

On the Power of Local Orientations^{*}

Monika Steinová

Department of Computer Science, ETH Zurich, Switzerland,
monika.steinova@inf.ethz.ch

Abstract. We consider a network represented by a simple connected undirected graph with N anonymous nodes that have local orientations, i.e. incident edges of each vertex have locally-unique labels – port names. We define a pre-processing phase that enables a right-hand rule using agent (RH-agent) to traverse the entire graph. For this phase we design an algorithm for an agent that performs the precomputation. The agent will alter the network by modifying the local orientations using a simple operation of exchanging two local labels in each step. We show a polynomial-time algorithm for this precomputation that needs only one pebble and $O(\log N)$ memory in the agent.

Furthermore we design a similar algorithm where the memory that the agent uses for the precomputation is decreased to $O(1)$. In this case, the agent is not able to perform some operations by itself due to the lack of memory and needs support from the environment.

KEYWORDS: Mobile computing, local orientation, right-hand rule

1 Introduction

The problem of visiting all nodes of a graph arises when searching for data in a network. A special case of this problem, when nodes need to be visited in a periodic manner, is often required in network maintenance. In this paper, we consider the task of periodic exploration by a mobile entity (called *robot* or *agent*) that is equipped with a small memory.

We consider an undirected graph that is anonymous, i.e. nodes in the graph are neither labeled nor marked. The way how we allow the agent to perceive the environment are the *local orientations*: while visiting a node v , the agent is able to distinguish between incident edges that are labeled by numbers $1 \dots d_v$, where d_v is the degree of the vertex v . A local orientation uniquely determines the ordering of the incident edges of a vertex.

Dobrev et al. [7] considered the problem of perpetual traversal (an agent visits every node infinitely many times in a periodic manner) in an anonymous undirected graph $G = (V, E)$. The following traversal algorithm called *right-hand rule* is fixed: “Start by taking the edge with the smallest label. Afterwards, whenever you come to a node, continue by taking the successor edge (in the local

^{*} This research was done as a part of author’s Master Thesis in Comenius University, Bratislava, Slovakia.

orientation) to the edge via which you have arrived". They showed that in every undirected graph the local orientation can be assigned in such a way that the agent obeying the right-hand rule can traverse all vertices in a periodic manner. Moreover, they designed an algorithm that, using the knowledge of the entire graph G , is able to precompute this local orientation so that the agent visits every node in at most $10|V|$ moves.

Further investigation of this problem was done by Ilcinkas and by Gąsieniec et al. [11,12]. In [12] the author continues in the study of the problem by changing the traversal algorithm and thus decreasing the number of moves in perpetual traversal. By making the agent more complex (a fixed finite automaton with three states) the author designed an algorithm that is able to preprocess the graph so that the period of the traversal is at most $4|V| - 2$. In the most recent publication [11] the authors use a larger fixed deterministic finite automaton, and are thereby able to decrease the period length to at most $3.75|V| - 2$.

1.1 Our results

In this paper we have returned to the original problem investigated by Dobrev et al. in [7] – we want to preprocess the graph so that an agent obeying the right-hand rule will visit all vertices. We are answering the questions: Is it possible to do the changes of the local orientation locally? How much memory do we need to make these local changes?

We consider a simple undirected anonymous graph $G = (V, E)$ with a local orientation. We design an algorithm for a single agent A , that is inserted into the graph and performs the precomputation so that later an agent B obeying the right-hand rule is able to visit all nodes in a periodic manner. The agent A recomputes the local orientations in polynomial time using $O(\log |V|)$ bits of memory and one pebble. Furthermore, we modify this algorithm using rotations of local orientations to decrease the memory in the agent to a constant, at the expense of losing the termination detection.

1.2 Related Work

The task of searching and exploring the unknown environment is studied under many different conditions. The environment is modeled either as geometric plane with obstacles or as graph-based, in which the edges give constraints on possible moves between the nodes. The first approach is often used to model landscape with possible obstacles where a single or multiple mobile entities are located. They navigate in the environment cooperatively or independently using sensors, vision, etc. to fulfill a particular mission. For more details see for example the survey [15].

The graph-based approach was investigated under various assumptions. The directed version where the mobile entities (called *robots* or *agents*) have to explore a strongly connected directed graph was extensively studied in the literature ([1,5,8]). In this scenario, agents are able to move only along the directed edges of the graph. The undirected version where the edges of a graph can be traversed

in both directions was studied in [2, 6, 9, 13, 14]. The labeling of the graph is studied under two different assumptions: either it is assumed that nodes of the graph have unique labels and the agent is able to recognize them (see [5, 13]) or it is assumed that nodes are anonymous (see [16]) with no identifiers. In the second case, the common requirement is that agents can locally distinguish between incident edges.

As the graph can be large and there can be more agents independently or cooperatively operating in it, various complexity measures were considered. Therefore not only the time efficiency but also both the local memory of the agent and its ability to mark the graph is investigated. The agent traversing the graph is often modeled as a finite automaton. This basic model allows the agent to use a constant amount of memory that is represented by its states. In 1978 Budach [4] proved that it is not possible to explore an arbitrary graph by a finite automaton without marking any node. Later Rollik [16] (improved by Fraigniaud et al. in [10]) proved that no finite group of finite automata can cooperatively explore all cubic planar graphs. These negative results lead to increasing the constant memory the finite automaton may use. The agent is given a *pebble* that can be dropped in a node to mark it and later taken and possibly moved to different nodes. Bender et al. [3] show that strongly connected directed graphs with n vertices can be traversed by an agent with a single pebble that knows the upper bound on the number of vertices, or with $\Theta(\log \log n)$ pebbles if no upper bound is known.

1.3 Outline of the paper

In Section 2 we introduce the notation used and give basic definitions and properties. Section 3 contains the algorithm for precomputing the graph in a local manner with $O(\log |V|)$ memory in the agent and one pebble in the graph. A modification of the algorithm from Section 3 is presented in Section 4. Here, rotations of the local orientation in vertices are used to decrease the memory that agent needs for the precomputation. On one hand, the memory complexity is minimized but on the other hand, the termination detection using so little memory remains an open problem. However, for example marking the initial edge in the graph can be used to terminate this algorithm. Section 5 contains the conclusion and the discussion about open problems.

2 Notation and preliminaries

Let G be a simple, connected, undirected graph. The degree of vertex v will be denoted d_v . In our model we assume that each vertex is able to distinguish its incident edges by assigning unique labels to them. Note that by such an assignment each edge has two labels – one in each endpoint. For simplicity we assume that for a vertex v these labels are $1, 2, \dots, d_v$. This labeling is called *local orientation* and denoted π_v .

Note that the existence of a local orientation in every vertex is a very natural requirement. Indeed, in order to traverse the graph, agents have to distinguish

between incident edges. The labels of the incident edges in a vertex v define a natural cyclic ordering, where $\text{succ}_v(e) = (\pi_v(e) \bmod d_v) + 1$ is the successor function and the corresponding predecessor function $\text{pred}_v(e)$ is defined similarly.

We want to construct a cycle in G that contains all vertices of G and satisfies the *right-hand rule* (RH-rule for short): “If the vertex v is entered by the edge e , leave it using an edge with label $\text{succ}_v(e)$.” We will call the traversal according to the RH-rule *RH-traversal*. More precisely we want to construct a cycle that contains all vertices such that RH-traversal started in any vertex of G by edge with label 1 leads to a visit of all nodes of G . We try to construct the cycle in a local manner with minimization of memory requirements for computations. The local computations are done by inserting an agent into the graph G and altering the local orientation by using a small amount of memory.

Rotation of a local orientation π_v by k steps is a local orientation π'_v such that $(\pi_v(e) + k) \bmod d_v + 1 = \pi'_v(e)$. An obvious Lemma follows.

Lemma 1. *Rotations of local orientation in a vertex of a simple, undirected, connected graph G do not have any influence on traversals according to the RH-rule in G . Formally, π_v and π'_v define the same $\text{succ}_v(\cdot)$ function.*

In certain situations we will need to speak about traversing edges in a certain direction. In such cases we will call the directed edges *arcs* and undirected edges *edges*. In the further text, we will denote the number of vertices of the graph by $N = |V|$ and the number of its edges by $M = |E|$.

Note that RH-traversal is reversible and therefore the trajectory of RH-traversal is always cycle. These trajectories are called *RH-cycles*. The initial arc of an RH-traversal unambiguously determines a RH-cycle. The graph G is always a union of disjoint RH-cycles.

3 Algorithm *MergeCycles*

Our algorithm will connect multiple RH-cycles into a single RH-cycle that passes through all vertices of graph G . This is done by applying rules *Merge3* and *EatSmall* from [7] in a local manner. In this section we will discuss the rules, their implementation and we show some interesting properties of the algorithm. In the beginning we choose one RH-cycle and call it the *witness cycle*. By applying rules *Merge3* and *EatSmall* the witness cycle is prolonged and new vertices are added. We show that if no rule can be applied in any vertex of the witness cycle then it contains all vertices of graph G . To know when to terminate the algorithm, we will count the number of steps while no rule could be applied.

The desired output of this algorithm is a new labeling of the edges in graph G such that the RH-traversal started in any vertex using the edge with label 1 will visit all vertices in G . Note that after the final witness cycle is constructed, our preprocessing agent can traverse the witness cycle once and rotate labeling in each vertex so that the outgoing arc with label 1 will be an arc of the witness cycle.

3.1 Rules Merge3 and EatSmall

Rule Merge3: [7] To apply this rule, we have to find three different RH-cycles and then connected them by exchanging labels so that the remaining RH-cycles stay unchanged. More precisely, let x_1 , x_2 and x_3 be three incoming arcs to a vertex v that define three different RH-cycles C_1 , C_2 and C_3 respectively. Then we change the ordering of the edges in v so that the successor of x_2 becomes the successor of x_1 , successor of x_3 becomes successor of x_2 , and successor of x_1 becomes successor of x_3 , while keeping the rest of the relative ordering edges unchanged. For illustration see Figure 1.

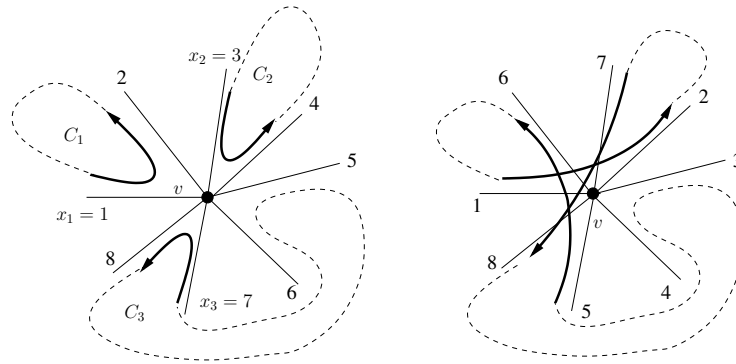


Fig. 1. Applying rule *Merge3*

To apply Merge3, we need to identify three arcs that enter v and determine three different RH-cycles (C_1 , C_2 and C_3). Note that we entered v while RH-traversing the witness cycle. Therefore we pick the arc used to enter v as one of those three arcs. The remaining two arcs are found by sequentially testing all incoming arcs in the order given by the current local orientation in v . The pebble is used to mark the processed vertex. To check whether two incoming arcs e_1 and e_2 define different RH-cycles, the agent RH-traverses the cycle defined by e_2 and checks whether it encounters e_1 before returning to e_2 . The agent will either find three different RH-cycles and merge them together, or it will ensure that no three different RH-cycles pass through vertex v .

Note that rule *Merge3* can only be applied finitely many times, as after each application of the rule *Merge3* the number of cycles in the graph decreases by two. If there are three different RH-cycles in vertex v , our approach always detects these cycles and rule *Merge3* is applied. Notice that for these operations we need one pebble and $O(1)$ local variables of size $O(\log N)$ bits in the agent's memory. The time complexity of one application of the rule *Merge3* in a vertex v is $O(Md_v)$.

Rule EatSmall: [7] To apply this rule in a vertex v , we have to find two different RH-cycles where the vertex v appears in one of them at least twice.

More precisely, let x and y be two incoming arcs to a vertex v that define two different RH-cycles C_1 and C_2 respectively; let z be the incoming arc to the vertex v by which C_1 returns to v after leaving via the successor of arc x . Then we modify the ordering of the edges in v such that the successor of x becomes the successor of y , the old successor of y becomes the successor of z , and the old successor of z becomes the successor of x , while preserving the order of the remaining edges. The application of the rule is shown in Figure 2. Note that the rule is applied on an ordered triplet of edges (y, x, z) .

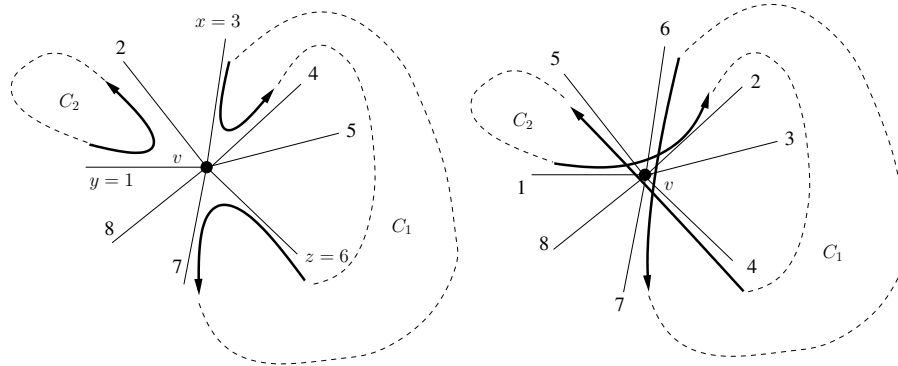


Fig. 2. Applying rule *EatSmall*. The edge y is the incoming edge via which the agent entered the vertex v and thus it is a part of the witness cycle.

The local version of rule *EatSmall* in a vertex v is similar to the *Merge3* rule. Note that blindly applying the rule *EatSmall* may lead to deadlocks when a part of a cycle will be transferred back and forth between two cycles. To prevent the deadlocks, we will always use our witness cycle as the cycle C_2 in our algorithm.

Note that the rule *EatSmall* can only be applied finitely many times as after an application of the rule *EatSmall* the witness cycle is prolonged. By similar argumentation as for rule *Merge3*, if two cycles C_1 and C_2 with required properties pass through vertex v , they are found and the rule *EatSmall* is applied. Again, one pebble and $O(1)$ local variables of size $O(\log N)$ are needed, and the time complexity of one application of the rule *EatSmall* in a vertex v is $O(Md_v)$.

The rule *EatSmall* has to be applied with care. More precisely, according to Figure 2, if the edge z (the edge via which the agent enters for the second time the vertex v in the RH-traversal of the cycle C_1 that started via the edge x) is the successor of edge y , the rule *EatSmall* will fail. The problem that occurs here is that setting the successor of z to be the successor of y means setting the successor of z to z and that is not allowed. However, our algorithm processes edges incident with a vertex in order given by the local orientation, starting by the arc of the witness cycle (in Figure 2 the arc y). Thus we will never encounter such a situation. In sequential testing of edges, the agent first picks edge x and then verifies the existence of edge z . If z is found, it will be the edge that was

not processed at that point by the agent. Thus in the ordering given by the local orientation and the initial incoming edge to v edge z is an edge that is always later than the edge x .

Lemma 2. *Let $G = (V, E)$ be a simple undirected connected graph. Let $v \in V$ and $x \in V$ be two neighbouring vertices such that the witness cycle (denoted by W) passes through v but not through x . Then, either the rule *Merge3* or *EatSmall* can be applied in vertex v so that vertex x will be added to the witness cycle.*

Proof. As vertices v and x are neighbouring, the arcs $\overrightarrow{(v, x)}$ and $\overleftarrow{(v, x)}$ determine two RH-cycles. If these RH-cycles are different, three different RH-cycles pass through vertex v and rule *Merge3* can be applied. In the case where both arcs determine the same RH-cycle C , we claim that C passes through vertex v at least twice and thus the rule *EatSmall* can be applied here. By the contradiction: if the RH-cycle C passes through vertex v only once, the successive arc of $\overrightarrow{(x, v)}$ is arc $\overleftarrow{(x, v)}$. Then by the definition of the cyclical ordering $\text{succ}(\cdot)$, vertex v has degree 1 and that is the contradiction.

Lemma 3. *Execute the algorithm *MergeCycles* on a simple connected undirected graph G and denote the resulting witness cycle by W . For any RH-cycle C in the resulting graph there is a vertex w such that $w \in C$ and $w \in W$.*

Proof. As the graph is connected, there exists a path between a vertex in witness cycle W and a vertex in C . Therefore by Lemma 2 rules *Merge3* and *EatSmall* were applied and all vertices on this path belong to W .

Lemma 4. *During our algorithm *MergeCycles* in each vertex v of a simple connected undirected graph $G = (V, E)$ we apply all the rules *Merge3* and *EatSmall* at the first time when the vertex is processed. In other words, once we finish processing a particular vertex v for the first time, this vertex is done – no rule applications in v will be possible in the future.*

Proof. Follows from previous discussion.

Theorem 1. *Let G be a simple connected undirected graph. Suppose that the algorithm *MergeCycles* already terminated on G . Let W be the witness cycle constructed by the algorithm. Then W contains all the vertices in G .*

Proof. By contradiction. Let $v \notin W$. Then there is a path between a vertex in W and v . Take the first vertex x on this path that is not in W (its predecessor y is in W). By Lemma 2 either *Merge3* or *EatSmall* can be applied in y .

The time complexity of our algorithm *MergeCycles* is $O(M^2\Delta)$, where Δ is the maximal degree of a vertex. The RH-cycle consists of at most all edges of graph in each direction and thus its length is $O(M)$. The termination detection is solved by comparing the number of the consecutive vertices of the cycle where no rule is applied with the length of the witness cycle. This can be done in $O(\log N)$ memory. To sum up, we used a few variables of size $O(\log N)$ in the agent and a single pebble.

4 Algorithm *MergeCycles+* using constant memory

In this section we present the modification of algorithm *MergeCycles* where rotating local orientations will enable us to decrease the memory needed in the agent to apply rules *Merge3* and *EatSmall*. For now, we will assume that the vertices of graph G have degrees greater than one.

Definition 1. Let $G = (V, E)$ be a simple undirected connected graph. Denote incident edges of vertex v by e_1, \dots, e_{d_v} , so that $e_i = (v, u_i)$. Let π_v, π_{u_i} be the local orientations in vertices v, u_i respectively. We will call label $\pi_v(e_i)$ the inner label of the edge e_i in vertex v and the label $\pi_{u_i}(e_i)$ the outer label of the edge e_i in vertex v .

As by Lemma 1 the rotations of local orientation do not have any influence on RH-rule, we will use these rotations to store information in local orientations. The general idea of the algorithm *MergeCycles+* is to set all outer labels of the incident edges of a vertex v to 1 and then find and mark representative arcs that are used in rules *Merge3* and *EatSmall* by outer label 2.

4.1 Memory needed by our algorithm

When trying to identify and minimize the memory requirements, we need to be more precise on how the local orientation changes are realized. Note that we can not get rid of the variable that our agent uses for storing the label of the incoming edge. By losing of this information the agent lost the sense of direction and it will not be able to distinguish between edges in G . Note that we will only use this as a read-only variable. Moreover, depending on the hardware realization, this variable does not even have to be stored in the agent's memory – the agent may be able to determine its value on demand from its environment. Thus this variable will not be counted in the agent's memory requirements.

In our algorithm the following local operations are necessary: rotation of the entire local orientation, application of rule *Merge3* and application of rule *EatSmall*. To realize these operations, the agent is able to use two primitives that relabel edges in the current vertex. The first primitive rotates the local orientation in vertex v by one. By multiple uses of this primitive the agent is able to rotate the local orientation so that the label of the incoming edge is 1 or 2. The second primitive operates in a vertex where three incident edges are marked by outer label 2 and the rest have outer label 1. It picks the three marked edges and performs changes in the local orientation as it is done in rules *Merge3* and *EatSmall*.¹ (As we know that the edges for either rule are tested sequentially, their correct order can be determined from the local orientation.)

The last part of the algorithm *MergeCycles* which forces the agent to use $\Omega(\log N)$ memory is the knowledge of the length of the witness cycle and the counter for the number of vertices visited in a row in which neither the rule

¹ The application of the rules *Merge3* and *EatSmall* are in fact the same, the difference is only in the conditions the edges have to satisfy.

Merge3 nor rule *EatSmall* could be applied. The price for getting rid of these computations is that we lose the termination detection – the precomputing agent will RH-traverse the witness cycle forever.

As the agent will execute the algorithm *MergeCycles+* forever, by attempting to apply the modified rules of *Merge3* and *EatSmall*, the local orientations will be changing. This does not match the desired output of the algorithm *MergeCycles*: There is no guarantee that if the RH-agent starts via the edge with label 1, then it will traverse the witness cycle. (Note that e.g. by marking the starting edge of the algorithm, termination detection can be solved easily and then the agent obeying the RH-rule can start the traversal via the marked edge and be sure to traverse the witness cycle.)

4.2 Basic instructions

The entire algorithm for our agent can be specified using the following basic instructions (where e is the edge used to enter vertex v):

- place/pick up pebble, test for its presence in v
- leave vertex v via e
- leave vertex v via $\text{succ}_v(e)$
- two tests: whether the edge e has inner label 1 or 2
- apply one of the primitives discussed in Subsection 4.1

Using these instructions we can build the following procedures (again, where $e = (u_k, v)$ is the incoming edge via which the agent entered v):

- rotate local orientation so that $\pi_v(e) = 1$, or so that $\pi_v(e) = 2$
- two tests: $\pi_{u_k}(e) = 1?$ and $\pi_{u_k}(e) = 2?$
- remember the incoming edge e by setting $\pi_v(e) = 1$
- rotate local orientation in u_k so that $\pi_{u_k}(e) = 1$ or $\pi_{u_k}(e) = 2$
- traverse all u_i in the order given by local orientation π_v
- for all $i \in \{1, \dots, d_v\}$ set $\pi_{u_i}(v, u_i) = 1$ and afterwards find e
(This is realized as follows: rotate π_v so that $\pi_v(e) = 1$, set $\pi_{u_k}(e) = 1$ and then repeatedly traverse via $\text{succ}_v(e)$, set the outer label of the edge to 1, and return back to v until the incoming edge has the inner label 1.)

We will show how (according to the order given by the local orientation) we can process all edges incident with a vertex v . During this processing we will use suitable rotations of local orientations in such a way that whenever we enter v we will be able to identify the initial and the currently processed edge. We start by placing a pebble into v , setting outer label 2 to the initial edge and outer label 1 to the remaining edges. We set the inner label of the initial edge to 1 and we start to process it. Whenever we are going to process the next edge, we rotate the local orientation by one so that the currently processed edge always has the inner label 1. If after the rotation of local orientation we find out that the edge to process already has outer label 2, we know that we processed all incident edges and we are done.

The test for two disjoint RH-cycles passing through vertex v is done as follows: The agent sets the outer labels of incident edges so that two tested edges e_1 and e_2 (in the incoming direction they represent two tested RH-cycles) have outer label 2 and the rest of incident edges have outer label 1. Then it rotates the local orientation in v so that e_1 has inner label 1, inserts pebble to v and RH-traverses RH-cycle via the successor of edge e_1 . Sooner or later the agent will enter v via edge e_1 (and recognize this thanks to the pebble and the local orientation). The cycles are disjoint iff the agent did not enter v via e_2 during this RH-traversal.

The discussed operations will now be used to describe the modified rules *Merge3* and *EatSmall*.

4.3 Changing the rule *Merge3* to *Merge3+*

The general idea is to set all outer labels of the incident edges of vertex v to 1, sequentially test edges to find three representatives of different RH-cycles and finally to apply the instruction to change the local orientation.

The simplified description of the rule *Merge3+* applied in a vertex v follows (for illustration see Figure 3):

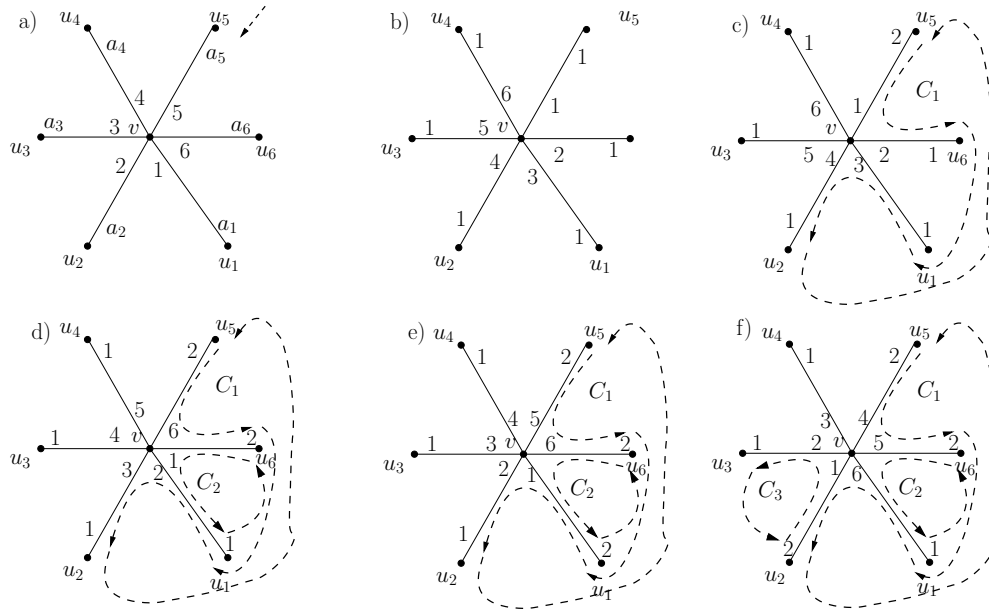


Fig. 3. a) initial state, edge e_5 used to enter v b) outer labels set to 1 c) witness cycle C_1 denoted by outer label 2 d) edge e_6 tested, a disjoint RH-cycle C_2 found e) edge e_1 tested, the cycle is found to be C_1 , thus e_1 will be unmarked now f) edge e_2 tested, RH-cycle C_3 found, rule *Merge3* can be applied to merge the three cycles

- Mark the incoming edge e with inner label 1.
- Sequentially traverse all incident edges of v and set their outer label to 1.
- Set outer label of e to 2 (this will remain unchanged during the rule *Merge3+*), denote the RH-cycle represented by incoming arc of e as C_1 .
- Sequentially process incident edges of v starting with the edge $\text{succ}_v(e)$. While doing this, rotate local orientation so that the processed edge is always marked by inner label 1. Find the first edge e' such that its incoming arc represents a RH-cycle C_2 different from C_1 and mark it by outer label 2.
- Continue in the process of marking the processed edge by inner label 1 with the edge $\text{succ}_v(e')$ and find the first edge e'' such that its incoming arc represents a RH-cycle C_3 different from C_1 and C_2 . Mark it by outer label 2.
- If these three edges are found, the call of instruction for changing the local orientation is made. Otherwise we reach an edge with outer label 2 (edge e). In this case, the agent found out that three disjoint cycles are not present.

4.4 Changing the rule *EatSmall* to *EatSmall+*

The change of rule *EatSmall* to *EatSmall+* is similar to the change in Subsection 4.3. The idea is same as the one in the previous subsection: we split edges into two partitions – those that will be used in the rule *EatSmall* (with outer label 2) and the rest (with outer label 1) and apply the corresponding primitive. Figure 4 illustrates the application of the rule *EatSmall+*.

4.5 Summary

In this section we will summarize the modifications of the algorithm *MergeCycles* to the algorithm with constant memory for the agent – *MergeCycles+*.

Until now, we assumed that the vertices of graph G have degrees greater than one. We will now explain how to handle vertices with degree one. When we start dealing with vertex v , we first need to process neighbouring vertices with degree 1 and then we can ignore them in the applications of the rules *Merge3* and *EatSmall*. In the case when a neighbouring vertex u of vertex v has degree 1 and the edge (u, v) is not the edge we just used to enter v , we check whether u is present on the witness cycle and if not, we add it by using the rule *EatSmall*. In the case when edge $e = (u, v)$ is the edge via which agent entered v , either the agent already visited more than one vertex (and thus already visited v and processed it at that time, see Lemma 4), or we can find the first incident edge of v with degree greater than one and use it as the incoming one. The only case when such an edge does not exist is if the topology of the graph is a star. This can be easily checked in the beginning of the entire algorithm.

By Lemma 4 and the fact that in *MergeCycles+* the RH-cycles are added to the witness cycle so that they are traversed in the first RH-traversal of the formed witness cycle, it is clear that to build the whole witness cycle the preprocessing agent only has to RH-traverse the witness cycle once. After the agent returns to the initial arc where the algorithm *MergeCycles+* begun, no rule can be applied in its future RH-traversal.

Algorithm 1 Algorithm *MergeCycles+*

1: check whether the graph is a star, if so, finish the algorithm, any local orientation forms the witness cycle
2: start the RH-traversal via edge with label 1
3: **procedure** RH-TRAVERSAL(v, e) // agent entered vertex v via $e = (u, v)$
4: **if** vertex u has degree 1 **then**
5: **if** the agent already visited two or more vertices **then**
6: continue with RH-traversal via edge $\text{succ}_v(e)$
7: **else**
8: execute this function with $e = \text{pred}_v(e)$
9: **end if**
10: **else**
11: rotate local orientation so that $\pi_v(e) = 1$
12: **for all** neighbouring vertices w **do**
13: set the outer label of edge (v, w) to 1
14: **end for**
15: **for all** neighbouring vertices w **do**
16: **if** $d_w = 1$ and w is not on the witness cycle **then**
17: apply *EatSmall* to add w to witness cycle
18: **end if**
19: **end for**
20: **while** it is possible **do**
21: ignoring neighbours with degree 1, apply *Merge3+* and *EatSmall+*
22: **end while**
23: continue the RH-traversal via $\text{succ}_v(e)$
24: **end if**
25: **end procedure**

The time complexity of this algorithm is polynomial. The agent doing the precomputations needs one pebble and $O(1)$ memory. The local orientation is used to store a few bits of information, at most $O(\log N)$ at any time.

5 Conclusions, Open Problems, and Further Research

We designed an algorithm that can create a right-hand rule cyclic walk of length $O(|E|)$ that contains all vertices of the given connected simple undirected graph. This goal is achieved by changes in the local orientations. The algorithm is performed by an agent that only uses $O(\log |V|)$ memory and a single pebble. We have discussed properties of this algorithm and we have shown how it is possible to further decrease the memory requirements of the agent. We believe that the main contribution of this paper is showing that local orientations can be used for storing information and decreasing the agent's memory requirements.

The termination detection in algorithm *MergeCycles+* remains an open problem. Further research can be focused on the amount of the memory that is needed to create a witness cycle of length $O(|V|)$. Another interesting question that needs further research is the amount of useful information that can be stored at once in the local orientations during the execution of an algorithm.

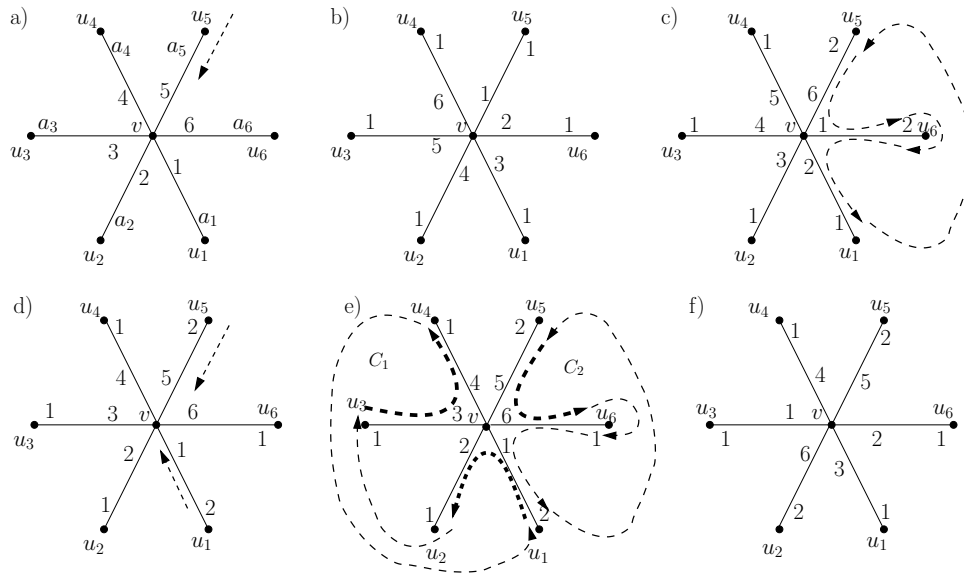


Fig. 4. a) initial state, edge e_5 is the incoming edge b) outer labels are set to 1 c) edges e_5 and e_6 are marked by outer label 2 and their RH-cycles tested for the rule *EatSmall*, answer is negative d) edge e_6 is unmarked (outer label set to 1), edge e_1 is processed now e) the answer is positive – two RH-cycles that are needed in the rule *EatSmall* were found and the rule *EatSmall* can be applied f) after the application of the rule *EatSmall*

6 Acknowledgements

I would like to thank Michal Forišek for many valuable notes, helpful comments and detailed suggestions during the preparation of this manuscript, to Ján Oravec for fruitful discussions and to Rastislav Kralovič for introducing the topic and guiding me in my Master Thesis.

References

1. S. Albers and M. R. Henzinger. Exploring unknown environments. In *STOC*, pages 416–425, 1997.
2. B. Awerbuch, M. Betke, R. L. Rivest, and M. Singh. Piecemeal graph exploration by a mobile robot. *Information and Computation*, 152(2):155–172, 1999.
3. M. A. Bender, A. Fernández, D. Ron, A. Sahai, and S. Vadhan. The power of a pebble: exploring and mapping directed graphs. In *STOC '98: Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 269–278, New York, NY, USA, 1998. ACM.
4. L. Budach. Automata and labyrinths. *Math. Nachr.*, 86:195–282, 1978.
5. X. Deng and C. H. Papadimitriou. Exploring an unknown graph. *J. Graph Theory*, 32(3):265–297, 1999.
6. A. Dessmark and A. Pelc. Optimal graph exploration without good maps. *Theor. Comput. Sci.*, 326(1-3):343–362, 2004.
7. S. Dobrev, J. Jansson, K. Sadakane, and W.-K. Sung. Finding short right-hand-on-the-wall walks in graphs. In *SIROCCO*, pages 127–139, 2005.
8. P. Fraigniaud and D. Ilcinkas. Digraphs exploration with little memory. In *STACS*, pages 246–257, 2004.
9. P. Fraigniaud, D. Ilcinkas, G. Peer, A. Pelc, and D. Peleg. Graph exploration by a finite automaton. *Theor. Comput. Sci.*, 345(2-3):331–344, 2005.
10. P. Fraigniaud, D. Ilcinkas, S. Rajsbaum, and S. Tixeuil. Space lower bounds for graph exploration via reduced automata. In *SIROCCO*, pages 140–154, 2005.
11. L. Gąsieniec, R. Klasing, R. Martin, A. Navarra, and X. Zhang. Fast periodic graph exploration with constant memory. In *SIROCCO*, pages 26–40, 2007.
12. D. Ilcinkas. Setting port numbers for fast graph exploration. In *SIROCCO*, pages 59–69, 2006.
13. P. Panaite and A. Pelc. Exploring unknown undirected graphs. In *SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 316–322, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.
14. P. Panaite and A. Pelc. Impact of topographic information on graph exploration efficiency. *Networks*, 36(2):96–103, 2000.
15. N. Rao, S. Karetí, W. Shi, and S. Iyengar. Robot navigation in unknown terrains: Introductory survey of non-heuristic algorithms, 1993.
16. H. A. Rollik. Automaten in planaren graphen. In *Proceedings of the 4th GI-Conference on Theoretical Computer Science*, pages 266–275, London, UK, 1979. Springer-Verlag.