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, Wing-Kin Sung

More efficient periodic traversal in anonymous undirected graphs☆

Jurek Czyzowicz[a], Stefan Dobrev[b], Leszek Gąsieniec[c,1], David Ilcinkas[d,2], Jesper Jansson[e,3], Ralf Klasing[d,2], Ioannis Lignos[f], Russell Martin[c,4], Kunihiko Sadakane[g], Wing-Kin Sung[h]

[a] *Département d'Informatique, Université du Québec en Outaouais, Gatineau, Québec J8X 3X7, Canada.*
[b] *Institute of Mathematics, Slovak Academy of Sciences, Dubravska 9, P.O.Box 56, 840 00, Bratislava, Slovak Republic.*
[c] *Department of Computer Science, University of Liverpool, Ashton Street, Liverpool, L69 3BX, United Kingdom.*
[d] *LaBRI, CNRS and University of Bordeaux, 351 cours de la Liberation, F-33405 Talence cedex, France.*
[e] *Ochanomizu University, 2-1-1 Otsuka, Bunkyo-ku, Tokyo 112-8610, Japan.*
[f] *Department of Computer Science, Durham University, South Road, Durham, DH1 3LE, United Kingdom.*
[g] *Principles of Informatics Research Division, National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan.*
[h] *Department of Computer Science, National University of Singapore, 3 Science Drive 2, 117543 Singapore.*

Abstract

We consider the problem of *periodic graph exploration* in which a mobile entity with constant memory, *an agent*, has to visit all n nodes of an input simple, connected, undirected graph in a periodic manner. Graphs are assumed to be anonymous, that is, nodes are unlabeled. While visiting a node, the agent may distinguish between the 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 algorithms for assigning the port numbers together with traversal algorithms for agents using these port numbers to obtain short traversal periods.

Periodic graph exploration is unsolvable if the port numbers are set arbitrarily, see [1]. However, surprisingly small periods can be achieved by carefully assigning the port numbers. Dobrev *et al.* [4] described an algorithm for assigning port numbers and an oblivious agent (i.e., an agent with no memory) using it, such that the agent explores any graph with n nodes within the period $10n$. When the agent has access to a constant number of memory bits, the optimal length of the period was proved in [7] to be no more than $3.75n - 2$ (using a different assignment of the port numbers and a different traversal algorithm). In this paper, we improve both these bounds. More precisely, we show how to achieve a period length of at most $(4 + \frac{1}{3})n - 4$ for oblivious agents and a period length of at most $3.5n - 2$ for agents with constant memory. To obtain our results, we introduce a new, fast graph decomposition technique called a *three-layer partition* that may also be useful for solving other graph problems in the future. Finally, we present the first non-trivial lower bound, $2.8n - 2$, on the period length for the oblivious case.

Keywords: algorithms and data structures, graph exploration, periodic graph traversal, oblivious agent, constant-memory agent, three-layer partition

1. Introduction

Efficient search in unknown or unmapped environments is a fundamental problem in algorithmics. Its applications range from robot navigation in hazardous environments to rigorous exploration (and 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 suitable for rigorous mathematical analysis, it is of practical importance to limit the memory of agents.

In this paper, we consider the task of graph exploration by a mobile entity equipped with a constant number of bits memory. The mobile entity may be, e.g., an autonomous piece of software navigating through a graph that represents the nodes and connections of a computer network. For the sake of simplicity, we call the mobile entity an *agent* and model it as a finite state automaton. We require that the agent visits all nodes in an input graph infinitely many times, in a periodic manner. 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.

To assist the agent, we assign local *port numbers* to the edges at each node as a preprocessing step. Then, while traversing the graph, the agent is allowed to use the local port numbers to ensure that all nodes are visited. Our goal is to minimize the length of the traversal period; in other words, we would like to assign the port numbers so that the maximum number of edge traversals performed by the agent between two consecutive visits to the same node and entering through the same port is minimized. From here on, we assume that the input graph is simple, connected, and undirected. We also assume it to be anonymous, i.e., all nodes are unlabeled.

1.1. Problem definition

Let $G = (V, E)$ be a simple, connected, undirected graph. For any node $v \in V$, the *degree of* v is the number of neighbors of v and is denoted by d_v. To enable an agent to distinguish between the different edges incident to a node, the edges at every node v will be assigned local *port numbers* from $\{1, 2, \ldots, d_v\}$ bijectively. (Every edge will therefore be assigned *two* port numbers; one at each of its two endpoints.) See Figure 1.

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 (cf. [10]). 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; to be precise, it equals the number of bits needed to encode these states. Note that in this model, the size of the agent's memory represents the amount of information that the agent can remember while moving. This does not restrict computations made on a node and thus the transition function can be any deterministic function; any additional memory needed for computations can be seen as provided temporarily by the hosting node. Nevertheless, our traversal algorithms only perform very simple tests and operations on the non-constant inputs i and d, namely equality tests and incrementations.

[☆] A preliminary version of this article appeared in *Proceedings of the 16th International Colloquium on Structural Information and Communication Complexity* (SIROCCO 2009), vol. 5869 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 174–188, 2009.

Email addresses: jurek@uqo.ca (Jurek Czyzowicz), stefan@ifi.savba.sk (Stefan Dobrev), L.A.Gasieniec@liverpool.ac.uk (Leszek Gąsieniec), david.ilcinkaslabri.fr (David Ilcinkas), Jesper.Jansson@ocha.ac.jp (Jesper Jansson), ralf.klasing@labri.fr (Ralf Klasing), yannis.lignos@durham.ac.uk (Ioannis Lignos), Russell.Martin@liverpool.ac.uk (Russell Martin), sada@nii.ac.jp (Kunihiko Sadakane), ksung@comp.nus.edu.sg (Wing-Kin Sung)

[1] Partially funded by the Royal Society International Joint Project, IJP - 2007/R1.

[2] Supported in part by the ANR projects ALADDIN and ALPAGE, the INRIA project CEPAGE, and the European projects GRAAL and DYNAMO.

[3] Funded by the Special Coordination Funds for Promoting Science and Technology, Japan.

[4] Partially funded by the Nuffield Foundation grant NAL/32566, "The structure and efficient utilization of the Internet and other distributed systems".

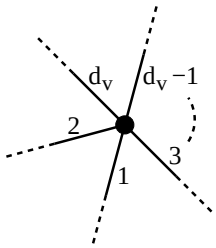

Figure 1: In a port number assignment, the d_v edges incident to node v are locally given the numbers $1, 2, 3, \ldots, d_v$ in some order.

The problem considered in this paper is to design a port number assignment algorithm and a traversal algorithm that enable the agent to periodically visit all nodes in an input graph. The efficiency measure we use to compare solutions is the resulting *period length*, which is the maximum number of edge traversals between two consecutive visits to a node entering through the same port, taken over all nodes. The period length is expressed in terms of n, the number of nodes in the input graph, and our main objective is to find algorithms achieving a small period length for any input graph. We focus on two cases: the *oblivious agent*, having a single state (or equivalently, zero memory bits), and the *constant-memory agent*, equipped with a constant number of bits independent of the size of the input graph. By the above discussion, oblivious agents can be regarded as having access to any amount of temporary memory while stationed at a node but losing all this memory when exiting the node.

1.2. Previous results

Budach [1] proved that no finite automaton can explore all graphs. Rollik [12] later proved that an agent needs $\Omega(\log n)$ memory bits to explore any graph with n nodes, even if restricted to cubic planar graphs. (This lower bound was in fact recently proved to be optimal by Reingold in his breakthrough paper [11].) Therefore, the basic periodic graph exploration problem is unsolvable for agents with small memory. Providing the agent with a pebble to mark nodes does not help much as the asymptotic size of the memory needed remains $\Omega(\log n)$ bits [5]. Furthermore, even a highly-coordinated multi-agent team capable of (restricted) teleportation cannot explore all graphs using only constant memory [3]. Nevertheless, placing some extra information in the graph can help a lot. Cohen *et al.* [2] demonstrated that putting two bits of advice at each node allows any graph to be explored by an agent with constant memory by a periodic traversal of length $O(m)$, where m is the number of edges.

The impossibility results mentioned above all use the ability of an adversary to assign local port numbers in a misleading order. On the other hand, if port numbers are carefully assigned beforehand (still under the condition that at each node v, port numbers from 1 to d_v are employed) then a simple agent, even an oblivious one, can perform periodic graph exploration within a period of length $O(n)$ [4]. More precisely, Dobrev *et al.* [4] showed that there exists an algorithm for setting the port numbers in such a way that an oblivious agent using the so-called Right-Hand-on-the-Wall algorithm as its traversal algorithm will traverse any graph with n nodes within the period $10n$. Significantly, this holds even if the nodes themselves are not marked in any way while the agent traverses the graph. For agents with constant memory, Ilcinkas [8] gave an algorithm achieving an upper bound of $4n - 2$ on the period length, which was subsequently improved to $3.75n - 2$ by Gąsieniec *et al.* [7]. (References [4] and [8] also considered dynamic versions of periodic graph exploration in which the graph may be modified while the agent is traversing it.)

As for corresponding lower bounds on the period length, the star graph with n nodes (i.e., having $n - 1$ edges) yields a trivial lower bound of $2n - 2$ for any type of agent, independent of the amount of available memory, since every edge of the graph must be traversed in both directions. Also note that in case the input graph contains a Hamiltonian cycle, the optimal period length is n: just set the port numbers to direct the agent along the cycle.

1.3. Our new results and organization of the paper

In this paper, we improve the previously best upper and lower bounds on the period length for periodic graph exploration by an oblivious agent [4] and by an agent with constant memory [7, 8] as follows. We present an efficient deterministic algorithm named FindWitnessCycle for assigning port numbers at the nodes of the input graph so that an oblivious agent using an extremely simple traversal algorithm named the Right-Hand-on-the-Wall algorithm (reviewed in Section 2.2) achieves a period length of at most $(4 + \frac{1}{3})n - 4$. Our port number assignment algorithm relies on a new three-layer partition technique for graphs, described in Section 3, permitting an optimal $O(|E|)$-time construction of the port labeling. The details of FindWitnessCycle can be found in Section 4. As a special case, we also consider a class of graphs for which an oblivious agent can obtain a traversal with period length at most $2n$ by using a simpler algorithm named TerseCycles as the port number assignment algorithm. Next, we provide the first non-trivial lower bound, $2.8n-2$, on the period length for oblivious agents in the general case in Section 5. Then, in Section 6, we give an algorithm (also based on the three-layer partition approach) which assigns port numbers so that an agent with constant memory is able to accomplish periodic graph exploration within a period length of at most $3.5n-2$. Finally, Section 7 summarizes our new results and discusses some related open problems.

2. Preliminaries

2.1. Notation and basic definitions

For any undirected graph $G = (V, E)$, we denote by $\overrightarrow{G}$ the symmetric directed graph obtained from G by replacing each undirected edge $\{u, v\} \in E$ by two directed edges in opposite directions: one directed edge from u to v, denoted by (u, v), and one 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 *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 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, regardless 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 *one-way edge*. Similarly, an edge that is traversed twice is called a *two-way edge*, and it is understood to be composed of two one-way edges. Hence, we extend the notion of one-way and two-way edges to general directed graphs in which the direction of edges is removed. In particular, we say that two nodes s and t are connected by a *two-way path*, if there is a finite sequence of nodes $v_1, v_2, \ldots, v_k$, 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 other way around.

2.2. Traversal algorithms for oblivious agents

A simple graph traversal algorithm for oblivious agents is the *Right-Hand-on-the-Wall algorithm* [4]. 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_v node v by port number i, it will exit v through port number $(i \bmod d_v) + 1$. The Right-Hand-on-the-Wall algorithm assumes that the initial starting node can be any node v in G and that the agent entered v from port number d_v; therefore, the traversal will always start with an edge with port number 1. See Figure 2 for an example. For any given graph, there exists at least one assignment of port numbers that allows the Right-Hand-on-the-Wall algorithm to visit all nodes periodically [4].

Graph traversal according to the Right-Hand-on-the-Wall algorithm is called *right-hand traversal* or *RH-traversal* for short.

Given a port number assignment algorithm and a traversal algorithm for the agent, 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

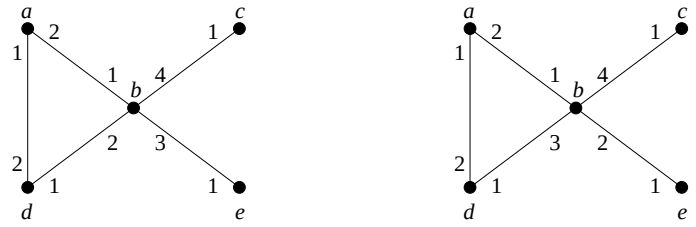

Figure 2: (a) Running the Right-Hand-on-the-Wall algorithm with the given port number assignment and starting at node a will visit all nodes of the graph in the order $\langle a, d, b, e, b, c, b, a, \dots \rangle$, with period length 7. (b) If this port number assignment is used then the Right-Hand-on-the-Wall algorithm will not be able to visit all nodes of the graph.

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 traversal algorithms are said to be equivalent. More precisely, two traversal algorithms described by their respective transition functions f and f' are *equivalent* if for any $d > 0$ there exists a permutation σ on $\{1, \dots, d\}$ such that $f' = \sigma \circ f \circ \sigma^{-1}$. The following lemma states that any pair consisting of a port number assignment algorithm and a traversal algorithm for oblivious agents, and solving the periodic graph exploration problem, can be expressed by using the Right-Hand-on-the-Wall algorithm as the traversal algorithm.

Lemma 1. *Any traversal algorithm enabling an oblivious agent to explore all graphs is equivalent to the Right-Hand-on-the-Wall algorithm.*

Proof. Consider an arbitrary algorithm $\mathcal{A}$ enabling an oblivious agent to periodically explore all graphs. Let f be its transition function. Fix an arbitrary $d > 1$ and let f_d be the function $i \mapsto f(s, i, d)$ from $\{1, \dots, d\}$ to $\{1, \dots, d\}$, where s is the single state of the oblivious agent. Consider the $d + 1$-node star of degree d. For $1 \leq i \leq d$, let v_i be the leaf reachable from the central node u by the edge with port number i.

For the purpose of obtaining a contradiction, first suppose that f_d is not surjective. Let i be a port number without pre-image. If the agent is started by the adversary in node v_j, with $j \neq i$, then the node v_i is never explored. Therefore f_d is surjective, and thus a permutation of the set $\{1, \dots, d\}$. Again for the purpose of contradiction, suppose that f_d can be decomposed into more than one cycle. Let i be a port number outside 1's orbit (i.e., 1 and i are not in the same cycle of the permutation). If the agent is started by the adversary in node v_1, then the node v_i is never explored. Hence, f_d is a cyclic permutation, i.e., it is constructed with a single cycle. Since the equivalence classes of permutations (often called conjugacy classes) correspond exactly to the cycle structures of permutations, the traversal algorithm $\mathcal{A}$ is equivalent to the Right-Hand-on-the-Wall algorithm. □

Because of Lemma 1, we will always assume in the rest of the paper when referring to oblivious agents that the Right-Hand-on-the-Wall algorithm is employed as the traversal algorithm.

2.3. Witness cycles and RH-traversability

Any (possibly non-simple) directed cycle formed when traversing a graph according to the Right-Hand-on-the-Wall algorithm described above for a fixed port number assignment is called an *RH-cycle*. A *witness cycle* for a graph G is an RH-cycle that contains every node of G at least once.

If we are given a witness cycle C for G, it is straightforward to assign port numbers to the nodes in G so that an oblivious agent using the Right-Hand-on-the-Wall algorithm will traverse G according to C. (To ensure that any node can be used as the starting node, at every node v, assign port numbers 1 and d_v to an underlying edge for an edge in C directed out from v and into v, respectively.) Therefore, to obtain a port number assignment algorithm for oblivious agents, we just need to specify how to construct a witness cycle for any input graph. This will be done in Section 4.

One key step in our method in Section 4 is to compute a set of RH-cycles and then merge them into a single witness cycle. Recall that $\overrightarrow{G}$ is the symmetric directed graph obtained from G by replacing each undirected edge by two directed edges. In the rest of this subsection, we characterize when a spanning subgraph of $\overrightarrow{G}$ is a union of RH-cycles.

Definition 1. *Let $\overrightarrow{H}$ be a spanning subgraph of $\overrightarrow{G}$. A node $v \in G$ is* RH-traversable in $\overrightarrow{H}$ *if there exists a port number assignment π for G 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 an underlying edge e' such that e' is the successor of e in π at node v.*

As a special case, if $\overrightarrow{H}$ is a witness cycle for G then every node is RH-traversable in $\overrightarrow{H}$. However, nodes may be RH-traversable in $\overrightarrow{H}$ even if $\overrightarrow{H}$ is not a witness cycle (q.v. Figure 4), and the next lemma (Lemma 2) gives a useful condition for checking RH-traversability in the general case. To state the lemma, we need the following additional notation. Let $\overrightarrow{H}$ be a fixed spanning subgraph of $\overrightarrow{G}$. Any undirected edge $\{u, v\}$ in G is called a *two-way edge for* $\overrightarrow{H}$ if both of the directed edges (u, v) and (v, u) belong to $\overrightarrow{H}$. Otherwise, if (u, v) belongs to $\overrightarrow{H}$ but (v, u) does not, then $\{u, v\}$ is called a *one-way edge to v in* $\overrightarrow{H}$ as well as a *one-way edge from u in* $\overrightarrow{H}$. For each node v in $\overrightarrow{H}$, define:

- b_v = The number of two-way edges for $\overrightarrow{H}$ incident to v.
- i_v = The number of one-way edges to v in $\overrightarrow{H}$.
- o_v = The number of one-way edges from v in $\overrightarrow{H}$.

Thus, $b_v + i_v + o_v$ equals the number of edges in G incident to v that are underlying edges for directed edges belonging to $\overrightarrow{H}$. For an example, see Figure 3.

Lemma 2. *A node v is RH-traversable in $\overrightarrow{H}$ if and only if $b_v = d_v$ or $i_v = o_v > 0$.*

Proof. ($\Rightarrow$) Let π be a port number assignment for G meeting the requirements of Definition 1 and denote the port number assigned by π to any edge e at v by $\pi(e)$. If $b_v = d_v$ then we are done, so consider the case $b_v \neq d_v$. Suppose for the sake of contradiction that there are no one-way edges to v or from v. Let e be any edge that is not two-way and let e' be the two-way edge with the largest possible $\pi(e')$ satisfying $\pi(e') < \pi(e)$. It follows that there must exist some one-way edge f from v with $\pi(e') < \pi(f) < \pi(e)$, which is a contradiction. Therefore, there exists at least one-way edge to v or from v, and thus $i_v > 0$ or $o_v > 0$ by definition. The number of incoming edges equals the number of outgoing edges at v, so $b_v + i_v = b_v + o_v$, i.e., $i_v = o_v$.
($\Leftarrow$) If $b_v = d_v$ then all edges incident to v are used in both directions and any ordering of the edges gives an acceptable port number assignment. Otherwise, $b_v \neq d_v$ and $i_v = o_v > 0$, and we can take the following port number assignment: First, all underlying edges that are two-way edges for $\overrightarrow{H}$ are numbered consecutively, starting from 1, followed by an underlying edge for any one-way edge from v. Next, all other underlying edges for one-way edges are numbered consecutively while alternating between one-way edges to v and one-way edges from v so that the last (incoming) edge gets port number d_v. See Figure 3. Finally, the remaining edges may be numbered arbitrarily by the unused port numbers. □

Lemma 2 immediately yields:

Corollary 1. *A spanning subgraph $\overrightarrow{H}$ of $\overrightarrow{G}$ is a union of RH-cycles if and only if at each node v of G, the number of one-way edges to v in $\overrightarrow{H}$ equals the number of one-way edges from v in $\overrightarrow{H}$, and if this number is zero then all two-way edges for $\overrightarrow{G}$ incident to v must also be present in $\overrightarrow{H}$.*

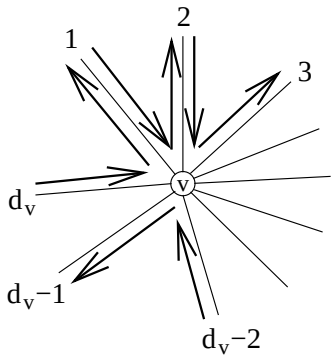

Figure 3: In this example, edges belonging to $\overrightarrow{H}$ are shown as arrows. Node v has $d_v > b_v = 2$, $i_v = 2$, and $o_v = 2$, so by Lemma 2, v is RH-traversable in $\overrightarrow{H}$. The displayed port number assignment corresponds to the construction in the proof of Lemma 2.

2.4. Operations on cycles by modifying port numbers

Consider the graph and the port number assignment in Figure 4. The port numbers induce a set $\mathfrak{C}$ of three RH-cycles. Every node is RH-traversable in the directed graph formed by taking the union of the cycles in $\mathfrak{C}$ according to Lemma 2, but there is no witness cycle in $\mathfrak{C}$. However, if we exchange two port numbers at one of the degree-three nodes, then the three cycles merge into a witness cycle.

In this subsection, we describe two operations on cycles (implemented by modifying the corresponding port number assignments) and the conditions under which these operations will produce a witness cycle. The operations are called *Merge3* and *EatSmall*, and were introduced by Dobrev *et al.* in [4].

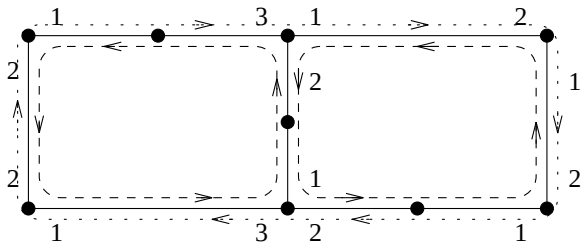

Figure 4: The above port number assignment induces three cycles, indicated by dashed and dotted lines. Note that every node is RH-traversable in the union of the three cycles, but there is no witness cycle among the three cycles.

Let $\overrightarrow{H}$ be a subgraph of G that has only RH-traversable nodes. Observe that any port number assignment partitions $\overrightarrow{H}$ 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 by changing the port number assignment. The first rule, *Merge3*, takes as input three cycles incident to a node and merges them into one cycle. In the case where the node is visited by more than three cycles, the rule is applied to arbitrarily chosen three cycles. The second rule, *EatSmall*, breaks a non-simple cycle into two subcycles and transfers one such subcycle to another cycle.

- **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 one-way edge or a two-way edge in $\overrightarrow{H}$). Assume w.l.o.g. that x_2 is between x_1 and x_3 in the port number assignment at v; see Figure 5. Modify the port number assignment at v as follows: (1) let the successor of x_2 become the new successor of x_1, (2) let the old successor of x_3 become the new successor of x_2, (3) let the old successor of x_1 become the new successor of x_3, and

(4) keep the same relative order of the other edges. It is easy to see that this operation connects the cycles C_1, C_2 and C_3 into a single cycle.

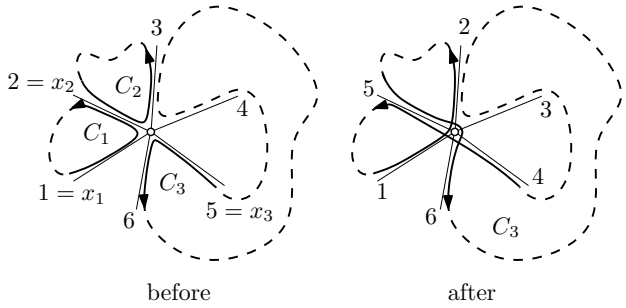

Figure 5: Applying rule *Merge3* will change the port number assignment at the shown node so that the three cycles C_1, C_2, and C_3 are merged into one cycle.

- **Rule EatSmall:**

  Let C_1 be the smallest cycle in the 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. See Figure 6. 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.

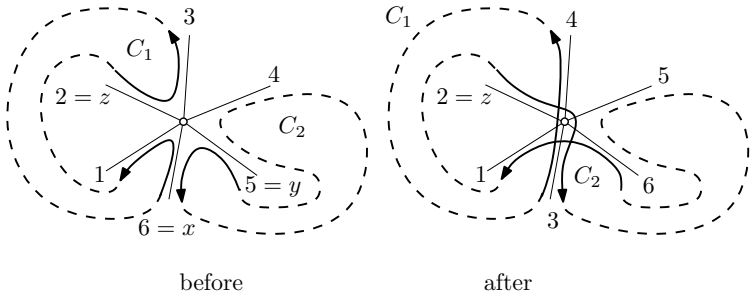

Figure 6: Applying rule *EatSmall* modifies the port number assignment at the shown node so that cycle C_1 becomes shorter and C_2 longer.

The next important lemma implies that a witness cycle can be found by repeatedly applying *Merge3* and *EatSmall*.

Lemma 3. *Let $\overrightarrow{K}$ be a two-way connected spanning subgraph of G such that all nodes in G are RH-traversable in $\overrightarrow{K}$. Consider the set of RH-cycles generated by some 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 a contradiction, proving the claim of the lemma. □

3. Three-layer partition

The three-layer partition is a new, fast graph decomposition method that we shall use to efficiently construct periodic tours, both for oblivious agents in Section 4 and bounded-memory agents in Section 6. It is defined as follows.

For any set X of nodes in a graph G, the *neighborhood of X* (denoted by $N_G(X)$) is the set of neighbors of X in G, excluding nodes belonging to X. For any node v in G and subgraph T of G, we say that v is *saturated in T* if v and all edges incident to v in G are also present in T.

Definition 2. *A* three-layer partition *of a simple, connected, undirected graph $G = (V, E)$ is a 4-tuple (X, Y, Z, T_B) such that:*

- *The three sets X, Y, and Z form a partition of V.*
- *$Y = N_G(X)$ and $Z = N_G(Y) \setminus X$.*
- *T_B is a connected, cycle-free subgraph of G (i.e., a tree) with node set $X \cup Y$ in which all nodes from X are saturated.*

See Figure 7 for an example. 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 is called *horizontal*, and the tree T_B is called a *backbone tree of G*. Note that a backbone tree is not the same thing as a spanning tree (in particular, it does not contain any node from the bottom layer Z); however, backbone trees will help us to find certain useful spanning trees later on.

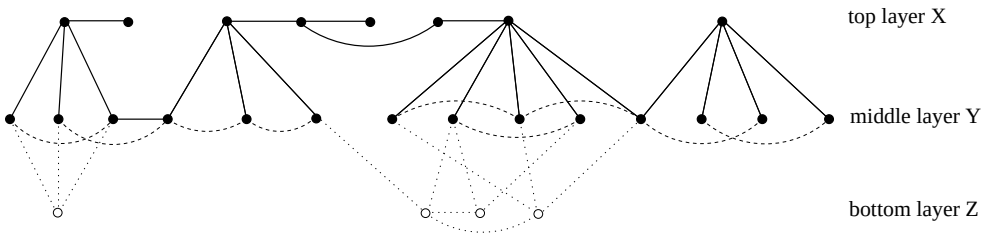

Figure 7: A three-layer partition. Solid lines and black nodes belong to the backbone tree T_B. Dashed lines represent horizontal edges outside T_B. Dotted lines represent edges that are incident to nodes from Z.

We now present a fast algorithm named 3L-Partition for constructing a three-layer partition with backbone tree T_B of any given graph $G = (V, E)$. The pseudocode is given in Figure 8. During execution, the nodes in V are dynamically partitioned into sets X, Y, Z, P, and R with temporary contents, where:

- X is the set of nodes currently saturated in T_B.

- $Y = N_G(X)$ contains all nodes at distance 1 from X.
- $Z = N_G(Y) \setminus X$ contains all nodes at distance 2 from X.
- $P = N_G(Z) \setminus Y$ contains all nodes at distance 3 from X.
- $R = V \setminus (X \cup Y \cup Z \cup P)$ contains all remaining nodes from V.

(Thus, the contents of sets Y, Z, P, and R strictly depend on the current contents of X.) Initially, all nodes belong to R and the backbone tree T_B is empty. Each iteration of the main loop (called a *round*) makes one node v saturated in T_B by moving it to X and inserting the corresponding edges into T_B. The algorithm terminates when no more nodes can be saturated, i.e., can be added to X without inducing a cycle.

Algorithm 3L-PARTITION
Input: A simple, connected, undirected graph $G = (V, E)$.
Output: A three-layer partition (X, Y, Z, T_B) of G.

1: $X = Y = Z = P = \emptyset$; $R = V$; $T_B = \emptyset$.
2: Select an arbitrary node $v \in R$.
3: **loop**
4: $X = X \cup \{v\}$ /* Insert the newly selected node v into X. */
5: Update the sets Y, Z, P, and R on the basis of the new X.
6: Make node v saturated in T_B by inserting every edge incident to v that is not already in T_B.
7: **if** node v was selected from P **then**
8: Take any horizontal edge from the middle layer Y and insert it into T_B to connect the newly formed star rooted in v to the rest of T_B.
9: **end if**
10: /* Select a node v for saturation in the next round. */
11: **if** any node $v \in Y$ can be added to X without inducing a cycle **then**
12: Select v for saturation.
13: **else if** any node $v \in Z$ can be added to X without inducing a cycle **then**
14: Select v for saturation.
15: **else if** P is non-empty **then**
16: Arbitrarily select a new node v from P for saturation.
17: **else**
18: Exit loop. /* No more nodes can be moved to X. */
19: **end if**
20: **end loop**
21: **return** (X, Y, Z, T_B)

Figure 8: Algorithm 3L-PARTITION.

Theorem 1. *Algorithm* 3L-PARTITION *computes a three-layer partition of any simple, connected, undirected graph G.*

Proof. We shall show that the algorithm outputs a three-layer partition of G with a distinguished backbone tree T_B. We use the following invariant: At the end of each round, nodes in X and Y are spanned by a partial backbone tree T_B and a new node v is selected for saturation in the next round.

At the end of the first round, the invariant is satisfied because X consists of a single node whose neighbors in G form Y (step 5) and all edges incident to it belong to T_B (step 6). Now assume that the invariant is satisfied at the beginning of any round $i > 1$. When the newly selected node v is inserted into X (step 4), the contents of all other sets are updated (step 5). By definition, v is always selected in such a way that

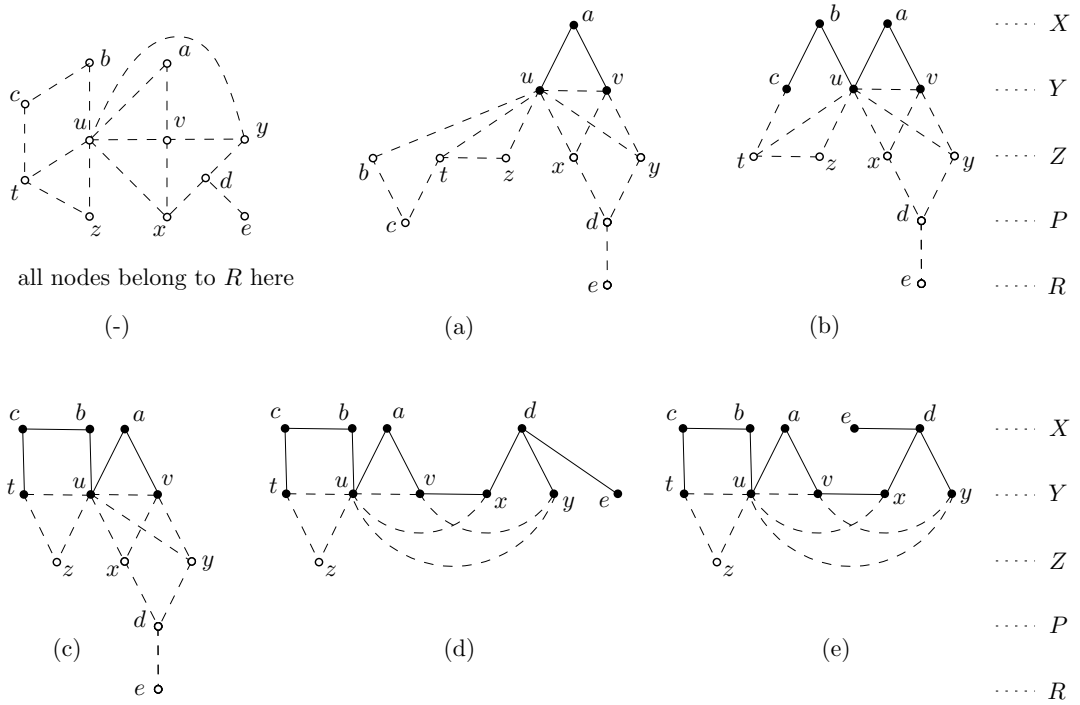

Figure 9: An example of running Algorithm 3L-Partition. The input is the graph shown in (-), and (a)–(e) present the configuration in each round after a new node has been saturated and the sets X, Y, Z, P, R as well as the backbone tree T_B have been updated. In (a)–(e), the current contents of each set X, Y, Z, P, R are displayed at different horizontal levels, and solid lines and black nodes belong to the backbone tree T_B. The saturated nodes are a, b, c, d, and e, chosen from different sets Y, Z, and P.

adding all edges incident to v will not create a cycle in T_B. If v was chosen from Y (this happens only when v has no horizontal incident edges), v is already connected to T_B so all edges incident to v (added in step 6) will be connected to the rest of T_B, too. Alternatively, if v comes from Z (this happens when all nodes in Y have horizontal edges outside of T_B) and v has exactly one neighbor $w \in Y$, then as soon as all edges incident to v are inserted, the new part of T_B gets connected to the old one via node w. Finally, if v was selected from P (this happens when all nodes in Y have horizontal edges outside of T_B and each node in Z has at least two neighbors in Y) then all edges incident to v are inserted into T_B. Note that when v was moved to X, all its neighbors in Z were moved to Y, forming at least one new horizontal edge in Y (formerly this edge lay across sets Y and Z). We use this new horizontal edge to connect a newly formed star with the remaining part of T_B. The algorithm exits its main loop when it attempts to select a new node for saturation from an empty set P, meaning that all nodes from V are already distributed among X, Y, Z, and in accordance with our invariant, this means that the backbone tree T_B is completed. □

Figure 9 illustrates the execution of Algorithm 3L-Partition.

The next two lemmas summarize some properties of Algorithm 3L-Partition that will be used later in the paper.

Lemma 4. *The three-layer partition output by Algorithm* 3L-Partition *satisfies the following:*

1. *Each node $y \in Y$ has an incident horizontal edge not belonging to T_B.*
2. *Each node $z \in Z$ has at least two neighbors in Y.*

Proof. To prove property 1, assume by contradiction that there exists a node $y \in Y$ with no horizontal edges outside of T_B. Observe that in this case, y can be saturated and thus moved to X, inserting into T_B all

remaining edges incident to y. Indeed, since all such edges led only to nodes in Z before y was saturated, their insertion does not create any cycles. Thus, property 1 holds.

Next, assume there is a node z in Z with at most one incident edge leading to layer Y. Then, we can also saturate z since all edges incident to z form a star that shares at most one node with T_B. Thus, no cycle is created, which proves property 2. □

Lemma 5. *Algorithm* 3L-PARTITION *can be implemented to run in* $O(|E|)$ *time.*

Proof. Below, we say that any node of G is colored *red* if it has already been tested for saturation (regardless of whether or not it was finally included in X), and *green* otherwise. All nodes are initially colored green and put in the set R. During the execution of steps 10–19 of 3L-PARTITION, a green node v is selected for saturation from one of the sets Y, Z, and P (in that order). Depending on which sets that v and its neighbors belong to, v either passes or fails the ensuing saturation test:

- If $v \in Y$ then v may be saturated if none of its neighbors belongs to Y.
- If $v \in Z$ then v may be saturated if only one neighbor of v belongs to Y.
- If $v \in P$ then v may always be saturated.

If v passes the test then it will be saturated in the next iteration and promoted to X; on the other hand, if v fails the test, this means that saturating v would create a cycle in T_B because of some edges already present in T_B. In both cases, the algorithm will never need to consider v for saturation again, and v can be safely colored red. This shows that each node needs to be considered for saturation by steps 10–19 only once. The saturation test takes $d(v)$ steps, i.e. $O(|E|)$ time for all vertices.

Moreover, every (green or red) neighbor of a node v that is subjected to the above saturation test may be promoted to a higher ranking set among Y, Z, P and R depending on the result of the saturation test for v. This happens in the step of updating these sets (Step 5). Also in the same step, neighbors of newly promoted vertices may also be promoted, and this process continues until every vertex is listed correctly, as a result of promotions. However, note that vertices which are already in the backbone tree or have the same ranking with their newly promoted neighbor, will not be promoted by definition. This implies that not every vertex at distance two or three of a newly saturated vertex is needed to be checked for update.

Under these observations, it is preferable to amortize the number of checks/updates with the edges of the graph. Each edge can participate in promotional checks whenever one of its two vertices is involved. If each of the two endvertices reaches its highest possible ranking, the edge also stops participating in any promotional check. Note that each vertex can be checked for promotion more than once, but always via a different edge. Given that every vertex can be checked/promoted at most three times, this implies that the number of checks/updates per edge is a constant factor, and thus the overall cost of the updating step is also $O(|E|)$.

Therefore, the complexity of the algorithm 3L-partition remains in $O(|E|)$. □

The three-layer partition method is employed in Section 4 and Section 6. We believe that this method may be of use for other problems as well in the future such as designing spanning trees with special properties, connected dominating sets, etc.

4. Efficient periodic graph traversal by an oblivious agent

The main result of this section is an algorithm named FINDWITNESSCYCLE that constructs a short witness cycle for any given graph G. By the remarks in Sections 2.2 and 2.3, this consequently solves the problem of periodic graph traversal by an oblivious agent.

According to Lemma 3, 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 first consider a restricted case of a *terse set of RH-cycles* in Section 4.1, for which it is possible to construct a spanning tree of G with no saturated nodes. In this case, we give a specialized algorithm that constructs a witness cycle of size $2n$. For the case

of arbitrary graphs considered in Section 4.2, we need a more involved argument, leading to a witness cycle of size $\leq (4 + \frac{1}{3})n$.

4.1. Terse set of RH-cycles

Suppose G is a graph that has a spanning tree T with no saturated nodes, i.e., for every node v, G contains some edge incident to v which does not belong to T. Here, we present an algorithm named TerseCycles that finds a very short witness cycle for this type of graphs.

Algorithm TerseCycles is listed in Figure 10. The idea is to first construct a spanning subgraph $\overrightarrow{K}$ of $\overrightarrow{G}$ that consists of RH-traversable nodes. For this purpose, TerseCycles takes the edges of a spanning tree T without saturated nodes as two-way edges in $\overrightarrow{K}$, inserts some extra one-way edges, and then runs a procedure named RestoreParity, outlined in Figure 11, to make sure that the number of one-way edges in $\overrightarrow{K}$ incident to each node is always even. Procedure RestoreParity visits each node v of the tree T in bottom-up order and counts all one-way edges incident to v; if this number is odd, the two-way edge leading to the parent is reduced to a one-way edge (whose direction is unspecified at this point in time). Note that the cumulative degree of the root must be even since the cumulative degree of all nodes before restoring parity is even. At the conclusion of RestoreParity we remove temporarily from $\overrightarrow{K}$ all two-way edges. All one-way edges forming a connected component are arranged to form a single cycle. Now all these cycles are merged into a single witness cycle by adding a minimal number of previously removed two-way edges.

Algorithm TerseCycles
Input: A graph G that admits a spanning tree with no saturated nodes.
Output: A witness cycle for G.

1: Let T be a spanning tree of G with no saturated nodes.
2: Construct $\overrightarrow{K}$ by replacing each edge $\{u, v\}$ in T by two directed edges (u, v) and (v, u).
3: For each node $v \in G$, add to $\overrightarrow{K}$ a one-way edge incident to v and belonging to $G \setminus T$.
4: Root T arbitrarily.
5: RestoreParity($\overrightarrow{K}, T, root(T)$)
6: Remove temporarily from $\overrightarrow{K}$ all two-way edges.
7: Take any port numbering as in Lemma 2 and produce a set $\mathfrak{C}$ of RH-cycles induced by it.
8: For any node v visited by two cycles entering v via ports i and j, swap i and j forming a single cycle.
9: Restore connectivity in $\overrightarrow{K}$ by adding back a minimal number of two-way edges.
10: Modify port numbers at each node to satisfy the construction in Lemma 2 while preserving the order of one-way edges.
11: **return** the cycle in $\mathfrak{C}$

Figure 10: Algorithm TerseCycles.

Lemma 6. *After the completion of Algorithm* TerseCycles*, every node of* $\overrightarrow{K}$ *is RH-traversable.*

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

Corollary 2. *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 $2n$.*

Proof. Observe that after the execution of Procedure TerseCycles, each node of $v \in \overrightarrow{K}$ has an even (and non-zero) number of one-way edges incident to it. One can provide direction to all one-way edges and port numbering at each node v so that all edges outgoing from and incoming to v belong to the same cycle. This

Procedure RESTOREPARITY
Input: A directed graph $\overrightarrow{K}$ (may be modified by the procedure), a tree T, and a node $v \in T$.
Output: 0 or 1 (the parity for node v in $\overrightarrow{K}$).

1: $P_v =$ (number of one-way edges incident to v in $\overrightarrow{K} \setminus T$) (mod 2)
2: **if** v is not a leaf in T **then**
3: **for** each node $c_v \in T$ that is a child of v **do**
4: $P_v = (P_v + RestoreParity(\overrightarrow{K}, T, c_v))$ (mod 2)
5: **end for**
6: **end if**
7: **if** $P_v = 1$ **then**
8: Reduce the two-way edge $(v, parent(v))$ to a one-way edge in $\overrightarrow{K}$ with unspecified direction.
9: **end if**
10: **return** P_v

Figure 11: Procedure RESTOREPARITY.

is done in two steps. First, the initial port numbering and the direction of one-way edges are obtained via greedy selection of one-way edges to form cycles. Later, if there is a node v that belongs to two or more cycles (based on one-way edges), the cycles are merged at v via direct port number manipulation. When this stage is done, the set of nodes in $\overrightarrow{K}$ is partitioned into components, with all nodes in the same component belonging to the same cycle based on one-way edges. Also note that each component is at distance one from some other component, where the components are connected by at least one two-way edge (this is a consequence of the fact that each node has at least two one-way edges incident to it). The two-way edge is used to connect the components. By successively connecting pairs of components at distance one, we end up with a single component, i.e., a witness cycle spanning all the nodes. It is important to only add a minimal set of two-way edges enforcing connectivity to actually end up with a single cycle. Note that for each one-way edge introduced in $\overrightarrow{K}$, a two-way edge from the spanning tree is reduced to a one-way edge during the restore parity process. This happens because one-way edges form a collection of stars and at least one endpoint of every one-way edge (in a star) is free. Thus, the number of all edges in the witness cycle is bounded by $2n$. □

By Corollary 2, Algorithm TERSECYCLES yields a small witness cycle for any graph that admits a spanning tree with no saturated nodes. This situation occurs for large, non-trivial classes of graphs, including two-connected graphs, graphs admitting two disjoint spanning trees, and many others. On the negative side, observe that in general, finding a spanning tree having no saturated nodes amounts to finding a Hamiltonian path, a problem known to be NP-hard even if restricted to 3-regular, planar graphs [6].

4.2. Construction of witness cycles in arbitrary graphs

Given any graph G, Algorithm FINDWITNESSCYCLE in Figure 12 can be used to construct a witness cycle for G. The algorithm is based on the following approach. First compute a spanning tree T of G. Let H_i for $i = 1, 2, \ldots, k$ be the connected components of $G \setminus T$, having, respectively, n_i nodes. For each such component, run Algorithm 3L-PARTITION, obtaining three sets X_i, Y_i, Z_i and a backbone tree T_i. Use the edges of T_i as two-way edges in G_i, insert extra one-way edges incident to the nodes of sets Y_i and Z_i, and apply the procedure RESTOREPARITY. We shall explain below how to do this so that the total number of edges in all resulting G_i-graphs is smaller than $(2 + \frac{1}{3})n$. Next, we let $\overrightarrow{K}$ be the union of T (where every edge of T is used both directions) with all the G_i-graphs, and take a port numbering that generates a set of RH-cycles as in Lemma 2. Finally, 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 according to Lemma 3.

Algorithm FindWitnessCycle
Input: A graph G.
Output: A witness cycle for G.

1: Compute a spanning tree T of G.
2: **for** each connected component H_i of $G \setminus T$ **do**
3: $(X_i, Y_i, Z_i, T_i) = $ 3L-Partition(H_i)
4: $G_i = T_i$ /* These edges are inserted into G_i as two-way edges. */
5: Form a set P_i by selecting, for each node in Z_i, two edges leading to Y_i.
Let $G_i = G_i \cup P_i$. /* One-way edges. */
6: Form a set S_i of independent stars spanning all nodes in Y_i that are not incident to P_i.
Let $G_i = G_i \cup S_i$. /* One-way edges. */
7: Root T_i arbitrarily and run RestoreParity$(G_i, T_i, root(T_i))$.
8: **end for**
9: $\overrightarrow{K} = T \cup G_1 \cup G_2 \cup \cdots \cup G_k$
10: Take any port numbering as in Lemma 2 and produce a set $\mathfrak{C}$ of RH-cycles induced by it.
11: Repeatedly apply *Merge3* or, if not possible, *EatSmall* to $\mathfrak{C}$ until neither rule can be applied.
12: **return** the largest cycle in $\mathfrak{C}$

Figure 12: Algorithm FindWitnessCycle.

Theorem 2. *For any n-node graph algorithm,* FindWitnessCycle *returns a witness cycle of size at most* $(4 + \frac{1}{3})n - 4$.

Proof. For each component H_i, we apply Algorithm 3L-Partition to obtain three sets X_i, Y_i, Z_i and a backbone tree T_i. By Lemma 4, we can add one-way edges incident to the nodes in Y_i as well as pairs of one-way edges incident to the nodes in Z_i and then apply Procedure RestoreParity to each G_i. Note that when each star S_i is constructed, we may do it in such a way that no path of length three or more is created. Indeed, otherwise we could remove a middle edge of any path of length three and the set of spanned nodes would remain the same. Hence, S_i is a forest of stars. Moreover, we can assume that only centers of such stars can be incident to edges forming P_i, otherwise any edge leading to a leaf node incident to P_i can be removed. Consequently, after termination of the "for" loop, each node of G is RH-traversable in $\overrightarrow{K}$. Moreover, since $\overrightarrow{K} \supseteq T$, $\overrightarrow{K}$ is two-way connected, so the conditions of Lemma 3 are satisfied. Hence, at the end of the algorithm, $\mathfrak{C}$ contains a witness cycle.

In order to bound the size of the witness cycle, we will bound the number of edges in $\overrightarrow{K}$. First note that $2n - 2$ edges originate from T (i.e., $n - 1$ two-way edges). Suppose that for each component G_i containing n_i nodes of $G \setminus T$, no one-way edges were added in lines 5 and 6, that is $P_i = \emptyset$ and $S_i = \emptyset$. Hence, the call to Procedure RestoreParity in line 7 did not modify G_i. In consequence, $2(n_i - 1)$ edges were added for G_i or $2(n_1 + n_2 + \cdots + n_k) - 2k$ in total. This value is maximized for $k = 1$, giving $2n - 2$ edges added in the "for" loop, and $4n - 4$ total edges in $\overrightarrow{K}$. The count remains the same if some $P_i \neq \emptyset$ since exactly two edges were added for each node of Z_i in line 5.

Now suppose that $S_i \neq \emptyset$ in line 6, for some components G_i. For each endpoint $v \in Y_i$ of a star belonging to S_i and a one-way edge e added for v in S_i in line 6, we check whether there is some other edge that was reduced (from two-way to one-way) during the call to RestoreParity on line 7. This happens when v is not incident to a horizontal edge of the backbone tree T_i, since one of the edges incident to v will then become a one-way edge. Thus, the addition of e is done at no extra cost, i.e., the total number of edges remains the same. However, when two endpoints of a horizontal edge are incident to two edges of T_i, only one such edge will be amortized. Consider then a collection of one-way horizontal edges, belonging to the backbone tree T_i with edges of S_i incident to both of their endpoints. The collection forms a forest. In each tree, pick a root arbitrarily and repeat the following process until there is only one edge left. Take an

arbitrary leaf and amortize the edge of S_i incident to it with the tree edge leading to the parent of the leaf. Remove the leaf and the edge that leads to its parent from further consideration. Note that in this case, amortization is one to one. When this process is finished, each tree has been reduced to one edge. In other words, we have a collection of independent one-way horizontal edges belonging to the backbone tree. Note that each such edge is associated with two independent edges of S_i. Clearly, the worst case happens when the forest was formed by independent one-way edges. This implies that the number of such horizontal edges is not larger than $\frac{n_i}{3}$.

Taking into consideration the maximal penalty that we have to pay for edges added in line 6 of the algorithm, the number of edges forming $\overrightarrow{K}$ is bounded by $(4+\frac{1}{3})n-4$. □

Next, we analyze the algorithm's time complexity.

Theorem 3. *Algorithm* FindWitnessCycle *can be implemented so it runs in $O(|E|)$ time.*

Proof. In $O(|E|)$ time we can find a spanning tree T of G and the connected components of $G\setminus T$. By Lemma 5, for each connected component G_i having n_i nodes and e_i edges, Algorithm 3L-Partition terminates in $O(e_i)$ time. The construction of sets P_i in line 5 and set S_i in line 6 as well as the call to procedure RestoreParity on line 7 are completed in $O(n_i)$ time. Altogether, the "for" loop terminates in $O(|E|)$ time. The construction of $\overrightarrow{K}$ in line 9 and $\mathfrak{C}$ in line 10 are done in time proportional to their sizes, i.e., $O(n)$.

We show now that line 11, where the rules *Merge3* and *EatSmall* are repeatedly applied, may be performed within $O(|E|)$ time. We chose any ordering γ of cycles and we attach to each edge a label corresponding to the cycle to which the edge belongs. Let C^* be the largest cycle according to γ and v be any node of C^*. We repeatedly apply rules *Merge3* (resulting cycle obtaining rank of $\gamma(C^*)$) and *EatSmall* to node v until no longer possible. Observe that, for each node v, this may be done in time proportional to the degree of node v in $\overrightarrow{K}$, resulting in the overall cost of $O(|E|)$. Each time, we traverse the edges of the cycle (or a part of the cycle) added to C^* and change their labels to $\gamma(C^*)$. When neither *Merge3* nor *EatSmall* is applicable to v we proceed to node v' (the actual successor of v in C^*) and repeat the procedure of applying rules *Merge3* or *EatSmall* to v'. Each edge introduced in C^* was relabeled exactly once, hence the overall cost of relabeling process is in $O(|E|)$. Although C^* changes dynamically and some nodes may be traversed many times we end up by traversing all nodes eventually in C^*. By Lemma 3, C^* becomes a witness cycle at the end of this process. Note that the complexity of each *Merge3* and *EatSmall* operation is proportional to the number of edges added to C^*. By Theorem 2, the overall complexity of line 11 is $O(|E|)$. □

Finally, we provide a lower bound example for the FindWitnessCycle algorithm which demonstrates that the bound stated in Theorem 2 for our algorithm is tight, up to an additive constant.

Lemma 7. *There exist graphs for which the* FindWitnessCycle *algorithm may produce a witness cycle of size $(4+\frac{1}{3})n-7$.*

Proof. Consider the graph in Figure 13.

The main part of the graph containing $n=(3k+1)$ nodes consists of k copies of four nodes $X_iY_{2i}Y_{2i+1}X_{i+1}$, for $i=1,2,\ldots,k$, where the last node of each but the last copy is identified with the first node of the next copy (see Figure 13). Moreover, an extra node Y_1 is adjacent to each of the nodes $Y_2,Y_3,\ldots,Y_{2k+1}$, and a node W is adjacent to all other nodes in the graph. Suppose that the star at node W is chosen by the algorithm as the spanning tree T, represented by the dotted edges in the picture. Algorithm 3L-Partition locates nodes $X_1,X_2,\ldots,X_k$ in set X and the nodes $Y_1,Y_2,\ldots,Y_{2k+1}$ in set Y (set Z is empty). Suppose that the backbone tree is the path $Y_1X_1\ldots X_{k+1}$ - represented by the solid edges in Figure 13. Since the algorithm adds one horizontal edge for each node from class Y, all edges incident to Y_1 are added to the structures. It is easy to see that the parity restoring procedure will chose the edges Y_iY_{i+1} as the one-way edges of the structure. In consequence, only $2k$ dashed edges and k thin solid edges in Figure 13 are chosen as one-way edges; all other edges (i.e., $3k+2$ dotted edges and $2k+1$ bold solid edges) are taken as two-way edges. This results in a witness cycle of size $13k+6$, i.e., containing $(4+\frac{1}{3})n-7$ edges. □

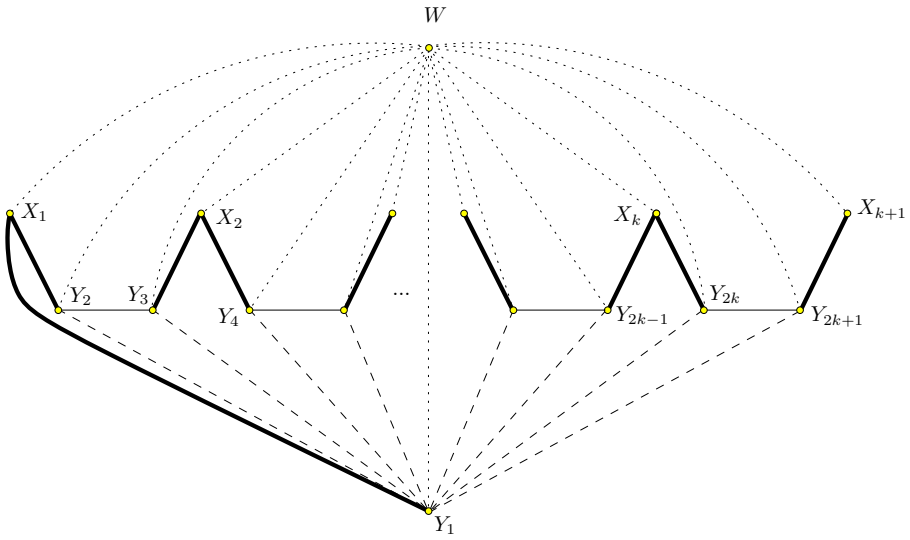

Figure 13: Example of a graph for which our algorithm gives a witness cycle of size not smaller than $(4 + \frac{1}{3})n - 7$.

5. A lower bound for oblivious agents

The previous section showed 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 a non-trivial lower bound of $2.8n - 2$.

Theorem 4. *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 $14k + 2l - 2$.*

Proof. First consider a single *diamond graph* G' with 5 nodes, defined on the left side of Figure 14. W.l.o.g., assume that the agent starts its traversal through the edge (v, x). By the structure of G', the agent then traverses the edge (x, u). Again, w.l.o.g., suppose the successor of (x, u) is the edge (u, y). Then there is only one feasible successor of (y, v), namely (v, z), because the other two edges either violate RH-traversability ((v, y)) or leave node z unvisited ((v, x)). Next, the only possible successor of (z, u) is (u, x) because (u, y) has already been traversed with a different predecessor and (u, z) violates RH-traversability. Similarly, the successor of (x, v) must be (v, y) and the successor of (y, u) must be (u, z). Therefore, each edge of G' must be used in both directions, and the witness cycle has length $12 = 2.4n$.

Next, consider the graph G having n nodes and consisting of a chain of k diamond graphs and path of l nodes attached to node u_k, as shown on the right side of Figure 14. Note that G contains $7k + (l - 1)$ edges. Assume that the agent start the graph traversal at node v_1. 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 to u_{i-1} from v_i, all nodes in G_i (as well as in all G_j for $j > i$) must have been visited. 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 analogous arguments as used above for the graph G' also apply to every G_i. Therefore, all edges of G must be traversed in both directions. □

Selecting $l = 0$ in Theorem 4 gives $n = 5k$, and we obtain the lower bound $14k - 2 = 2.8n - 2$ on the period length for oblivious agents.

6. Periodic graph traversal by an agent with constant memory

In this section, we focus on algorithms for periodic graph traversal by agents with constant memory. The main idea of the periodic graph traversal mechanism proposed in [8], and further developed in [7], is to visit

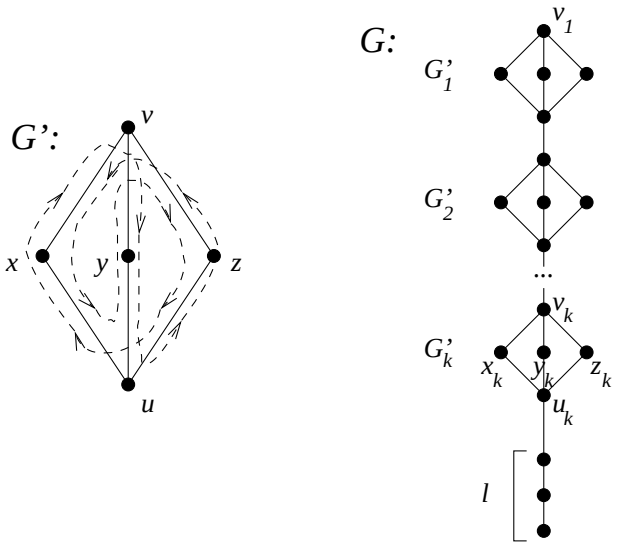

Figure 14: The diamond graph G' and the chain of diamond graphs used to prove the lower bound in Theorem 4. The witness cycle for G' shown on the left is $\langle v, x, u, y, v, z, u, x, v, y, u, z, v\rangle$ with length 12.

all nodes in the graph while traversing along an *Euler tour* of a (in [7], particularly chosen) spanning tree.

In [8], an arbitrary spanning tree T of G is rooted at any leaf, and the port numbers assigned so that at each non-root node v, port 1 is assigned to the port leading to the parent of v while ports $\{2, \ldots, i+1\}$ are assigned to the children of v in T and ports $\{i+2, \ldots, d_v\}$ to the remaining ports. Then, after entering node v via port 1, the agent recursively visits all subtrees accessible from v via ports $2, \ldots, i+1$, where i is the number of children of v. When the agent returns from the last (ith) child it either: (1) returns to its parent via port 1, when $i+1$ is also the degree of v (i.e., v is saturated in T); or (2) it attempts to visit another child of v by traversing the edge e associated with port $i+2$. In case (2), the agent learns at the other end of e that the port number is different from 1, i.e., that this node is not a child of v in the spanning tree T, and uses its constant memory to immediately backtrack via the same edge (first return to v and then directly to the parent of v), and then continues the tree traversal process. In these circumstances, the edge e is called a *penalty edge* since e does not belong to the spanning tree and an extra cost has to be charged for traversing it. Since the spanning tree has $n-1$ edges, and at each node the agent can be forced to traverse a penalty edge, the number of steps performed by the agent (equal to the length of the periodic tour) may be as large as $4n-2$ ($n-1$ edges of the spanning tree and n penalty edges, where each edge is traversed in both directions). The main result of [7] is the efficient construction of a specific spanning tree supported by a more advanced visiting mechanism stored in the agent's memory. They showed that the agent is able to avoid penalties at a fraction of at least $\frac{1}{8}n$ nodes. This in turn gave the length of the periodic tour not larger than $3.75n-2$.

In what follows, we show a new construction of the spanning tree, based on the earlier three-layer partition. This, supported by a new labeling mechanism together with slightly increased memory of the agent, allows us to avoid penalties at $\frac{1}{4}n$ nodes, resulting in a periodic tour of length $\leq 3.5n$. In the new scheme, shortcuts are created by performing "port swap operations", where some leaves in the spanning tree are connected to their parents via port 2 (in [7], this port is always assumed to be 1). The rationale behind this modification is to treat edges towards certain leaves as penalty edges (rather than the regular tree edges) and in turn to avoid visits beyond these leaves, i.e., to avoid unnecessary examination of certain penalty edges.

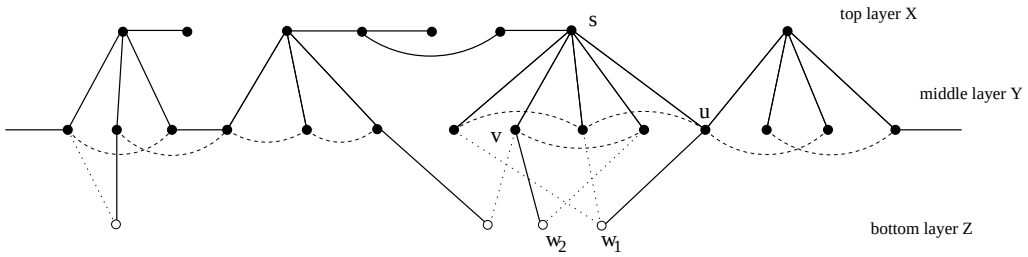

Figure 15: Fragment of the spanning tree with the root located to the right of w_1 and w_2.

6.1. The construction

Recall that the nodes of the input graph can be partitioned into three layers X, Y, and Z, where all nodes in X and Y are spanned by a backbone tree; see Section 3. The spanning tree T is obtained from the backbone tree by connecting every node in Z to one of its neighbors in Y. Also recall that every node $v \in X$ is *saturated*, i.e., all edges incident to v in G also belong 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 $2, \ldots, i+1$ in such a way 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.

Port swap operations:

Now, we modify 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 15), as well as in 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 15).

Every leaf w located in layer Z has an incident edge e outside of T that leads to some node v in Y by Lemma 4 (property 2). When swapping port numbers at some leaf w, we set the port number on the tree edge leading to the parent of w to 2. We call such an edge a *sham penalty edge* since it appears to be a penalty edge but in fact connects w to its parent in the spanning tree T. We also set the port number on the lower end of e to 1. 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 recognized 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 $2, \ldots, i+1$ 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 [7] 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 [7]) in a node that is not incident to the lower end of a sham penalty edge.

Lemma 8. *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 $\mathcal{A}^+$.*

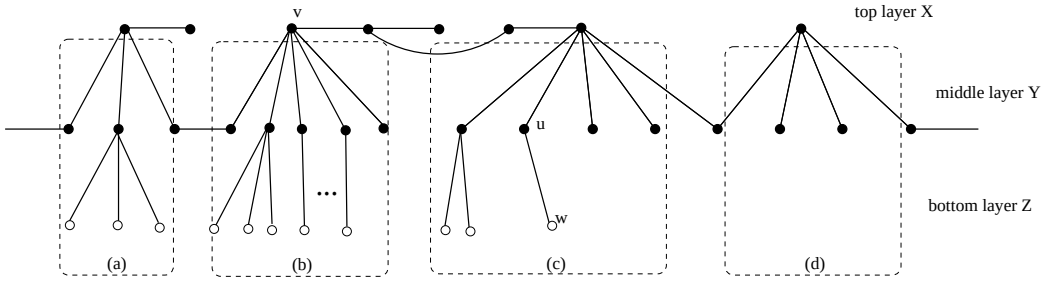

Figure 16: Cases (a), (b), (c), and (d) of the local amortization argument used in the proof of Theorem 5.

Proof. It suffices to prove that no difficulty arises at nodes with numbers affected by the modified labeling scheme.
Case C1: First consider the case when the port numbers are swapped at some node w_1 which is a single child in Z of a bonding node u (see Figure 15). When during traversal the agent returns from the subtree rooted in a child of u accessible via port $i-1$, it enters via port i the edge leading to w_1. This edge is interpreted as a penalty edge and the agent after visiting w_1 immediately returns to u and then it goes with no further action to the parent of u. Note that if the labeling was not changed the agent would act similarly; however, it would additionally examine a penalty edge located at w_1. Thus, thanks to the new labeling scheme we save one penalty at the node w_1.
Case C2: Next, consider the case when the port numbers are swapped at a single child w_2 of a local node v such that v has no siblings different from leaves to its right (accessible via larger ports), see Figure 15. Assume that s is the (saturated) parent of v and port i at v leads to w_2. When during traversal the agent returns from the subtree rooted in a child of v accessible via port $i-1$, it enters via port i the edge leading to w_2. When it learns that the port label at w_2 is different from 1, it interprets the sham penalty edge linking v and w_2 as the penalty edge. The agent immediately returns to v while switching to the leaf recognition state [7] (v would be interpreted as the first leaf of s). This means that all remaining leaves accessible from s (if any) will be visited at no extra charge, i.e., without paying penalty at them. Thus, the agent does not miss the node w_2 and it also saves penalty at w_2 and possibly at all leaves that are siblings of v. □

6.2. Analysis

Theorem 5. *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 $3.5n-2$.*

Proof. The main line of the proof explores the fact that the fraction of nodes at which the agent manages to save on penalties is at least $\frac{1}{4}$. The proof is split into global and local amortization arguments.

- **Global amortization** [saturated nodes amortize all bonding nodes and single children of saturated nodes]

  Note that in a three-layer partition with k saturated nodes, there are at most $2k-2$ bonding nodes since introducing a new saturated node implies the creation of at most two bonding nodes. Also note that there are at most k single leaves (with no siblings) that are children of saturated nodes. In the global amortization argument, we assume that at these nodes, i.e., all bonding nodes and all single leaves of saturated nodes, in the worst case the agent always pays penalty (examines the penalty edge). Fortunately, all of these $\leq 3k-2$ nodes ($2k-2$ bonding nodes and k single leaves of saturated nodes) can be amortized by k saturated nodes. Thus, as required, the fraction of nodes where the agent does not pay penalty is $\frac{1}{4}$. For all other nodes in T, we use the local amortization argument.

- **Local amortization** [direct amortization of nodes within small subtrees]

  The local amortization argument is used solely on two-layer subtrees accessible from saturated nodes, i.e., formed of local nodes and (possibly) their children, cases (a), (b), (c), and (d), see Figure 16, as well as on leaves accessible from bonding nodes, cases (e) and (f).

  The local amortization argument involving local nodes is split into cases (a), (b), (c), and (d) in relation to the size of subtrees rooted in local nodes. We start the analysis with the largest subtrees in case (a) and gradually move towards smaller structures in cases (b) and (c), finishing with single local nodes in case (d).

  (a) Consider any subtree T_S with at least two children rooted in a local node. In this case, the initial labeling remains unchanged. During traversal of T_S the agent pays penalties at the local node and at its first child where it switches to the leaf recognition state. In this state, no further penalties at the leaves of T_S are paid. Since the number of children $i \geq 2$ the fraction of nodes in the subtree without penalties is at least $\frac{1}{3}$.

  (b) Now consider the case where a saturated node v has at least two children (local nodes) with single children (two *extended leaves* according to the notation from [7]) accessible from v. In this case, the number of penalties paid during traversal of all extended leaves is limited to two since the penalties are paid at both nodes of the first extended leaf where the agent switches to extended leaves recognition state. The remaining nodes of the extended leaves are visited at no extra cost. In this case, the fraction of nodes without penalties is at least $\frac{1}{2}$.

  (c) Next, consider the case where a saturated node has only one extended leaf (a local node u and its single child w) possibly followed by some regular leaves formed of local nodes. In this case, the initial labeling is changed and the sham penalty edge (u, w) is introduced (case C1 in the proof of Lemma 8). When the agent visits the extended leaf it enters the sham penalty edge interpreting it as the penalty edge. Moreover, if u has sibling leaves all of them are visited at no extra cost since after visiting a sham penalty edge the agent is in the leaf search state ([7]). Thus, in this case there is no penalty to be paid, i.e. the fraction of nodes where the penalty is not paid is 1.

  (d) It may happen that a saturated node has several children that are leaves in T not preceded by an extended leaf. In this case, the penalty is paid only at the first leaf and all other leaves are visited (in leaf search mode) at no extra cost. Therefore, a penalty of at least $\frac{1}{2}$ is avoided. (Recall that the case when a saturated node has only one child that is a leaf in T was already considered in the global amortization argument.)

  The remaining cases of the local amortization argument refers to the leaves accessible via bonding nodes.

  (e) When a bonding node has at least two children (all children are leaves) during traversal the agent pays penalty only at the first child while all other children are visited at no extra cost (thanks to the leaf search state). Thus, the fraction of nodes (leaves) where the penalty is avoided is at least $\frac{1}{2}$.

  (f) Finally, consider the case where a bonding node u has exactly one child w (case C1 from the proof of Lemma 8). In this case, thanks to the sham penalty edge (u, w), no penalty is paid at w, i.e., the fraction of nodes without penalties is 1.

In conclusion, the fraction of nodes at which the penalty is avoided is bounded from below by $\frac{1}{4}$ in all considered cases. Thus, the number of visited penalty edges is bounded by $\frac{3}{4}n$. Since the number of edges in the spanning tree is $n-1$ the agent visits at most $\frac{7}{4}n-1$ edges where each edge is visited in both directions. This concludes the proof that the length of the tour is bounded by $3.5n-2$. □

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.

7. Concluding remarks

The following table summarizes our new results on the period length of periodic graph traversal:

	Lower bound	Upper bound
Oblivious agent:	$2.8n - 2$ (Section 5)	$(4 + \frac{1}{3})n - 4$ (Section 4.2)
Constant-memory agent:	$2n - 2$ (folklore; see Section 1.2)	$3.5n - 2$ (Section 6)

For the special class of graphs defined in Section 4.1, the upper bound for oblivious agents was improved to $2n$. However, for general graphs, there is still a substantial gap between the known lower and upper bounds, both in the oblivious agent case and the constant-memory agent case. The major open problem is to close these gaps.[5]

Also, further studies on trade-offs between the length of the periodic tour and the memory of the agent are needed (our algorithm basically uses the same amount of memory as the one in [7], see section 6.1).

In particular, note that the only known lower bound $2n-2$ for agents with memory holds independently of the size of the available memory, and it refers to trees.

Another open problem is to generalize our techniques to *edge-weighted* graphs, where the notion of period length can be naturally extended to the total weight of all edges traversed in a period. For the constant-memory agent case, it may be useful to try to modify the three-layer partition step employed in Section 6 in such a way that the spanning tree T obtained from the backbone tree by connecting every node in Z to one neighbor in Y becomes a minimum weight spanning tree.

Finally, it would be interesting to further study the computational complexity of the problem of finding the shortest witness cycle for an input graph. Computing the shortest witness cycle corresponds to locating a Hamiltonian cycle in certain graphs, which is an NP-hard problem, so one should consider polynomial-time approximability in the general case.

Acknowledgments

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] L. Budach. Automata and labyrinths. *Mathematische Nachrichten*, pages 195–282, 1978.

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

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

[4] S. Dobrev, J. Jansson, K. Sadakane, and W.-K. Sung. Finding short right-hand-on-the-wall walks in graphs. In *Proceedings of the 12th Colloquium on Structural Information and Communication Complexity* (SIROCCO 2005), volume 3499 of *Lecture Notes in Computer Science*, pages 127–139. Springer-Verlag, 2005.

[5] P. Fraigniaud, D. Ilcinkas, S. Rajsbaum, and S. Tixeuil. The reduced automata technique for graph exploration space lower bounds. In *Theoretical Computer Science : Essays in Memory of Shimon Even*, volume 3895 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag, 2006.

[6] M.R. Garey, D.S. Johnson, and R.E. Tarjan. The planar Hamiltonian circuit problem is NP-complete. *SIAM Journal on Computing*, 5(4):704–714, 1976.

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

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

[5] After the publication of the preliminary (conference) version of this paper, Kosowski and Navarra [9] improved the upper bound for oblivious agents to $4n - 2$.

[9] A. Kosowski and A. Navarra. Graph decomposition for improving memoryless periodic exploration. In *Proceedings of the 34th International Symposium on Mathematical Foundations of Computer Science* (MFCS 2009), volume 5734 of *Lecture Notes in Computer Science*, pages 501–512. Springer-Verlag, 2009.

[10] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[11] O. Reingold. Undirected ST-connectivity in log-space. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing* (STOC 2005), pages 376–385, 2005.

[12] H.A. Rollik. Automaten in planaren graphen. *Acta Informatica*, 13:287–298, 1980.