C_2, while maintaining RH-traversability.*

Proof. Straightforward from construction. □

The overall strategy of applying these rules is as follows:

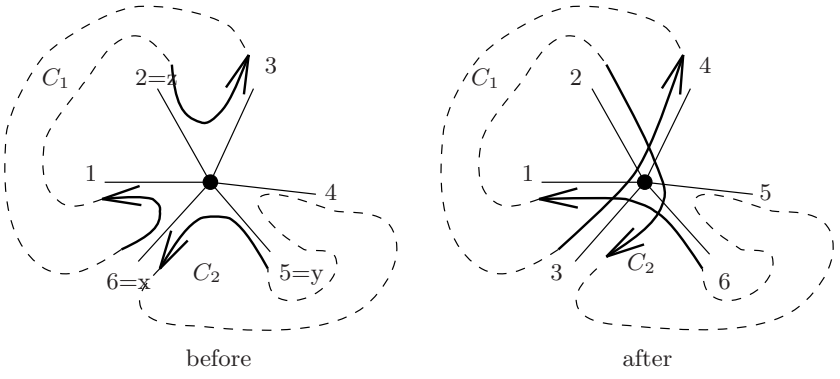

Fig. 4. Applying rule *EatSmall*

Algorithm MergeCycles:

1: **repeat**
2: **while** rule *Merge3* can be applied **do**
3: Apply rule *Merge3*.
4: **end while**
5: Apply rule *EatSmall*.
6: **until** neither *Merge3* nor *EatSmall* can be applied
7: (Optional) remove all simple cycles

Lemma 6. *The algorithm* MergeCycles *terminates in* $O(n^3)$ *time.*

Proof. Note that initially there are at most $10n/3$ cycles ($\boldsymbol{H}$ has at most $10n$ edges and each cycles has at lest 3 edges). Since rule *Merge3* decreases the number of cycles by 2 and rule *EatSmall* does not increase the number of cycles, rule *Merge3* can be applied at most $5n/3$ times during the whole execution of algorithm MergeCycles.

Since rule *EatSmall* transfers come edges from the smallest non-simple cycle to a bigger (in γ) cycle, it can be successively (without intervening *Merge3*) applied only $O(n)$ times (remember, the number of edges is at most $10n$). This means that the rule *EatSmall* can be applied only $O(n^2)$ times.

In order to apply rule *Merge3*, we need to find a vertex incident to three different cycles. In order to apply rule *EatSmall*, we need to find the smallest non-simple cycle and a repeated vertex on this cycle which is incident to a bigger cycle. Both tests can be straightforwardly done in $O(n)$ time by traversing and marking the different cycles, resulting in $O(n^3)$ overall complexity for algorithm MergeCycles. □

Lemma 7. *If neither* Merge3 *nor* EatSmall *can be applied,* $\boldsymbol{H}$ *consists of a single non-simple cycle spanning all the vertices and of a set of pairwise vertex disjoint simple cycles.*

Proof. Before proceeding with the proof, let us remind you that the graph H' consisting of bidirectional edges of $\boldsymbol{H}$ is connected and contains all vertices of $\boldsymbol{H}$. (Follows directly from lines 6 and 7 of the construction of $\boldsymbol{H}$.)

Let C_1 be the smallest non-simple cycle at the moment when neither rule can be applied. Let E' be the set of all underlying edges which are not used by C_1, but are incident to C_1. Each edge of E' is used by a single cycle, otherwise rule *Merge3* could be applied.

Assume first that all these edges are unidirectional. Then all edges of H' are used by C_1, because H' is connected and E' would separate it. Since H' contains all vertices, C_1 does as well. No underlying bidirectional edges outside C_1 means that all other cycles are pairwise edge-disjoint. However, they must be also vertex disjoint, because the rule *Merge3* cannot be applied and C_1 contains all vertices. Similarly, all other cycles are simple, since C_1 contains all vertices and rule *EatSmall* cannot be applied.

To complete the proof, we prove by contradiction that there is no bidirectional edge incident to C_1, but not belonging to C_1. Assume the opposite. From the properties of H' we get that either there is a vertex $v \in C_1$ incident to both an external bidirectional edge and a bidirectional edge in C_1 (contradiction, as that would allow rule *EatSmall* to apply, since the outside bidirectional edge can only belong to a larger non-simple cycle) or that each of the vertices of C_1 is incident to an outside bidirectional edge (in such case either C_1 is simple cycle or *EatSmall* can be applied – contradiction in both cases). □

Now we are ready for the main theorem:

Theorem 1. *There exists a witness cycle* $\boldsymbol{C}$ *of length at most* $10n$ *covering all vertices of* G.

Proof. From Lemma 6 we know that eventually no rule can be applied. From Lemma 7 we get that at that moment there exists single non-simple cycle (which we choose as $\boldsymbol{C}$) covering all vertices. Since this cycle uses each directed edge of $\boldsymbol{H}$ at most once, from Lemma 2 we get the length property. □

Note 1: We can remove from $\boldsymbol{H}$ all the remaining simple cycles to get a graph containing only $\boldsymbol{C}$. The RH-traversability will obviously not be violated.

Note 2: In each vertex we can rotate the edges in such way that the edge labelled 1 will always be in $\boldsymbol{C}$.

From Lemmas 4 and 6 we get the main complexity theorem:

Theorem 2. *Witness cycle of length at most* $10n$ *covering all vertices of* G *can be constructed in time* $O(n^3)$.

4 Adapting to Dynamic Topology Changes

In previous section we described how to initialize the network so that the RH-traversal leads to an efficient traversal. In this section we show how to maintain

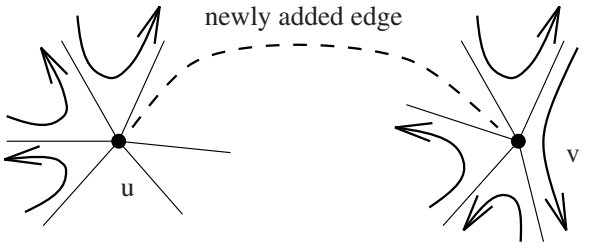

Fig. 5. Adding an edge connecting two unsaturated vertices

this property even in the case of topology changes. More precisely, we show how to modify the local orientations in case of adding new vertices and edges to the network.

In order to simplify the algorithm, we assume the only topology changes are (1) connecting a new vertex to the existing graph by a single edge, and (2) adding an edge between two existing vertices. More complex changes can easily be implemented using a sequence of these basic operations.

If a new edge (u, v) connects two unsaturated vertices, it can be inserted between the outgoing and incoming underlying edges without violating RH-traversability – see Figure 5. However, if one of the vertices is saturated, we have to use the new edge in both directions. Inserting the edge at position 1 ensures that it is a successor for some incoming edge, and that it has a successor outgoing edge, i.e. both u and v remain RH-traversable. However, this might result in splitting $\boldsymbol{C}$ into two cycles. That can be easily corrected by applying rule *EatSmall* while possible, as we known that *Merge3* is not applicable. Note that it is sufficient to perform the test only at node u, as we know that if there are indeed two cycles, they meet at u and v.

Algorithm ADAPT:

1: { *Edge* (u, v) *(and possibly a new vertex* v*) has been added.*}
2: **if** either u or v are saturated **then**
3: Insert (u, v) as a an edge used bidirectionally in $\boldsymbol{C}$ to location 1 in the local orientations of u and v.
4: **else**
5: Insert (u, v) as an edge unused in $\boldsymbol{C}$ to a place between outgoing and incoming underlying edges in u and v – see Figure 5.
6: **end if**
7: Apply rule *EatSmall* at u while possible.
8: (Optional) Remove all non-Hamiltonian simple cycles from.

By construction and from Lemma 7 we get:

Theorem 3. *Applying algorithm* ADAPT *after each topology change will maintain* $\boldsymbol{C}$ *as the witness cycle containing every node of the graph. Moreover, at most* 2 *directed edges are added to* $\boldsymbol{C}$ *for each edge newly added to* G.

After adding n' new vertices and m' new edges, the resulting witness cycle is guaranteed to grow by no more then $2m'$ edges. If m' becomes too high, recomputing the witness cycle might be necessary to bring the length back to $O(|V|)$. Our approach does not handle vertex and/or edge removal, as the graph could become disconnected and/or severe non-local changes might be needed.

5 Conclusions

We have shown that for every connected simple undirected graph the local orientations in the vertices can be chosen in a way that creates a right-hand rule cyclical walk of length at most $10n$ covering all vertices. Moreover, we have shown how to maintain this property even when more vertices and edges are added to the graph. Still, several questions remain unanswered:

- Can the length of the walk be further reduced? What is the lower bound?
- Can the time complexity of finding a witness cycle of length $O(n)$ be reduced from $O(n^3)$? How?
- What is the time complexity of finding the *shortest* witness cycle? How to find it?
- The only property of the walk we were interested in was its length. Suppose we want to use these walks for mutual search [10] instead of traversal. How do we design the local orientations so that performing RH-walk leads to efficient mutual search?
- How to compute the local orientations in a distributed manner? What can be done if the nodes are anonymous?
- How to react to node or edge removal?

We can view our construction as a way to create globally consistent edge labelling. Comparing it to another globally consistent edge labelling, namely Sense of Direction, we observe that our approach uses minimal number of different labels and allows much simpler and more memory efficient graph traversal. However, Sense of Direction is more general and can be used in ways our construction cannot (e.g. avoiding entering a node - see [15].)

References

1. S. Albers and M. R. Henzinger. Exploring unknown environments. *SIAM Journal on Computing,* 29:1164–1188, 2000.
2. N. Alon, Y. Azar, and Y. Ravid. Universal sequences for complete graphs. *Discrete Appl. Math.*, 27(1-2):25–28, 1990.
3. B. Awerbuch, M. Betke, and M. Singh. Piecemeal graph learning by a mobile robot. *Information and Computation*, 152:155–172, 1999.
4. A. Bar-Noy, A. Borodin, M. Karchmer, N. Linial, and M. Werman. Bounds on universal sequences. *SIAM J. Comput.*, 18:268–277, 1989.
5. M. Bender, A. Fernandez, D. Ron, A. Sahai, and S. Vadhan. The power of a pebble: Exploring and mapping directed graphs. In *Proc. of STOC 98*, pages 269–287, 1998.

6. M. Bender and D. K. Slonim. The power of team exploration: two robots can learn unlabeled directed graphs. In *Proc. of FOCS 94*, pages 75–85, 1994.
7. A. Blum, P. Raghavan, and B. Schieber. Navigating in unfamiliar geometric terrain. *SIAM Journal on Computing*, 26:110–137, 1997.
8. M. Blum and D. Kozen. On the power of the compass (or, why mazes are easier to search than graphs). In *Proc. of FOCS 78*, pages 132–142, 1978.
9. L. Budach. Automata and labyrinths. *Math. Nachrichten*, pages 195282, 1978.
10. Harry Buhrman, Matthew Franklin, Juan A. Garay, Jaap-Henk Hoepman, John Tromp, and Paul Vitányi. Mutual search. *J. ACM*, 46(4):517–536, 1999.
11. X. Deng, T. Kameda, and C. H. Papadimitriou. How to learn an unknown environment i: The rectilinear case. *Journal of the ACM*, 45:215–245, 1998.
12. X. Deng and C. H. Papadimitriou. Exploring an unknown graph. *Journal of Graph Theory*, 32(3):265–297, 1999.
13. A. Dessmark and A. Pelc. Optimal graph exploration without good maps. In *Proc. 10th European Symposium on Algorithms (ESA'02)*, pages 374–386, 2002.
14. K. Diks, P. Fraigniaud, E. Kranakis, and A. Pelc. Tree exploration with little memory. *Journal of Algorithms*, 51:38–63, 2004.
15. S. Dobrev, P. Flocchini, G. Prencipe, and N. Santoro. Finding a black hole in an arbitrary network: optimal mobile agents protocols. In *Proc. of PODC 2002*, pages 153–162, 2002.
16. C.A Duncan, S.G. Kobourov, and V.S.A Kumar. Optimal constrained graph exploration. In *12th ACM-SIAM Symposium on Discrete Algorithms (SODA'01)*, pages 807–814, 2001.
17. P. Flocchini, B. Mans, and N. Santoro. On the impact of sense of direction on communication complexity. *Information Processing Letters*, 63(1):23–31, 1997.
18. P. Flocchini, B. Mans, and N. Santoro. Sense of direction: definition, properties and classes. *Networks*, 32(3):165–180, 1998.
19. P. Fraigniaud, C. Gavoille, and B. Mans. Interval routing schemes allow broadcasting with linear message-complexity. *Journal of Distributed Computing*, 14(4):217–229, 2001.
20. P. Fraigniaud and D. Ilcinkas. Digraph exploration with little memory. In *21st Symp. on Theoretical Aspects of Computer Science (STACS'04)*, 2004.
21. P. Fraigniaud, D. Ilcinkas, G. Peer, A. Pelc, and D. Peleg. Graph exploration by a finite automaton. In *Proc. of MFCS 2004*, pages 451–462, 2004.
22. U. Friege. A tight upper bound on the cover time for random walks on graphs. *Random Structures and Algorithms*, 6(1):51–54, 1995.
23. S. Hoory and A. Wigderson. Universal traversal sequences for expander graphs. *Inf. Process. Lett.*, 46(2):67–69, 1993.
24. D. Kozen. Automata and planar graphs. In *Proc. of Fundations Computatial Theory (FCT 79)*, pages 243–254, 1979.
25. P. Panaite and A. Pelc. Impact of topographic information on graph exploration efficiency. *Networks*, 36.
26. M.O. Rabin. Maze threading automata. Technical Report Seminar Talk, University of California at Berkeley, October 1967.
27. O. Reingold. Undirected st-connectivity in log-space. *Electronic Colloquium on Computational Complexity*, 94, 2004.
28. H.A. Rollik. Automaten in planaren graphen. *Acta Informatica*, 13:287–298, 1980.
29. N. Roo, S. Hareti, W. Shi, and S. Iyengar. Robot navigation in unknown terrains: Introductory survey of length,non-heuristic algorithms. Technical Report ORNL/TM12410, Oak Ridge National Lab., 1993.