More Efficient Periodic Traversal in Anonymous Undirected Graphs

Jurek Czyzowicz, Stefan Dobrev, Leszek Gasieniec, David Ilcinkas, Jesper Jansson, Ralf Klasing, Ioannis Lignos, Russell Martin, Kunihiko Sadakane, Wing-Kin Sung

More Efficient Periodic Traversal in Anonymous Undirected Graphs

Jurek Czyzowicz*, Stefan Dobrev**, Leszek Gąsieniec***, David Ilcinkas†, Jesper Jansson‡, Ralf Klasing†, Ioannis Lignos§, Russell Martin***, Kunihiko Sadakane¶, and Wing-Kin Sung‖

Abstract. We consider the problem of *periodic graph exploration* in which a mobile entity with (at most) constant memory, *an agent*, has to visit all n nodes of an arbitrary undirected graph G in a periodic manner. Graphs are supposed to be anonymous, that is, nodes are unlabeled. However, while visiting a node, the robot has to distinguish between edges incident to it. For each node v the endpoints of the edges incident to v are uniquely identified by different integer labels called *port numbers*. We are interested in the minimisation of the length of the exploration period.

This problem is unsolvable if the local port numbers are set arbitrarily, see [1]. However, surprisingly small periods can be achieved when assigning carefully the local port numbers. Dobrev et al. [2] described an algorithm for assigning port numbers, and an oblivious agent (i.e., an agent with no persistent memory) using it, such that the agent explores all graphs of size n within period $10n$. Providing the agent with a constant number of memory bits, the optimal length of the period was proved in [3] to be no more than $3.75n$ (using a different assignment of the port numbers). In this paper, we improve both these bounds. More precisely, we show a period of length at most $4\frac{1}{3}n$ for oblivious agents, and a period of length at most $3.5n$ for agents with constant memory. Finally, we give the first non-trivial lower bound, $2.8n$, on the period length for the oblivious case.

* Département d'Informatique, Université du Québec en Outaouais, Gatineau, Québec J8X 3X7, Canada. E-mail: jurek@uqo.ca.

** Institute of Mathematics, Slovak Academy of Sciences, Dubravska 9, P.O.Box 56, 840 00, Bratislava, Slovak Republic. E-mail: stefan@ifi.savba.sk.

*** Department of Computer Science, University of Liverpool, Ashton Street, Liverpool, L69 3BX, U.K. E-mail: {L.A.Gasieniec,Russell.Martin}@liverpool.ac.uk. L. Gąsieniec partially funded by the Royal Society International Joint Project, IJP - 2007/R1. R. Martin partially funded by the Nuffield Foundation grant NAL/32566, "The structure and efficient utilization of the Internet and other distributed systems".

† LaBRI, CNRS and Université de Bordeaux, 351 cours de la Liberation, 33405 Talence, France. E-mail: {david.ilcinkas,ralf.klasing}@labri.fr, Supported in part by the ANR projects ALADDIN and ALPAGE, the INRIA project CEPAGE, and the European projects GRAAL and DYNAMO.

‡ Ochanomizu University, 2-1-1 Otsuka, Bunkyo-ku, Tokyo 112-8610, Japan. E-mail: Jesper.Jansson@ocha.ac.jp. Funded by the Special Coordination Funds for Promoting Science and Technology.

§ Department of Computer Science, Durham University, South Road, Durham, DH1 3LE, UK. E-mail: i.m.lignos@durham.ac.uk.

¶ Principles of Informatics Research Division, National Institute of Informatics. 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan. E-mail: sada@nii.ac.jp.

‖ Department of Computer Science, National University of Singapore, 3 Science Drive 2, 117543 Singapore. E-mail: ksung@comp.nus.edu.sg.

1 Introduction

Efficient search in unknown or unmapped environments is one of the fundamental problems in algorithmics. Its applications range from robot navigation in hazardous environments to rigorous exploration (and, e.g., indexing) of data available on the Internet. Due to a strong need to design simple and cost effective agents as well as to design exploration algorithms that are suitable for rigorous mathematical analysis, it is of practical importance to limit the local memory of agents.

We consider the task of graph exploration by a mobile entity equipped with small (constant number of bits) memory. The mobile entity may be, e.g., an autonomous piece of software navigating through an underlying graph of connections of a computer network. The mobile entity is expected to visit all nodes in the graph in a periodic manner. For the sake of simplicity, we call the mobile entity an *agent* and model it as a finite state automaton. The task of periodic traversal of all nodes of a network is particularly useful in network maintenance, where the status of every node has to be checked regularly.

We consider here undirected graphs that are anonymous, i.e., the nodes in the graph are neither labelled nor colored. To enable the agent to distinguish the different edges incident to a node, edges at a node v are assigned *port numbers* in $\{1, \ldots, d_v\}$ in a one-to-one manner, where d_v is the degree of node v.

We model agents as *Mealy automata*. The Mealy automaton has a finite number of states and a transition function f governing the actions of the agent. If the automaton enters a node v of degree d_v through port i in state s, it switches to state s' and exits the node through port i', where $(s', i') = f(s, i, d_v)$. The memory size of an agent is related to its number of states; more precisely it equals the number of bits needed to encode these states. For example, an oblivious agent has a single state, or, equivalently, zero bits of *persistent* memory. Note that in this model the size of the agent memory represents the amount of information that the agent can remember *while moving between nodes in the graph*. This does not restrict computations made on a node and thus the transition function can be any deterministic function. Additional memory needed for computations can be seen as provided temporarily by the hosting node. Nevertheless, our agent algorithms perform very simple tests and operations on the non-constant inputs i and d, namely equality tests and incrementations.

Periodic graph exploration requires that the agent has to visit every node infinitely many times in a periodic manner. In this paper, we are interested in minimising the length of the exploration period. In other words, we want to minimise the maximum number of edge traversals performed by the agent between two consecutive visits of a generic node, while the agent enters this node in the same state through the same port.

Cohen et al. [4] showed that putting two bits of advice at each node allows to explore all graphs by an agent with constant memory, by a periodic traversal of length $O(m)$, where m is the number of edges. In the general adversarial setting (where the adversary can set the port numbers in a misleading order), the exploration problem is unsolvable, even restricted to cubic planar graphs [5]. On

the other hand, even if nodes are not marked in any way but if port numbers are carefully assigned (still satisfying the condition that at each node v, port numbers from 1 to d_v are used), then a simple agent, even oblivious, can perform periodic graph exploration within period of length $O(n)$. Using appropriate assignment of the local port numbers, the best known period achieved by an oblivious agent is $10n$ [2] whereas the best known period achieved by an agent with constant memory is $3.75n$ [3].

In this paper, we improve both these bounds. Due to space limitations, the missing proofs can be found in the full version [6].

1.1 Related Work

Graph exploration by robots has recently attracted growing attention. The unknown environment in which the robots operate is often modelled as a graph, assuming that the robots may only move along its edges. The graph setting is available in two different forms.

In [7, 8, 9, 10, 11], the robot explores strongly connected directed graphs and it can move only in one pre-specified direction along each edge. In [12, 13, 4, 14, 15, 16, 17], the explored graph is undirected and the agent can traverse edges in both directions. Also, two alternative efficiency measures are adopted in most papers devoted to graph exploration, namely, the *time* of completing the task [7, 12, 8, 9, 13, 10, 14], or the number of *memory bits* (states in the automaton) available to the agent.

In this paper, we are interested in robots characterised by very low memory utilisation. In fact, the robots are allowed to use only a constant number of memory bits. This restriction permits modelling robots as finite state automata. Budach [1] proved that no finite automaton can explore all graphs. Rollik [5] showed later that even a finite team of finite automata cannot explore all planar cubic graphs. This result is improved in [18], where Cook and Rackoff introduce a powerful tool, called the *JAG*, for Jumping Automaton for Graphs. A JAG is a finite team of finite automata that permanently cooperate and that can use *teleportation* to move from their current location to the location of any other automaton. However, even JAGs cannot explore all graphs [18].

2 Preliminaries

2.1 Notation and basic definitions

Let $G = (V, E)$ be a simple, connected, undirected graph. We denote by $\overrightarrow{G}$ the symmetric directed graph obtained from G by replacing each undirected edge $\{u, v\}$ by two directed edges in opposite directions – the directed edge from u to v denoted by (u, v) and the directed edge from v to u denoted by (v, u). For each directed edge (u, v) or (v, u) we say that the undirected edge $\{u, v\} \in G$ is its *underlying* edge. For any node v of a directed graph the *out-degree* of v is the number of directed edges leaving v, the *in-degree* of v is the number of directed

edges incoming to v, and the *cumulative degree* of v is the sum of its out-degree and its in-degree.

Directed cycles constructed by our algorithm traverse some edges in G once and some other edges twice in opposite directions. However, at early stages, our algorithm for oblivious agents is solely interested in whether the edge is unidirectional or bidirectional, indifferently of the direction. To alleviate the presentation (despite some abuse of notation), in this context, an edge that is traversed once when deprived of its direction is called a *single edge*. Similarly, an edge that is traversed twice is called a *two-way edge*, and it is understood to be composed of two single edges (in opposite directions). Hence we extend the notion of single and two-way edges to general directed graphs in which the direction of edges is removed. In particular, we say that two remote nodes s and t are connected by a *two-way path*, if there is a finite sequence of vertices $v_1, v_2, \ldots, v_k$, where each pair v_i and v_{i+1} is connected by a two-way edge, and $s = v_1$ and $t = v_k$. We call a directed graph $\overrightarrow{K}$ *two-way connected* if for any pair of nodes there is a two-way path connecting them. Note that two-way connectivity implies strong connectivity but not the opposite.

2.2 Three-layer partition

The three-layer partition is a new graph decomposition method that we use in constructing periodic tours efficiently in both the oblivious and the constant-memory cases.

For any set of nodes X we call the *neighborhood* of X the set of their neighbors in graph G (excluding nodes in X) and we denote it by $N_G(X)$. One of the main components of the constructions of our technique are *backbone trees* of G, that is, connected cycle-free subgraphs of G. We say that a node v is *saturated* in a backbone tree T of G if all edges incident to v in G are also present in T.

A *three-layer partition* of a graph $G = (V, E)$ is a 4-tuple (X, Y, Z, T_B) such that (1) the three sets X, Y and Z form a partition of V, (2) $Y = N_G(X)$ and $Z = N_G(Y) \setminus X$, (3) T_B is a tree of node-set $X \cup Y$ where all nodes in X are saturated. We call X the *top layer*, Y the *middle layer*, and Z the *bottom layer* of the partition. Any edge of G between two nodes in Y will be called *horizontal*.

During execution of procedure 3L-PARTITION the nodes in V are dynamically partitioned into sets X, Y, Z, P and R with temporary contents, where X is the set of saturated nodes, $Y = N_G(X)$ contains nodes at distance 1 from X, $Z = N_G(Y) \setminus X$ contains nodes at distance 2 from X, $P = N_G(Z) \setminus Y$ contains nodes at distance 3 from X and $R = V \setminus (X \cup Y \cup Z \cup P)$ contains all the remaining nodes in V.

Procedure 3L-PARTITION$(in : G = (V, E); out : X, Y, Z, T_B)$;
(1) $X = Y = Z = P = \emptyset$; $R = V$; $T_B = \emptyset$;
(2) select an arbitrary node $v \in R$;
(3) **loop**
  (a) $X = X \cup \{v\}$; (insert into X newly selected node);

(b) update contents of sets Y, Z, P and R (on the basis of new X);
(c) saturate the newly inserted node v to X (i.e., insert all new edges in T_B);
(d) **if** the new node v in X was selected from P **then** insert in T_B an arbitrary horizontal edge (on middle level) to connect the newly formed star rooted in v with the rest of T_B.
(e) **if** any new node $v \in Y$ can be saturated **then**
select v for saturation;
else-if any new node $v \in Z$ can be saturated **then**
select v for saturation;
else-if P is non-empty **then**
select a new v from P for saturation arbitrarily;
else exit-loop;
end-loop
(4) **output** (X, Y, Z, T_B)

Figure 1 below shows a representative example of the output from the 3L-PARTITION procedure.

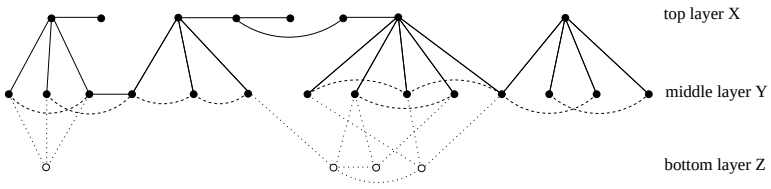

Fig. 1. Three-layer partition. Solid lines and black nodes belong to the backbone tree T_B. Dashed lines represent horizontal edges outside T_B. Dotted lines are incident to nodes from Z.

Lemma 1. *Procedure* 3L-PARTITION *computes a three-layer partition for any connected graph* G.

Lemma 2. *The three-layer partition has the following properties:*
(1) *each node in* Y *has an incident horizontal edge outside of* T_B*;*
(2) *each node in* Z *has at least two neighbors in* Y.

Proof. To prove property (1) assume, by contradiction, that there exists a node $u \in Y$ that has no horizontal edges outside of T_B. Observe that in this case u can be saturated , i.e., u may be moved to X, inserting into T_B all remaining edges incident to u. Indeed, since before u was saturated all such edges lead only to nodes in Z their insertion does not form cycles. Thus property (1) holds. Finally, assume there is a node w in Z with no more than one incident edge leading to level Y. Also in this case we can saturate w since all edges incident to w form a star that shares at most one node with T_B. Thus, no cycle is created, which in turn proves property (2). □

Lemma 3. *For any graph $G = (V, E)$ a three-layer partition may be computed in $O(|E|)$ time.*

2.3 RH-traversability and witness cycles

In this section we discuss the conditions for the oblivious periodic traversals. Given a port number assignment algorithm and an agent algorithm, it is possible, for a given degree d, to permute all port numbers incident to each degree-d node of a graph G according to some fixed permutation σ, and to modify the transition function f of the agent accordingly, so that the agent behaves exactly the same as before in G. The new transition function f' is in this case given by the formula $f' = \sigma \circ f \circ \sigma^{-1}$ and the two agent algorithms are said to be equivalent.

More precisely, two agent algorithms described by their respective transition functions f and f' are *equivalent* if for any $d > 0$ there exists a permutation σ on $\{1, \ldots, d\}$ such that $f' = \sigma \circ f \circ \sigma^{-1}$.

The most common algorithm used for oblivious agents is the Right-Hand-on-the-Wall algorithm. This algorithm is specified by the transition function $f : (s, i, d) \mapsto (s, (i \bmod d)+1)$. Differently speaking, if the agent enters a degree-d node v by port number i, it will exit v through port number $(i \bmod d) + 1$.

The following lemma states that any couple consisting of a port number assignment algorithm and an oblivious agent algorithm, and solving the periodic graph exploration problem, can be expressed by using the Right-Hand-on-the-Wall algorithm as the agent algorithm. We will thus focus on this algorithm in all subsequent parts referring to oblivious agents.

Lemma 4. *Any agent algorithm enabling an oblivious agent to explore all graphs (even all stars) is equivalent to the Right-Hand-on-the-Wall algorithm.*

Graph traversal according to the Right-Hand-on-the-Wall algorithm has been called *right-hand traversals* or shortly *RH-traversals*, see [2]. Similarly, cyclic paths formed in the graph according to the right-hand rule are called *RH-cycles*. The aim of our first oblivious-case algorithm is to find a short RH-traversal of the graph, i.e., to find a cycle $\overrightarrow{C}$ in $\overrightarrow{G}$ containing all nodes of $\overrightarrow{G}$ and satisfying the right-hand rule: If $e_1 = (u, v)$ and $e_2 = (v, w)$ are two successive edges of $\overrightarrow{C}$ then e_2 is the successor of e_1 in the port numbering of v. We call such a cycle a *witness cycle* for G, and the corresponding port numbering a *witness port numbering*.

Given graph $\overrightarrow{G}$ we first design $\overrightarrow{H}$, a spanning subgraph of $\overrightarrow{G}$ that contains all edges of a short witness cycle $\overrightarrow{C}$ of $\overrightarrow{G}$. Then we look for the port numbering of each node in $\overrightarrow{H}$ to obtain $\overrightarrow{C}$. The characterisation of such a graph $\overrightarrow{H}$ is not trivial, however it is easy to characterise graphs which are unions of RH-cycles.

Definition 5. *A node $v \in \overrightarrow{G}$ is* RH-traversable *in $\overrightarrow{H}$ if there exists a port numbering π_v such that, for each edge $(u, v) \in \overrightarrow{H}$ incoming to v via an underlying edge e there exists an outgoing edge $(v, w) \in \overrightarrow{H}$ leaving v via the underlying edge e', such that e' is the successor of e in the port numbering of v.*

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

Let $\overrightarrow{H}$ be a spanning subgraph of $\overrightarrow{G}$. For each node v, denote by b_v, i_v and o_v the number of two-way edges incident to v used in $\overrightarrow{H}$, only incoming and only outgoing edges, respectively. The following lemma characterises the nodes of a graph being a union of RH-cycles.

Lemma 6. *A node v is RH-traversable if and only if $b_v = d_v$ or $i_v = o_v > 0$.*

Proof. ($\Rightarrow$) The definition of RH-traversability implies $i_v = o_v$.
($\Leftarrow$) If $b_v = d_v$, i.e., all edges incident to v are used in two directions, any ordering of the edges is acceptable. Otherwise ($b_v \neq d_v$,) choose a port numbering in which outgoing edges that contribute to two-way edges are arranged in one block followed by an outgoing edge. All remaining directed edges are placed in a separate block, in which edges alternate directions and the last (incoming) edge precedes the block of all two-way edges. □

We easily obtain the following

Corollary 7. *A spanning subgraph $\overrightarrow{H}$ of $\overrightarrow{G}$ is a union of RH-cycles if and only if each node v has an even number of single edges incident to v in $\overrightarrow{H}$, and, in case no single edge is incident to v in $\overrightarrow{H}$, all two-way edges incident to v in $\overrightarrow{G}$ must be also present in $\overrightarrow{H}$.*

In the rest of this section we introduce several operations on cycles, and the conditions under which these operations will result in a witness cycle.

Consider a subgraph $\overrightarrow{H}$ of G that has only RH-traversable nodes. Observe that any port numbering implies a partitioning of $\overrightarrow{H}$ into a set of RH-cycles. Take any ordering γ of this set of cycles. We define two rules which transform one set of cycles to another. The first rule, *Merge3*, takes as an input three cycles incident to a node and merge them to form a single one. The second rule, *EatSmall*, breaks a non-simple cycle into two sub-cycles and transfers one of them to another cycle.

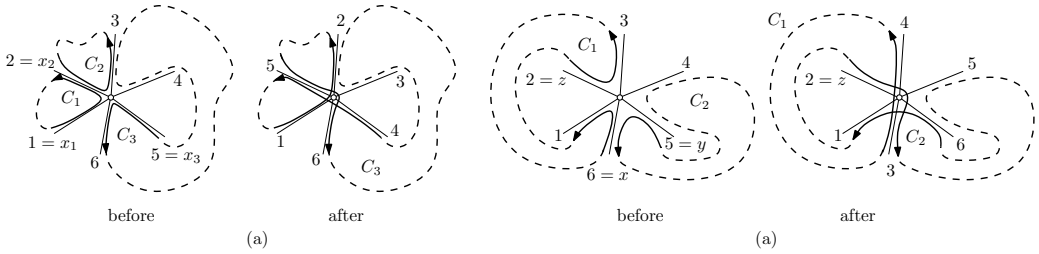

Fig. 2. (a) Applying rule *Merge3*; (b) applying rule *EatSmall*.

1. **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 the underlying edges at v containing incoming edges for cycles C_1, C_2 and C_3, respectively (x_1, x_2 and x_3 can be a single edge or a two-way edge in $\overrightarrow{H}$). Suppose w.l.o.g., that x_2 is between x_1 and x_3 in cyclic port numbering of v. The port numbering which makes the successor of x_2 become the successor of x_1, the successor of x_3 become the successor of x_2 and the successor of x_1 become the successor of x_3 and keeps the relative order of the remaining edges the same (see Figure 2(a)) connects the cycles C_1, C_2 and C_3 into a single cycle C_3, while remaining a witness port numbering for v (due to the original port numbering).
2. **Rule EatSmall:** Let C_1 be the smallest cycle in ordering γ such that
   - there is a node v that appears in C_1 at least twice
   - there is also another cycle C_2 incident to v
   - $\gamma(C_1) < \gamma(C_2)$

   Let x and y be underlying edges at v containing incoming edges for C_1 and C_2, 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 the 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 other edges remains unchanged – see Figure 2(b).

Lemma 8. *Let $\overrightarrow{K}$ be a two-way connected spanning subgraph of G with all nodes RH-traversable in $\overrightarrow{K}$. Consider the set of RH-cycles generated by some witness port numbering of its nodes, with C^* being the largest cycle according to some ordering γ. If neither* Merge3 *nor* EatSmall *can be applied to the nodes of C^* then C^* is a witness cycle.*

Proof. Suppose, by contradiction, that C^* does not span all the nodes in G. Let V' be the set of nodes of G not traversed by C^*. Since $\overrightarrow{K}$ is two-way connected there exist two nodes $u, v \in G$, such that v belongs to C^* and $u \in V'$, and the directed edges (u, v) and (v, u) belong to $\overrightarrow{K}$. Edges (u, v) and (v, u) cannot belong to different cycles of $\overrightarrow{K}$ because *Merge3* would be applicable. Hence (u, v) and (v, u) must both belong to the same cycle C'. However (u, v) and (v, u) cannot be consecutive edges of C' because this would imply $d_v = 1$ which is not the case, since v also belongs to C^*. Hence C' must visit v at least twice. However, since C^* is the largest cycle we have $\gamma(C') < \gamma(C^*)$ and the conditions of applicability of rule *EatSmall* are satisfied with $C_1 = C'$ and $C_2 = C^*$. This is the contradiction proving the claim of the lemma. □

3 Oblivious periodic traversal

In this section we describe the algorithm that constructs a short witness cycle for graph G. This witness cycle will allow an oblivious agent (i.e., one with

no persistent memory) to perform the periodic traversal of G. According to Lemma 8 it is sufficient to construct a spanning subgraph $\overrightarrow{K}$ of G which is two-way connected, such that, each node of G is RH-traversable in $\overrightarrow{K}$. We will present first a restricted case of a *terse set of RH-cycles*, when it is possible to construct a spanning tree of G with no saturated node. In this case we can construct a witness cycle of size $2n$. In the case of arbitrary graphs, we need a more involved argument, which will lead to a witness cycle of size $4\frac{1}{3}n$. We conclude this section with the presentation of a lower bound of $2.8n$.

3.1 Terse set of RH-cycles

Suppose that we have a graph G, which has a spanning tree T with no saturated node. This happens for large and non-trivial classes of graphs, including two-connected graphs, graphs admitting two disjoint spanning trees, and many others. For those graphs we present an algorithm that finds a shorter witness cycle than one that we can find for arbitrary graphs. The idea of the algorithm is to first construct a spanning subgraph of G, $\overrightarrow{K}$ of size $2n$, which contains only RH-traversable nodes (cf. algorithm TerseCycles). Then we apply a port numbering which partitions $\overrightarrow{K}$ into a set of RH-cycles that can then be merged into a single witness cycle (cf. Corollary 10).

Algorithm TerseCycles:

1: Find T – a spanning subgraph of G with no saturated nodes;
2: $\overrightarrow{K} \leftarrow T$; {each edge in T is a two-way edge in $\overrightarrow{K}$}
3: For each node $v \in \overrightarrow{K}$ add to $\overrightarrow{K}$ a single edge from $G \setminus T$; {the single edges form a collection of stars S}
4: Restore-Parity($\overrightarrow{K}, T, root(T)$);

Procedure Restore-Parity has to assure that the number of single edges incident to each node is even. The procedure visits each node v of the tree T in the bottom-up manner and counts all single edges incident to v. If this number is odd, the two-way edge leading to the parent is reduced to a single edge (with the direction to be specified later). The procedure terminates when the parity of all children of the root in the spanning tree is restored. Note also that the cumulative degree of the root must be even since the cumulative degree of all nodes in S is even. Note also that no decision about the direction of single edges is made yet.

Procedure RestoreParity(directed graph $\overrightarrow{K}$, tree T, node v): integer;

1: $P_v =$ (number of single edges in $\overrightarrow{K} \setminus T$) (mod 2);
2: **if** v is not a leaf in T **then**
3: **for** each node $c_v \in T$ being a child of v **do**
4: $P_v \leftarrow (P_v + RestoreParity(\overrightarrow{K}, T, c_v))$ (mod 2);
5: **end for**
6: **end if**

7: **if** $P_v = 1$ **then**
8: reduce the two-way edge $(P, parent(P))$ to single;
9: **end if**
10: **return** P_v;

Lemma 9. *After the completion of procedure* TerseCycles *every node of* $\overrightarrow{K}$ *is RH-traversable.*

Proof. Every node is either saturated or it has at least two single edges incident to it. □

Corollary 10. *For any graph* G *admitting a spanning tree* T*, such that none of the nodes is saturated (i.e.,* $G \setminus T$ *spans all nodes of* G*) it is possible to construct a witness cycle of length at most* $2n$*.*

Corollary 10 gives small witness cycles for a large class of graphs. It should be noted for 3-regular graphs, finding a spanning tree having no saturated nodes corresponds to finding a Hamiltonian path, a problem known to be NP-hard even in this restricted setting [19].

3.2 Construction of witness cycles in arbitrary graphs

The construction of witness cycles is based on the following approach. First select a spanning tree T of graph G composed of two-way edges. Let G_i, for $i = 1, 2, \ldots, k$ be the connected components of $G \setminus T$, having, respectively, n_i nodes. For each such component we apply procedure 3L-Partition, obtaining three sets X_i, Y_i and Z_i and a backbone tree T_i. We then add single edges incident to the nodes of sets Y_i and Z_i, and we apply the procedure RestoreParity to each component G_i. We do this in such a way that the total number of edges in G_i is smaller than $2\frac{1}{3}n$. For the union of graphs $T \cup G_1 \cup G_2 \cup \cdots \cup G_k$ we take a port numbering that generates a set of cycles. The port numbering and orientation of edges in the union of graphs is obtained as follows. First we remove temporarily all two-way edges from the union. The remaining set of single edges is partitioned into a collection of simple cycles, where edges in each cycle have a consistent orientation. Further we reinstate all two-way edges in the union, such that each two-way edge is now represented as two arcs with the opposite direction. Finally we provide port numbers at each node of the union, such that it is consistent with the RH-traversability condition, see Lemma 6. We apply rules *Merge3* and *EatSmall* to this set of cycles until neither rule can be applied. The set of cycles obtained will contain a witness cycle, using Lemma 8.

Algorithm FindWitnessCycle;
1: Find a spanning tree T of graph G {two-way edges}
2: **for** each connected component G_i of $G \setminus T$ **do**
3: 3L-Partition$(G_i, X_i, Y_i, Z_i, T_i)$;
4: Form set P_i by selecting for each node in Z_i two edges leading to Y_i; {single edges};

5: Form a set of independent stars S_i spanning all nodes in Y_i that are not incident to P_i; {single edges};
6: RestoreParity($G_i \cup P_i \cup S_i, T_i, root(T_i)$);
7: **end for**
8: $\overrightarrow{K} \leftarrow T \cup G_1 \cup G_2 \cup \cdots \cup G_k$;
9: Take any port numbering and produce a set $\mathfrak{C}$ of RH-cycles induced by it;
10: Apply repeatedly *Merge3* or, if not possible, *EatSmall* to $\mathfrak{C}$ until neither rule can be applied;
11: **return** the witness cycle of $\mathfrak{C}$;

Theorem 11. *For any n-node graph algorithm* FindWitnessCycle *returns a witness cycle of size at most $4\frac{1}{3}n - 4$.*

Theorem 12. *The algorithm* FindWitnessCycle *terminates in $O(|E|)$ time.*

3.3 Lower Bound

We have shown in the previous section that for any n-node graph we can construct a witness cycle of length at most $4\frac{1}{3}n - 4$. In this section we complement this result with the lower bound $2.8n$:

Theorem 13. *For any non-negative integers n, k and l such that, $n = 5k + l$ and $l < 5$, there exists an n-node graph for which any witness cycle is of length $14k + 2l$.*

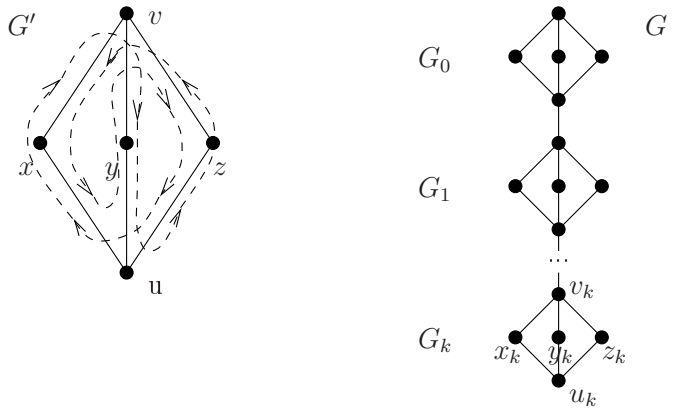

Fig. 3. The lower bound based on diamond graphs.

Proof. Consider first a single *diamond graph* G', see the left part of Figure 3. Without loss of generality, we can assume that we start the traversal through (v, x). Consider the successor of (x, u). Also, without loss of generality, we can

take (u, y) as the successor. Now there is only one feasible successor of (y, v) and that is (v, z). All other edges violate either RH-traversability ((v, y)) or leave z unvisited. Similarly, the only possible successor of (z, u) is (u, x) ((u, y) has already been traversed with a different predecessor, and (u, z) violates RH-traversability), of (x, v) is (v, y) and of (y, u) is (u, z). Therefore, each edge of G' must be used in both directions.

Consider now a chain of diamond graphs from the right side of Figure 3, starting the graph traversal at node v_0. From the fact that each edge in the witness cycle is traversed at most twice (one time in each direction) it follows that when returning from v_i to u_{i-1}, all nodes in G_i (as well as in all G_j, for $j > i$) must have been visited. Note that from RH-traversability it follows that the successor of (u_{i-1}, v_i) cannot be the same (in reverse direction) as the predecessor of (v_i, u_{i-1}), and similarly the successor of (v_i, u_{i-1}) cannot be the same as the predecessor of (u_{i-1}, v_i). In turn this means that the analogous arguments (as used in G') apply also to each G_i, therefore all edges of G must be traversed in both directions.

The theorem now follows directly for $n = 5k$. If n is not a multiple of 5, an extra path of l nodes can be added to u_k to satisfy the claim of the theorem. □

4 Periodic traversal with constant memory

In this section we focus on the construction of a tour in arbitrary undirected graphs to be traversed by an agent equipped with a constant memory. The use of the constant amount of memory allows the agent to change its behavior between a small number of (internal) states for its operation, i.e., the agent has a deterministic transition function and can change from one state from another according to pre-defined rules. As in the case of oblivious agents, we do not impose restrictions on the amount of *local* memory it might have available for use at any vertex, but this local memory is temporary and is lost when an agent leaves the vertex. The main idea of the periodic graph traversal mechanism proposed in [20], and further developed in [3], is to visit all nodes in the graph while traversing along an *Euler tour* of a (particularly chosen) spanning tree (together with a few additional, specially chosen, edges). Due to space constraints, we refer the reader to [3] for more background and details on the mechanism the agent uses to perform the exploration. In what follows, we concentrate on the new construction of the spanning tree (with additional edges) that the agent uses for its exploration.

Recall that the nodes of the input graph can be partitioned into three sets X, Y and Z where all nodes in X and Y are spanned by a backbone tree, see Section 2.2. The spanning tree T is obtained from the backbone tree by connecting every node in Z to one of its neighbors in Y. Recall also that every node $v \in X$ is *saturated*, i.e., all edges incident to v in G belong also to the spanning tree. Every node in Y that lies on a path in T between two nodes in X is called a *bonding node*. The remaining nodes in Y are called *local*.

Initial port labeling When the spanning tree T is formed, we pick one of its leaves as the root r where the two ports located on the tree edge incident to r are set to 1. Initially, for any node v the port leading to the parent is set to 1 and ports leading to the i children of v are set to $2, \ldots, i+1$,, such that the subtree of v rooted in child j is at least as large as the subtree rooted in child j', for all $2 \leq j < j' \leq i+1$. All other ports are set arbitrarily using distinct values from the range $i+2, \ldots, d_v$, where d_v is the degree of v. Later, we modify the allocation of ports at certain leaves of the spanning tree located in Z. In particular we change labels at all children having no other leaf-siblings in T of bonding nodes (see, e.g., node w_1 in Figure 4), as well as at single children of local nodes, but only if the local node is the last child of a node in X that has children on its own (see, e.g., node w_2 in Figure 4).

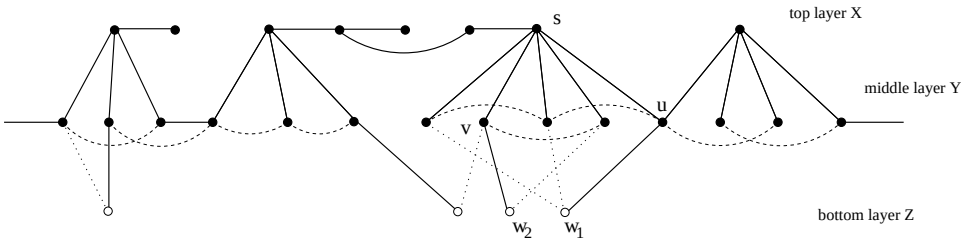

Fig. 4. Fragment of the spanning tree with the root located to the right of w_1 and w_2.

Port swap operation Recall that every leaf w located at the level Z has also an incident edge e outside of T that leads to some node v in Y (property 2 of the three-layer partition). When we swap port numbers at w, we set to 2 the port on the tree edge leading to the parent of w. We call such edge a *sham penalty edge* since it now pretends to be a penalty edge while, in fact, it connects w to its parent in the spanning tree T. We also set to 1 the port number on the lower end of e. All other port numbers at w (if there are more incident edges to w) are set arbitrarily. After the port swap operation at w is accomplished we also have to ensure that the edge e will never be examined by the agent, otherwise it would be wrongly interpreted as a legal tree edge, where v would be recognised as the parent of w. In order to avoid this problem we also set ports at v with greater care. Note that v has also an incident horizontal edge e' outside of T (property 1 of the three-layer partition). Assume that the node v has i children in T. Thus if we set to $i+2$ the port on e' (recall that port 1 leads to the parent of v and ports $2, .., i+1$ lead to its children) the port on e will have value larger than $i+2$ and e will never be accessed by the agent. Finally note that the agent may wake up in the node with a sham penalty edge incident to it. For this reason we introduce an extra state to the finite state automaton $\mathcal{A}$ governing moves of the agent in [3] to form a new automaton $\mathcal{A}^+$. While being in the wake up state the agent moves across the edge accessible via port 1 in order to start regular

performance (specified in [3]) in a node that is not incident to the lower end of a sham penalty edge.

Lemma 14. *The new port labeling provides a mechanism to visit all nodes in the graph in a periodic manner by the agent equipped with a finite state automaton* $\mathcal{A}^+$.

Theorem 15. *For any undirected graph* G *with* n *nodes, it is possible to compute a port labeling such that an agent equipped with a finite state automaton* $\mathcal{A}^+$ *can visit all nodes in* G *in a periodic manner with a tour length that is no longer than* $3\frac{1}{2}n - 2$.

Note that in the model with implicit labels, one port at each node has to be distinguished in order to break symmetry in a periodic order of ports. This is to take advantage of the extra memory provided to the agent.

5 Conclusion

Further studies on trade-offs between the length of the periodic tour and the memory of a mobile entity are needed. The only known lower bound $2n - 2$ holds independently of the size of the available memory, and it refers to trees. This still leaves a substantial gap in view of our new $3.5n$ upper bound. Another alternative would be to look for as good as possible tour for a given graph, for example, in a form of an approximate solution. Indeed, for an arbitrary graph, finding the shortest tour may correspond to discovering a Hamiltonian cycle in the graph, which is NP-hard.

Acknowledgements. Many thanks go to Adrian Kosowski, Rastislav Kralovic, and Alfredo Navarra for a number of valuable discussions on the main themes of this work.

References

[1] Budach, L.: Automata and labyrinths. *Mathematische Nachrichten* (1978) 195–282.

[2] Dobrev, S., Jansson, J., Sadakane, K., Sung, W.K.: Finding short right-hand-on-the-wall walks in graphs. In: *Proc. 12th Colloquium on Structural Information and Communication Complexity (SIROCCO).* Volume LNCS 3499. (2005) 127 – 139.

[3] Gąsieniec, L., Klasing, R., Martin, R., Navarra, A., Zhang, X.: Fast periodic graph exploration with constant memory. *J. Computer and System Science* **74**(5) (2008) 808 – 822.

[4] Cohen, R., Fraigniaud, P., Ilcinkas, D., Korman, A., Peleg, D.: Label-guided graph exploration by a finite automaton. *ACM Transactions on Algorithms* **4**(4) (2008) 331–344

[5] Rollik, H.: Automaten in planaren graphen. *Acta Informatica* **13** (1980) 287–298.

[6] Czyzowicz, J., Dobrev, S., Gasieniec, L., Ilcinkas, D., Jansson, J., Klasing, R., Lignos, I., Martin, R., Sadakane, K., Sung, W.K.: More efficient periodic traversal in anonymous undirected graphs (full version). `http://arxiv.org/abs/0905.1737`

[7] Albers, S., Henzinger, M.R.: Exploring unknown environments. *SIAM J. Computing* **29** (2000) 1164–1188.

[8] Bender, M., Fernandez, A., Ron, D., Sahai, A., Vadhan, S.: The power of a pebble: Exploring and mapping directed graphs. *Information and Computation* **176**(1) (2002) 1–21.

[9] Bender, M., Slonim, D.K.: The power of team exploration: two robots can learn unlabeled directed graphs. In: *Proc. 35th Annual Symposium on Foundations of Computer Science (FOCS)*. (1994) 75–85.

[10] Deng, X., Papadimitriou, C.H.: Exploring an unknown graph. *J. Graph Theory* **32**(3) (1999) 265–297.

[11] Fleischer, R., Trippen, G.: Exploring an unknown graph efficiently. In: *Proc. 13th Annual European Symposium on Algorithms (ESA)*. (2005) 11–22.

[12] Awerbuch, B., Betke, M., Rivest, R., Singh, M.: Piecemeal graph exploration by a mobile robot. *Information and Computation* **152**(2) (1999) 155–172.

[13] Betke, M., Rivest, R., Singh, M.: Piecemeal learning of an unknown environment. *Machine Learning* **18**(2-3) (1995) 231–254.

[14] Duncan, C., Kobourov, S., Kumar, V.: Optimal constrained graph exploration. *ACM Transaction on Algorithms* **2(3)** (2006) 380–402.

[15] Fraigniaud, P., Ilcinkas, D., Peer, G., Pelc, A., Peleg, D.: Graph exploration by a finite automaton. *Theoretical Computer Science* **345**(2-3) (2005) 331–344.

[16] Fraigniaud, P., Ilcinkas, D., Rajsbaum, S., Tixeuil, S.: The reduced automata technique for graph exploration space lower bounds. In: *Essays in Memory of Shimon Even*. Volume 3895 of LNCS. (2006) 1–26.

[17] Panaite, P., Pelc, A.: Exploring unknown undirected graphs. *J. Algorithms* **33** (1999) 281–295.

[18] Cook, S.A., Rackoff, C.: Space lower bounds for maze threadability on restricted machines. *SIAM J. Computing* **9**(3) (1980) 636–652.

[19] Garey, M., Johnson, D., Tarjan, R.: The planar Hamiltonian circuit problem is NP-complete. *SIAM J. Computing* **5(4)** (1976) 704–714.

[20] Ilcinkas, D.: Setting port numbers for fast graph exploration. *Theoretical Computer Science* **401** (2008) 236 – 242.