Department of Computer Science
Faculty of Mathematics, Physics and Informatics
Comenius University, Bratislava

On the power of local orientations

(Master Thesis)

Monika Steinová

Advisor:
doc. RNDr. Rastislav Královič, Ph.D.

Bratislava, 2007

I hereby declare that I wrote this thesis by myself,
only with the help of the reference literature, under
the careful supervision of my thesis advisor.

. . . .

Acknowledgements

I would like to thank to my advisor doc. Rastislav Královič for his guidance and many useful suggestions during my work on this thesis.

I would like to express my gratitude to my family for the all the support I received from them and for their willingness to help.

A part of my gratefulness belongs also to my friends for having and feeling their presence.

My thanks go especially to my friends Michal Forišek and Peter Košinár for their encouragement and number of advice.

Abstract

We consider a network represented by a simple connected undirected graph with N vertices. The nodes of the graph do not have any special unique identifiers, the only requirement is that each incident edge of a vertex v of degree d_v has assigned a locally-unique label (portname) between 1 and d_v, inclusive. This is called a local orientation.

Our main goal is to have an agent visit all vertices of the graph, with the focus on the simplicity of the agent in terms of memory usage. We show that after a suitable pre-processing phase in which we alter the local orientations, the graph can be traversed using the simplest possible agent – an right-hand-rule using agent (RH-agent). Formally, this is an extremely simple finite automaton that always obeys the following rule: "*Start by taking the edge with the label 1. Then, whenever you enter a node, continue by taking the successor edge (in local orientation) to the edge you arrived through.*".

For the above mentioned pre-processing phase we design an algorithm for an agent that performs the precomputation. The agent will be able to alter the network by modifying the local orientations using a simple operation of exchanging two local labels in one step. The goal of this precomputation is to change the local orientations in such a way that the RH-agent's algorithm will cause the RH-agent to visit all the nodes. We show a polynomial-time algorithm for this precomputation that needs only one pebble and $O(\log N)$ memory in the agent. Furthermore, we also show a modification of this algorithm that needs one pebble and only constant memory for the precomputing agent. In the second case, the termination detection is not solved yet, so one extra bit of information in the initial vertex is needed for terminating the precomputation.

As an example of the use of our algorithm, we introduce the problem of building a spanning tree of the graph. More precisely, we present an algorithm for an agent that will construct a rooted spanning tree. This spanning tree will be constructed by altering the local orientations: in any non-root vertex the edge with local label 1 will lead to the parent of this vertex in the spanning tree. Our algorithm only needs to use one pebble and $O(\log N)$ memory in the agent that performs the computation.

KEYWORDS: Mobile computing, distributed algorithms, changing of labeling

Contents

Chapter 1

Introduction

Robots and their behaviour in various conditions were subject to a thorough study in previous decades. The most widely used environment settings were the geometric setting and the graph setting, in which the environment is modeled as a graph with moves restricted to its edges.

Many fields, such as networking and artificial intelligence are concerned with mobile computation in a different way and approach it in a different manner. On one side of the spectrum, mobile computation is interested in the processes of physical robots – mobile devices with an interface for communicating with the environment and other robots. On the opposite side of the spectrum, mobile communication works with agents, virtual processes in a network that can migrate from one machine to another in order to satisfy requests made by their clients. In the middle between these two, there is the theoretical level of the subject which models concrete situations for physical robots and/or agents and tries to solve some common problems.

The focus of this master thesis is this theoretical approach. A model of the environment and of objects working in it will be introduced. Then, a few concrete problems will be defined and a solution will be found to each, such that it will be applicable in the physical environment or in the environment of operation system.

1.1 The Model

There are several different models of distributed information processing used to study distributed algorithms. The choice of a particular model depends on the type of algorithm present.

There are many different computer systems and thus the model needs to be designed in such a way that it is applicable to a class of related systems sharing the basic properties that make them distributed. The model must describe precisely and concisely the relevant aspects of the class.

At the abstract level, a distributed mobile environment can be described as a collection of autonomous mobile entities located in the space. The entities have computing capabilities and use identical programs, thus they behave identically and appear to be same. Depending on the context, the entities are called *robots* at one time and *agents* at other. In this text, the terms *robot* and *agent* will be used interchangeably.

Mobile computing is concerned with determining what tasks can be performed by these entities, under what conditions and what are the minimal costs for obtaining the results.

1.1.1 Motivation

The field of cooperative autonomous mobile robotics is a quite new topic. Some principal topic areas that have generated significant interests of study are summarized according to Parker [Par00].

1. *Biological inspirations:* The introduction of robotics paradigm of behaviour-based control [Ark90, Bro90] has had a strong influence in cooperative mobile robotics research. By the biological inspirations, the results of examination of social characteristics of insects and animals was applied to the design of multi-robot systems. The common way of application of the knowledge of animal behaviour is to introduce simple local control rules inspired by biological societies (e.g., ants, bees, birds, ...) in order to develop similar behaviour. Work in this manner has demonstrated the ability for multi-robot teams to flock, disperse, aggregate, ...

2. *Communication:* Researchers have studied the effect of communication on the performance in a variety of tasks for a multi-robot system. They concluded that communication provides certain benefit for particular types of tasks and found that in many cases, communication of even a small amount of information can lead to a great advantage. Additionally, the work was extended to achieve fault tolerance in communication, such as setting up and maintaining distributed communication networks and ensuring reliability of the nodes of the network. Currently, the multi-robot teams operations in a faulty communication environments are investigated.

3. *Task planning, control and architectures:* This research area addresses topics like action selection, delegation of authority and control, achieving local actions, resolution of conflicts, etc. The general question in this topic is whether specialized architectures for each type of robot team and application domain are needed or whether a more general architecture can be developed that easily fits a wider range of multi-robot systems.

4. *Localization, exploration and mapping:* This topic was extensively researched for single autonomous robot. Large number of algorithms for a multi-robot team was developed from an existing solution for a single robot and extended to multiple robots. These problems are studied in two models of terrains – plane and graphs. In the plane model, the multi-robot teams use landmarks, scan-matching or some range or vision sensors for communication. In the graph model, the communication is usually done by whiteboards – the space in nodes of the network where the information can be written and subsequently read or modified.

5. *Object transport and manipulation:* This topic deals with multiple robot cooperation in carrying, pushing and manipulating object. Numerous variations of this task have been studied, including constrained and unconstrained motions. As the extension of this problem, manipulation with object by extra items such as ropes, etc. was studied. Most of the cooperative movement is done in a flat surface an the challenging issue is this area of cooperative transport over outdoor terrains.

6. *Motion coordination:* The research themes in this domain that have been studied include multi-robot path planning, traffic control, formation generation and keeping. These works have been aimed at 2D and 3D environments. One of the most limiting characteristics in this topic is the computational complexity of the approaches. There is a limiting factor of applicability of some approaches in dynamic, real-time robot teams.

7. *Learning:* Many robot researchers believe in big potential for the development of cooperative control mechanisms in autonomous learning. Challenging domains in this learning are cooperative tasks where certain actions of one robot depend on the current actions of another one. These tasks can not be decomposed into independent subtasks that can be solved by distributed robot team. The success of the team is measured by the combined actions of team members rather than the individual robot actions.

8. *Software mobility:* In this topic, the agents are virtual entities such as processes that are able to migrate in the system to satisfy the requests made by their clients. One idea is to execute a mobile agent in a machine and when the required resources are not provided, the mobile agent is able to save its state, transfer to a machine containing the necessary resources and resume the work. Mobile agents have been developed as an extension to the client-server model. The limitation of the client-server model is that the client is limited to the operations provided by the server. Thus client must find a server that satisfies the request by sending out messages to all servers, which is clearly inefficient and limits network scalability because of managing and updating issues.

1.1.2 The model of agent and environment

It is very common to represent the agents in the theoretical field by finite automata. Such an agent has a finite number of states that can be used as its constant memory. Likewise, the environment is described by a finite number of states which characterize it completely. Then, using the current state of the agent and the knowledge of the state of environment, the next state and an action possibly taking place is obtained.

When the agent needs to remember more than a constant amount of information, the memory can be extended. The agent is then able to perform computations in its memory, change its state and/or move in the environment accordingly.

There are two approaches to modeling the environment: the geometric, where the environment is modeled as a plane and the graph-based one, in which the graph consists of locations interconnected with lines representing the only possible ways how to move between the locations.

In our case, the environment is modeled as a network – a graph with a number of nodes. If the agent is able to traverse directly from node p to a node q, the *channel* or *link*[1] is said to exist between them. This channel can be *bidirectional* – the agent can traverse in both directions – or *unidirectional* – the agent can traverse only in one direction.

In many algorithms, some extra memory is required in the environment. This assumption is stronger, as it can store data related to the location. Storing information in the graph can be realized by "whiteboards" – a space in nodes that can be used for writing, reading and modifying. Another very popular method of information holding is a *pebble*. We can imagine it to be a device that the robot places in a vertex of the graph, removes it and carries it around while it works. It represents one bit of information that can be stored in an arbitrary node of the graph. It is often used to identify a particular vertex and distinguish it from the others. As will be shown later on, there are some problems that are unsolvable without a pebble.

Using a pebble in the algorithm allows for lower memory complexity than using a whiteboard. The whiteboard is a storing place in a vertex where at least one bit of information can be written. Unlike the pebble, the whiteboards allow each vertex to hold its own bits of information simultaneously, whereas the pebble can be stored only in one node at the same time.

Network topology

Since the network topology has a major influence on the nonexistence of algorithm in the particular conditions, a brief overview of the commonly used topological models follows (see Figure 1.1.2).

Ring – A network of nodes that are connected into one cycle

Tree – A network of nodes that are connected without forming any cycles

Star – A network with a central node (later called center of the star) that is connected to all the other nodes, with no other connection present

Clique – A network in which every two nodes have a direct connection

Hypercube – A network representing an n-dimensional cube. An n-dimensional hypercube is a graph (V, E) with 2^n vertices, where $V = \{(b_0, b_1, \ldots, b_{n-1}) \ : \ b_i \in \{0, 1\}\}$ and there is an edge between two vertices b and c if and only if the bit strings b and c differ in exactly one bit.

The topology can be *static* or *dynamic*. Static topology remains fixed during the whole computation. In dynamic topology, channels (sometimes also nodes) can be added or removed during the computation in process.

[1] In this text, we will use these terms interchangeably.

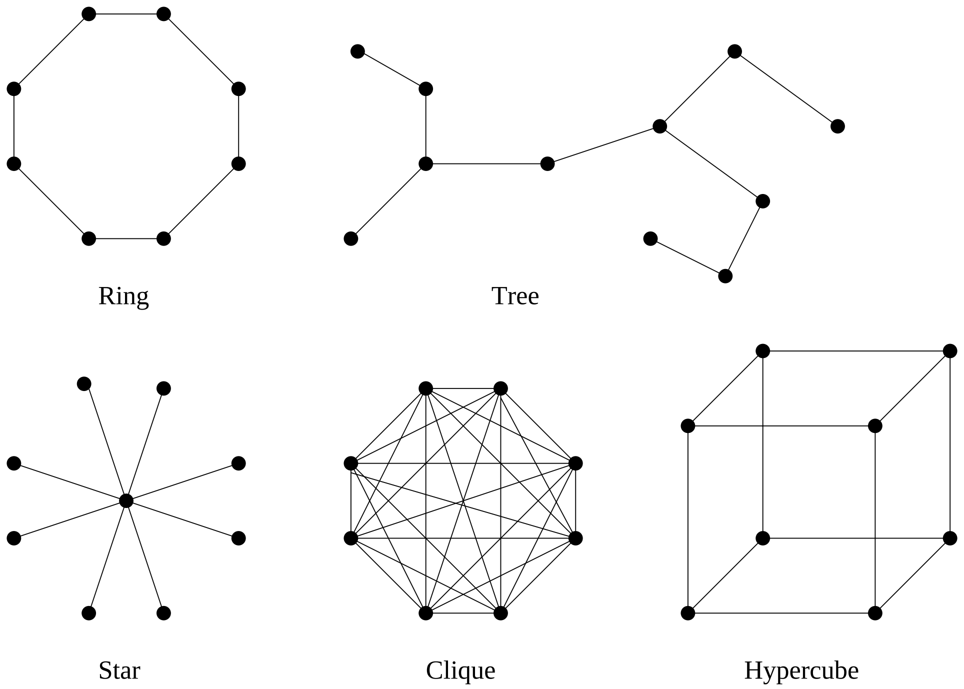

Figure 1.1: Examples of different network topologies

Link, node and agent properties

Some commonly made assumptions about the link, node and agent properties are:

- *Reliability.* A link is said to be reliable if an agent traversing via it is not lost and appears in the destination in a finite time. A node is reliable if the agent is not lost or damaged there.

- *The FIFO property.* The channel is said to be FIFO if it respects the order of the agents traversing via it. When FIFO channels are used a possibility of another communication failure – reordering of agents in a channel – arises. The FIFO property can be added to channels by adding an extra sequence number to the agent while traversing via the particular channels.

- *Channel capacity.* The channel capacity is the number of agents that can be in the channel transit at the same time. The agent can enter the channel only as long as it is not full.

Agent's prior knowledge

In some cases initial information is needed in algorithms. An example of such knowledge includes following information:

1. *Topological information:* This includes information about the number of nodes in the network, the number of links in the network, the number of the agents, network diameter, etc.

2. *Agent identity:* The assignment of a unique name to each agent might be required. This is used in the leader election problem described in section 1.1.4.

3. *Node identity:* A unique name for each node in the network is required in many algorithms.

4. *Neighbor identities:* If nodes are distinguished by a unique name, it is possible to assume that each node knows initially the names of its neighbors. This is useful for addressing. If a set of unique, distinguishable names was not assessed to the nodes, a demand might still arise for a distinction between channels incident with the same node. In this case, the unique names are assigned ad hoc to the particular channel as if from the "viewpoint" of the node. This is called *local orientation* and is discussed in the next sections.

The manner of communication between agents

Communication between the agents is usually achieved through writing of signs on the whiteboards, i.e. local storages the agents are capable of manipulating with. There is one whiteboard per node and the access to it is usually done on the principle of mutual exclusion

1.1.3 Timing in models

One of the most fundamental aspects of distributed systems and mobile computing is the distinction between the asynchronous and the synchronous formal models:

1. *Synchronous model.* This is the simplest model. We assume that the agents' moves are done simultaneously in synchronous rounds. More precisely, there is a prior knowledge of the upper and the lower bounds for the time which is needed for the execution of the steps and also for one traverse via the link in such a model. From these upper limits, the length of one synchronous round can be determined. In systems where the synchronization is not natural, a clock has to be used. This can lead to a problem of maintaining the correct time in all agents of the system. In spite of this complication, it is sometimes possible to "simulate" a synchronous system using an asynchronous one.

   The important property of a synchronous model that needs to be mentioned is that if a problem cannot be solved in the synchronous model it cannot be carried over to a weaker model.

2. *Asynchronous model.* In the asynchronous model, all communication steps can require arbitrary amount of time. Since the asynchronous model is not concerned with

time to the extent of synchronous model, algorithms designed for the asynchronous models are more general and portable. As such, the asynchronous model is more general and all proofs of its properties also apply to the synchronous model but not vice versa. On the other hand, the asynchronous model might not provide enough power to solve problems efficiently, or even to solve them at all.

1.1.4 Main algorithmic problems

There are several primitive problems that the algorithmic mobile computing deals with. Some of them are interrelated and thus solving one of the problems simplifies solving the related ones. The basic problems studied are:

1. *Leader election* is a process when a group of autonomous mobile entities agrees on the selection of one of them as the leader.

2. *Exploration and mapping:* The task is to explore the environment – the graph or the plane – and eventually map it. This topic is investigated in next section.

3. *Termination detection:* This problem is related to terminating the algorithm for an agent and the manner of broadcasting this information to other agents in the network.

4. *Rendezvous* refers to arranging a meeting of two mobile agents situated in different nodes in same node of the graph.

5. *Gathering:* The objective is similar to the Rendezvous problem, the only difference being that more than two agents meet in the same node.

6. *Symmetry breaking:* As agents usually run similar programs and neither the environment nor the agents have to have unique identifiers, more agents can appear in the same situation and, as a result, a method for breaking the symmetry needs to be applied.

1.2 Overview of graph explorations

The task of visiting all nodes of a network is a fundamental part of many algorithms (e.g. searching for data stored in unknown nodes of network). At the same time, traversing all edges might be crucial for network maintenance and checking for defective components.

If the edges do not have any labels, the problem of deterministic exploration is unsolvable even for a 3-node triangle graph. This is so, because after visiting the second node the robot cannot distinguish the edge leading to the node not visited yet. Therefore, names need to be assigned to all edges. This is often called *local orientation* and it is formally defined in Section 2.1. The names of the edges are often called *port labels* or *port names* and it is necessary that they are unique from the perspective of the corresponding vertex (they do not have to be unique in the context of the whole graph).

The exploration of graph with nodes having unique labels can be easily performed by a depth-first search. However, in some cases such a unique labeling may not be available in unknown environment. Hence, the ability to explore anonymous graphs, i.e. graphs without the labeling of nodes or edges, is most wanted.

The restrictions placed on the graph are very important – if they are too weak, some problems might become unsolvable. For example, if we consider an anonymous graph with local orientation and forbid marking of the nodes, it is impossible to explore even a cycle of unknown length or to stop in it. Hence, some marking of nodes (e.g. by dropping and removing pebble) must be available in order to make the problem solvable.

A different approach to the problem requires it to be solved only in a class of graphs with specific properties, such as graphs without cycle. A similar exploration problem was solved for a tree graph with no marking, proving it is indeed possible for anonymous graphs [DFKP02].

The graph exploration was studied further on while taking various factors into consideration. Betke et al. [BRS94] and Awerbuch et al. [ABRS99] studied the problem of exploring an undirected graph with the additional requirement of the robot returning to its starting position every so often. This approach models the situation of refueling.

Bender and Slonin [BS94] showed how two cooperating robots can map a directed graph with indistinguishable nodes, in which each node has the same number of outgoing edges.

Theoretical studies of exploration and navigation problem in an unknown environment were first studied by Papadimitriou and Yannakakis [PY91]. They considered the problem of finding the shortest path from the point s to the point t in unknown environment and came up with many geometric and graph-based variants of it.

S. Alberts and M. R. Henzinger [AH97] considered exploration problems in which the robot has to construct a complete map of an unknown, directed, strongly connected graph. The goal of the robot was to visit all nodes and edges of the graph using the minimum number of edge traversals R. Koutsoupias [Kou] gave a lower bound $\Theta(d^2m)$ on R and Deng and Papadimitriou [DP90] showed an upper bound $d^{O(d)}m$, where m is the number of edges in the graph and d is *deficiency* – the minimum number of edges which have to be added to make the graph Eulerian. They offered the first sub-exponential algorithm for such an exploration problem with the upper bound of $d^{O(\log d)}m$.

Yet different approach to graph exploration was shown by Stephen Kwek[Kwe97]. The space here is represented by a strongly connected directed multi-graph G in which the robot has to explore vertices and edges by traversing them. The robot knows all the traversed edges and visited nodes and can recognize them when he enters them again. However, the robot does not know the number of vertices and edges in the graph, where the unseen edge leads to, nor the origin point of any unseen edge of a visited node. The algorithm presented in the paper is a depth-first search strategy for exploring graph G with m edges and n vertices by traversing $\min(mn, dn^2 + m)$ edges at most, where d is the deficiency of the graph G.

Bender et al.[BFR$^+$98] came up with some interesting results. Authors model the environment as an unknown, strongly connected directed graph G and consider the problem of the robot exploration of G. They do not assume that vertices are labeled and thus the robot cannot distinguish them. For this reason, robot is provided with a "pebble" – a device it places in a vertex and uses it to identify the vertex later on. Two deterministic algorithms are designed in the paper with following conclusion:

1. when the robot knows the upper bound on the number of vertices, it can explore graph efficiently with one pebble,
2. when the robot does not know the upper bound on the number of vertices n then $\Theta(\log\log n)$ pebbles are both necessary and sufficient.

The exploration with the team of non-cooperative robots was studied by Fraigniaud et al. in [FIRT05]. The paper is interested in memory requirements of a team of robots for the graph exploration. First, they show that for any set of q non-cooperative K-state robots, there exists a graph of size $O(qK)$ that no robot of this set can explore (no robot can visit each vertex and traverse each edge at least once). Then, they deal with exploration with one robot stopping at its end. For this task, the robot is provided with a pebble. They prove that the exploration with a stop requires $\Omega(\log n)$ bits for an n-node graph. At the same time, they prove that there exists an exploration with a stop-algorithm using a robot with $O(D \log \Delta)$ bits of memory to explore all graphs of diameter at most D and the maximal degree at most Δ.

Cohen et al. [CFI$^+$05] investigates labeling of vertices. It presupposes that the designer of the system is allowed to add short labels to the graph nodes in a pre-processing stage. These labels are then used to guide the robot in an exploration. They give an exploring algorithm, which placing 2-bit labels into vertices, allows the robot to explore the whole graph. Furthermore, they describe a suitable labeling algorithm for generating the labels required in a linear time. They show how to modify the labeling scheme for the graphs of a bounded degree using 1-bit labels so that robot is still able to explore it completely. Finally, they bring a proof of a negative result of a graph exploration by a robot with no internal memory (i.e. a single state automaton).

Another investigation of labeling is done in Dobrev et al., Ilcinkas, Gąsieniec et al. [DJSS05, Ilc06, GKM$^+$07]. In all three papers, the changes in edge-labeling are done as a pre-computation in order to obtain a graph with certain good properties, such that a simple traverse algorithm for visiting all nodes can be designed. By Dobrev et al. [DJSS05] the problem of perpetual traversal (assuming that the robot visits every node infinitely many times in a periodic manner) by a single agent in an anonymous undirected graph $G = (V, E)$ is approached. The following traversal algorithm 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 orientation) to the edge via which you have arrived"* and poses a question whether it is possible to assign the local orientation providing that the agent visits every node in $O(|V|)$ moves employing the resulting perpetual traverse. In [Ilc06] the author continues in the study of the problem, changing the approach. The change includes the traversal algorithm and the number of moves in perpetual traversal required for visiting every node being lowered by a constant factor. The most recent publication, [GKM$^+$07], studies the problem of periodic exploration of all nodes in an undirected graph using a finite state automaton (robot) with a decreased upper bound of the solution.

Our goal is derived from the research in [DJSS05, Ilc06, GKM+07]. We are investigating in what way additional information can be stored in the graphs by using local orientation. In Chapter 3 we show the algorithm for constructing a special cycle, in which very simple finite automaton is able to traverse all nodes. In Chapter 4 the algorithm for building a spanning tree is presented. The local orientation is used for marking the edges in a spanning tree. The modification of the algorithm from Chapter 3 to decrease the memory requirements is described in Chapter 5. 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, an extra bits of information stored in the graph can be used to terminate this algorithm. All algorithms are handled with a local approach – they are designed for an agent which has to make changes in local orientation of vertices at minimal memory requirements.

Chapter 2

Notation and preliminaries

2.1 Definitions

In this section we shall provide formal definitions used in the rest of the thesis. We use a terminology similar to that from [DJSS05]. We also expect the reader to have basic knowledge of graph theory (for details see [Die00]).

We model the network in which the agents travel by an undirected connected graph. Vertices correspond to *nodes* of the network and edges to the *links* between pairs of nodes in the network. To be able to distinguish between directed and undirected links, we will call the directed links *arcs* and undirected links *edges*. In the further text, we will denote the degree of a vertex v by d_v, the number of vertices of the graph by $N = |V|$ and the number of its edges by $M = |E|$.

Definition 2.1.1 *Let G be a simple connected undirected graph. Consider a vertex v with degree d_v. Let us denote by π_v a function that assigns to each edge e incident to v a unique label $\pi_v(e) \in \{1, 2, \ldots, d_v\}$. Function π_v is called a* local orientation *in v.*

Note that the existence of 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

$$succ_v(e) = \pi_v(e) \bmod d_v + 1$$

is the successor function and the corresponding predecessor function $pred_v(e)$ is defined similarly.

Definition 2.1.2 *Let G be a simple connected directed graph. We say that a vertex v of graph G is* RH-traversable *(right-hand-traversable) if there exists a local orientation π_v in v such that for each arc e of G incoming to v there exists an outgoing arc e' in G such that $e' = succ_v(e)$. We call such local orientation a witness ordering for v.*

Definition 2.1.3 (Right-hand rule) *Let v be an RH-traversable vertex of a graph G. The rule:* " if the vertex v is entered by the edge e, leave it using an edge with label $succ_v(e)$" *is called the right-hand rule.*

Definition 2.1.4 *Let us consider a graph with a witness ordering in every vertex v. An agent that obeys the following rules in such a graph is called an RH-agent:*

1. *in the initial vertex v start the traversal by following the edge labeled 1*
2. *at each vertex continue the traversal by using the right-hand rule*

The traversal according to these rules is called an RH-traversal.

Informally, the right-hand rule is very similar to the imagination of right-hand rule in a maze. We can imagine vertex v with incident edges as a crossroad where all tunnels are meeting. Then, we can order edges according to their labels from 1 to d_v into a circle. Suppose that the RH-agent entered vertex v via edge with label a, then the edge via it will leave, is the one that is the next on the right in the circle and the corresponding edge label is $succ_v(a) = a \bmod d_v + 1$. This is the same idea as choosing the first tunnel on the right-hand side upon arrival to a crossroad.

We say that vertex is RH-traversable when such movement is possible. RH-traversal is always possible in maze, but in directed graph this does not have to hold.

Careful reader may note that in mazes, choosing arbitrary tunnel and obeying the right-hand rule leads to finding the exit or to returning back to the origin. Very similar property holds in graphs with witness orderings in vertices. This will be later shown as Theorem 2.2.4.

The notion of RH-traversable vertex is defined for easier description of the fact that the right-hand rule can be used in the vertex. Graph where right-hand rule can be applied in each vertex is defined as RH-traversable. Agent that is traversing in RH-traversable graph starting by edge with label 1 and using the right-hand rule is called RH-agent. This agent can be represented as a simple finite automaton (obviously, it cannot terminate the traversal process) and thus it is the simplest structure of agent that exists. Later, we will talk about RH-traversal that is simply the traversal done by an RH-agent.

Definition 2.1.5 *A graph G is called RH-traversable if and only if all vertices are RH-traversable.*

Definition 2.1.6 *(Pebble) Let $G = (V, E)$ be a graph. Pebble is a global variable $p \in V \cup \{\perp\}$ such that $p = \perp$ in the beginning. Only an agent located in vertex $v \in V$ can manipulate with p and only the following operations are permitted:*

- *if $p = \perp$ then set $p = v$*
- *if $p = v$ then set $p = \perp$*
- *test whether $p = \perp$*
- *test whether $p = v$*

Informally, the pebble is a device that can be held by the agent and carried while walk. Whenever the agent is carrying pebble and enters a node, agent can put this pebble to the node. Whenever agent comes to a node where pebble is stored, agent can take it and carry it while walking in graph. The pebble is used to distinguish a specific vertex from the others. It is kind of a mark of the vertex that is often used in graph exploration. As we already note in Section 1.1, it is the simplest model that leaves an information in graph. The wide range of studies about using the pebble in graph algorithms was done. The fundamental result was proved in [BFR+98] – even directed strongly connected graph can be traversed by agent with pebble. On the other hand traversal even with higher number of agents, modeled as finite automata, without a pebble, leads to an unsolvable problem ([FIRT05]).

2.2 Basic statements

In this section, we will demonstrate a few basic properties of the presented model.

Lemma 2.2.1 *Let G be a graph and $v \in V$ vertex with witness ordering π_v. Let $E_v = \{e \in E \mid e \text{ is incident with vertex } v\}$. Let $succ_v(e) = \pi_v(e) \bmod d_v + 1,\ e \in E_v$ be the successor function for witness ordering π_v. Then function $succ_v$ is a bijection.*

Proof: Straightforward from the definition of function $succ_v$. □

Lemma 2.2.2 [DJSS05]: *Let $G = (V, E)$ be a simple directed graph. Let $v \in V$. Let i_v be the number of incoming edges to v and o_v be the number of the outgoing edges from v. Then v is RH-traversable if and only if $i_v = o_v$.*

Proof: The if direction: Take all undirected incident edges of v and then alternate outgoing incident edges with incoming incident edges of v. Label these edges by $1, \dots, d_v$ – see Figure 2.1. The result is RH-traversable vertex v.

The only if direction: As v is RH-traversable and by Lemma 2.2.1, the number of incoming and outgoing edges incident with v equals. Moreover the undirected edges are both incoming and outgoing and thus $i_v = o_v$. □

Corollary 2.2.3 *Let $G = (V, E)$ be a simple undirected graph. Denote $E_v = \{e \in E \mid e$ is incident with vertex $v\}$. Let σ_v be an arbitrary ordering of edges incident to vertex v. Formally, let σ_v be a bijection from E_v to $\{1, \dots, d_v\}$. Then σ_v is a witness ordering.*

Note that by the Corollary 2.2.3 an arbitrary labeling of edges from 1 to d_v (the degree of vertex v) in a simple undirected graph is witness ordering of graph and furthermore, the graph is RH-traversable. This is a statement similar to the one about using the right-hand rule in mazes discussed in the paragraph in the previous section.

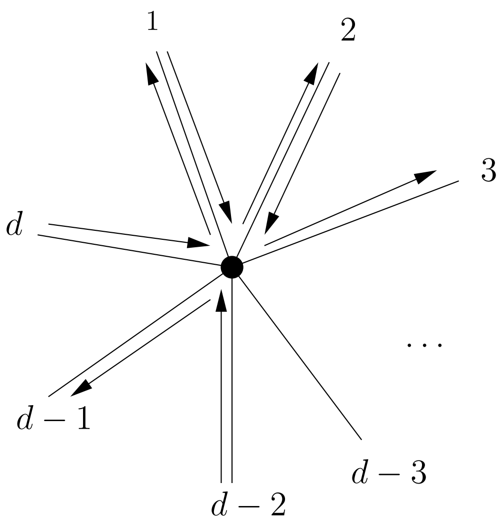

Figure 2.1: Ordering of two bidirectional, two incoming and two outgoing edges.

Theorem 2.2.4 *Let $G = (V, E)$ be a graph with witness ordering π_v for each vertex $v \in V$. Consider the following infinite traversal τ: Start in an arbitrary vertex $v \in V$ and leave v via an arbitrary incident edge e. Then, continue using the right-hand rule forever. The set of vertices from τ then forms a cycle that enters v via the edge with the label $pred_v(e)$.*

Proof: Consider the traversal τ starting from the vertex v via the edge e. Note that at any moment during the traversal, the rest of the traversal is uniquely determined by the last edge we traversed. As the traversal is infinite, sooner or later we have to traverse some edge for the second time. From this point on the traversal will be periodic and thus denote the cycle created this way as P. We claim that P enters vertex v via the edge with the label $pred_v(e)$. [1] Suppose, for the sake of contradiction, that it is not the case. Then, the arc e (outgoing from v) is not a part of P, since it is not traversed infinitely many times. Consider the maximal prefix τ' of τ that is not part of P. Let e_3, leading to some vertex w, be the last arc of τ'. Note that it may be the case that $w = v$. Then P leaves w along the arc $e_2 = succ_w(e_3)$. Consider the arcs along which P enters and leaves w – see Figure 2.2. As P is a cycle, the edge $e_1 = \pi_w^{-1}(pred_w(e_2))$ $(\pi_w^{-1}(succ_w(e_1)) = e_2)$ belonging to the cycle P must exists. By the Lemma 2.2.1 about the bijectivity of function $succ_w$ and from $succ_w(e_3) = \pi_w(e_2) = succ_w(e_1)$ we obtain $e_3 = e_1$ and that is the contradiction with the maximality of prefix τ' of τ. □

Since the local orientations can be rotated arbitrarily, we can assume that the arc with label 1 can be used in the outgoing direction at every vertex.

Definition 2.2.5 *From Theorem 2.2.4 it follows that by starting at an arbitrary vertex v*

[1]Note that P may enter v many times.

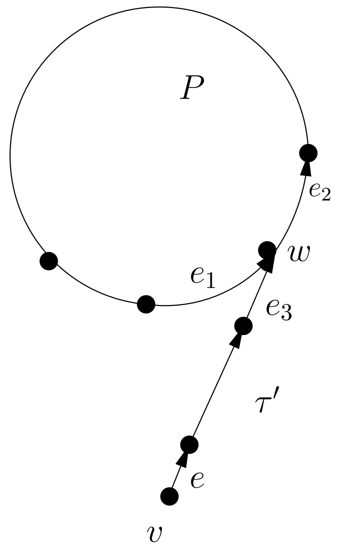

Figure 2.2: The configuration under examination in proof of Theorem 2.2.4.

of graph $G = (V, E)$ with a witness ordering π_u for each vertex $u \in V$ and leaving via the arc with label 1*, the cycle will be RH-traversed. We will call this cycle a witness cycle.*

Theorem 2.2.6 *Let C_1 and C_2 be two witness cycles of graph $G = (V, E)$. Let directed edge $e \in E$ be contained in C_1 and C_2, then $C_1 = C_2$.*

Proof: Straightforward from definition of witness cycle and Lemma 2.2.1. □

Lemma 2.2.7 *Let v be a vertex of a simple connected undirected graph $G = (V, E)$ with witness ordering π_u, $u \in V$. Then degree of vertex v is k if and only if there exist k different arcs $e_1, e_2, \ldots e_k$ oriented toward v and k different arcs $e'_1, e'_2, \ldots, e'_k$ oriented from v such that the undirected edges e_i and e'_i are the same ($1 \leq i \leq k$) and $succ_v(e_1) = \pi_v(e'_2)$, $succ_v(e_2) = \pi_v(e'_3)$, $\ldots$, $succ_v(e_k) = \pi_v(e'_1)$.*

Proof: We will prove the statement by mathematical induction on k, the degree of vertex v:

- *the base of induction $k = 1$:* The statement holds by the definition of the function $succ_v$ as a cyclical ordering of labels of edges and Lemma 2.2.1.

- *the inductive step:* Let the statement hold for the degrees d, $d < k$. Let undirected edge e be incident to the vertex v and let e_i and e'_i be the corresponding arcs ($1 \leq i \leq k$). Let $a = pred_v(e'_i)$ and $b = succ_v(e_i)$. Remove the edge e from graph G and modify $succ_v(\pi_v^{-1}(a)) = b$ and $pred_v(\pi_v^{-1}(b)) = a$ as shown in Figure 2.3. Then by the induction hypothesis the statement holds. Then the edge e can be added to the vertex v between edges with labels $k - 1$ and 1 and the statement will hold again.

□

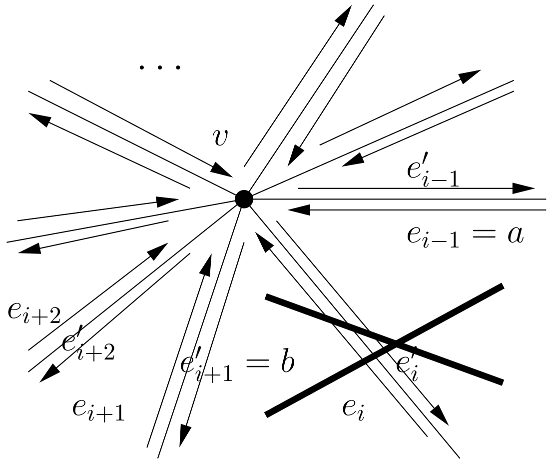

Figure 2.3: Removing edge in vertex v of degree k.

Chapter 3

Algorithm MergeCycles

In this chapter we will discuss the algorithm for constructing a witness cycle that passes through all vertices of a graph. Our goal is to apply the ideas from [DJSS05] in a local manner and to minimize memory requirements.

We will consider the input to be a network modeled as an undirected simple connected graph $G = (V, E)$, along with the labeling of its edges. We suppose that the labels of edges incident to vertex $v \in V$ are $1, 2, \dots d_v$, where d_v is the degree of vertex v.

The desired output of this algorithm is a new labeling of the edges in graph G such that if an RH-agent starts the traversal at any vertex, after a finite number of transits via the links it will traverse all the nodes in the network and return to the starting point of its traversal.

The change of the labels is done by exchanging labels of two edges incident to a particular vertex. This is done in the precomputation phase by a different, more powerful agent. An extra requirement of the algorithm is to do the precomputation of labeling while minimizing the memory requirements. In the further text, we will use two terms

- *RH-agent* will denote an agent using the output of our algorithm and traversing all the vertices of the graph according to the right-hand rule.

- *agent* will denote a more powerful agent which will perform the precomputation according to our algorithm.

Note that, as shown in [BFR$^+$98], at least one pebble is required for graph exploration. Thus a pebble will be necessary in the following algorithm.

3.1 Main idea

Since the input graph is undirected and its ordering is given, by Corollary 2.2.3 the assigned labels of edges form a witness ordering. Then by Theorem 2.2.4 the input graph consists of a several witness cycles. The basic idea of our algorithm is to find a way to merge multiple witness cycles into a single one (we will call such witness cycle *kernel witness cycle* or *kernel* *cycle* in short). This is done by applying rules *Merge3* and *EatSmall* that are explained in the next two sections. Pick an arbitrary vertex v. By the Theorem 2.2.4 the RH-traversal starting in a vertex v and continuing through arbitrary edge (e.g. edge with label 1) is a witness cycle. This is going to be the initial kernel witness cycle that will be enlarged by applying rules *Merge3* and *EatSmall* to its vertices. More precisely, as this cycle is a witness cycle, the agent can simulate the walk of the RH-agent (RH-traverse). While walking along the kernel cycle, our agent will try to apply rules *Merge3* and *EatSmall* in each visited vertex.

If any of the rules given above is applied, the kernel witness cycle that is being built is enlarged. If none of the rules can be applied in a given vertex, the algorithm continues in the RH-traversal to the next vertex, trying to apply one of the rules there.

The algorithm terminates when an agent RH-traverses the whole kernel cycle and neither *Merge3* nor *EatSmall* can be applied in any vertex. This is checked easily by counting the number of visited vertices in a row, in which neither of the rules could be applied. This can be further compared with the length of the kernel cycle.

Finally, the kernel cycle is RH-traversed for the last time and in each visited vertex the label 1 is assigned to some outgoing arc that belongs to the kernel cycle.

3.2 Rule Merge3

Rule Merge3: [DJSS05] Let v be a node incident to at least three different witness cycles C_1, C_2 and C_3. Let x_1, x_2 and x_3 be edges incident to v containing incoming arcs for witness cycles C_1, C_2 and C_3, respectively. Consider the ordering of the edges in v, which makes the successor of x_2 become the successor of x_1, the successor of x_3 become the successor of x_2, successor of x_1 become successor of x_3, and the relative order of the remaining edges remains the same (see Figure 3.1). This new ordering connects the witness cycles C_1, C_2 and C_3 into one witness cycle, while remaining a witness ordering for v (because the original ordering was).

Now we show the algorithm for finding three different witness cycles that are incident to vertex v in a local manner. The agent remembers the labels of three arcs leaving vertex v, e.g. e_1, e_2, e_3. By Theorem 2.2.6, an arc is a representative of a witness cycle and therefore e_1, e_2, e_3 represent three possibly different witness cycles C_1, C_2 and C_3. For a special purpose, there is a need to remember one extra arc a initialized as $a = succ_v(e_3)$.

The implementation of the rule *Merge3* in local manner follows: First of all, our agent marks the vertex v by inserting a pebble (this pebble serves to mark the vertex where the merge of witness cycles is going to be made). The agent sets the initial values of e_1, e_2 and e_3 to edges with labels $1, 2$ and 3 respectively. Then it RH-traverses the whole witness cycle C_1 starting via the edge e_1. The end of the traversal can be found in vertex v, where the successive edge of incoming edge to v is e_1. During this RH-traversal, the agent checks whether arcs e_2 and e_3 are present in the cycle C_1. After the RH-traversal, it is known whether the cycle C_1 is different from the cycles C_2 and C_3. Similar check can also be made for the cycle C_2 and cycle C_3. Whenever two witness cycles C_i and C_j are the same

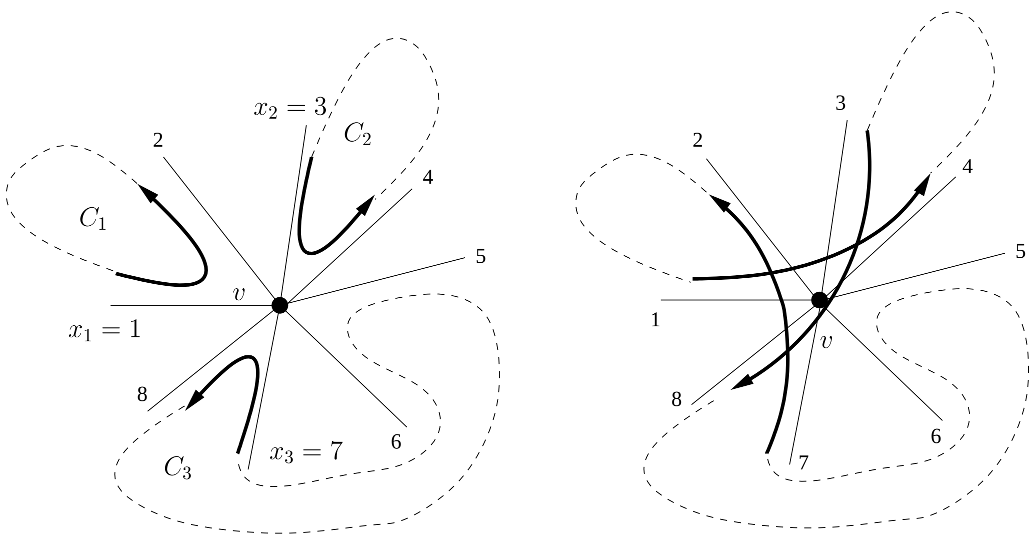

Figure 3.1: Applying rule *Merge3*

$(i < j)$, the agent sets $e_j = \pi_v^{-1}(a)$ and $a = succ_v(\pi_v^{-1}(a))$. If $\pi_v(e_j) \neq 1$ the algorithm starts new check for difference of cycles C_1, C_2 and C_3 represented by arcs e_1, e_2 and e_3.

This algorithm terminates in the following two cases:

- Three different witness cycles are found. Edges $pred_v(e_1)$, $pred_v(e_2)$ and $pred_v(e_3)$ correspond to edges x_1, x_2 and x_3. Then the *Merge3* rule can be applied as described in the first paragraph and a single witness cycle is made.
- If we get to the situation where e_j $(1 \leq j \leq 3)$ is assigned the value 1 again, we have tried all incident edges of v as the edges of the witness cycle and we have not found three different cycles and thus this rule cannot be applied here.

Finally, the pebble needs to be removed from vertex v.

Lemma 3.2.1 *Rule Merge3 can be applied on any three different witness cycles incident to a vertex.*

Proof: Straightforward from the construction, since there are no constraints. □

Lemma 3.2.2 *The rule Merge3 can only be applied finitely many times. After an application of the rule Merge3 the number of cycles in the graph decreases by two.*

Proof: By the application of rule *Merge3* three witness cycles are merged together and no cycle is separated. Thus the number of witness cycles decreases by two. As the number of the witness cycles is finite the statement holds. □

Lemma 3.2.3 *Let v be a vertex of a simple undirected connected graph G. Let there be at least three different witness cycles passing through the vertex v. Then, these cycles are detected and the rule Merge3 is applied.*

Proof: It is straightforward from the Lemma 3.2.1 and the fact that the value of a variable e_i, $i \in \{1, 2, 3\}$ is changed when it represents the same witness cycle as the variable e_j, $j \in \{1, 2, 3\}$, $i > j$. □

Lemma 3.2.4 *The time complexity of one application of the rule Merge3 in a simple undirected connected graph $G = (V, E)$ is $O(M\Delta)$, where Δ is the maximal degree of a vertex in the graph and $|E| = M$.*

Proof: The length of one cycle that needs to be traversed is $O(M)$. We need to try $O(\Delta)$ edges incident to a particular chosen vertex during the search for three different witness cycles. Thus the total time complexity of one usage of the rule *Merge3* is $O(M\Delta)$. □

3.3 Rule EatSmall

Rule EatSmall: **[DJSS05]** Define a non-simple witness cycle as a witness cycle where a vertex is passed at least twice. Let C_1 be a non-simple witness cycle in this ordering and let v be a vertex appearing in C_1 at least twice. Suppose that v is also incident to a different witness cycle C_2. Let x and y be arbitrary edges containing incoming arc of C_1 and C_2 in v, respectively; let z be the edge containing the incoming arc by which C_1 returns to v after leaving via the successor of x. If z is successor of y, choose a different x. Modify the ordering of the edges in v as follows: (1) the successor of x becomes the new successor of y, (2) the old successor of y becomes the new successor of z, (3) the old successor of z becomes the new successor of x and (4) the order of the remaining edges does not change – see Figure 3.2.

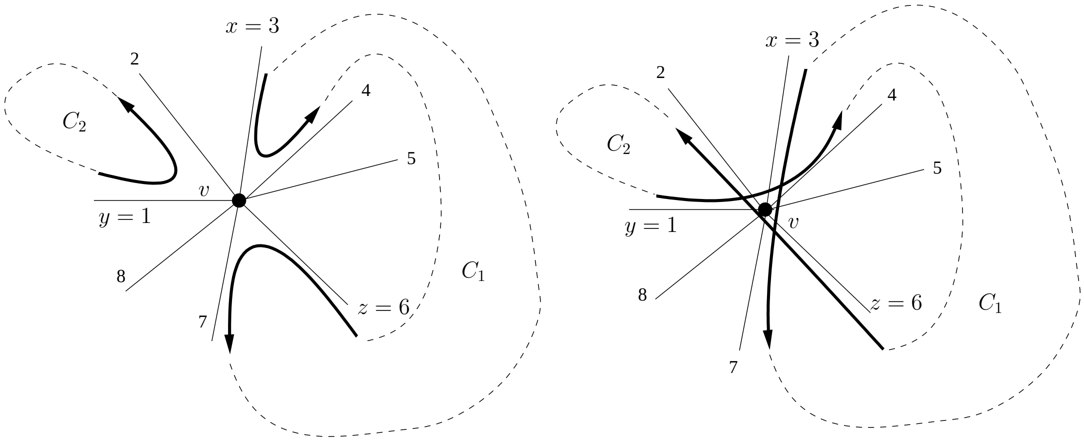

Figure 3.2: Applying rule *EatSmall*

The local version of using rule *EatSmall* in a vertex v is similar to the *Merge3* rule. The agent chooses two different witness cycles C_1 and C_2. Edges x, y and z can be easily found and checked while finding cycles C_1 and C_2.

Note that blindly applying the rule *EatSmall* may lead to deadlocks when a part of a cycle will be transfered back and forth between two cycles. To prevent the deadlocks, we will always use our kernel cycle as the cycle C_2 in our algorithm.

Lemma 3.3.1 *[DJSS05] Applying the rule EatSmall results in a transfer of a loop of the edges from witness cycle C_1 to a different witness cycle C_2, while maintaining RH-traversability.*

Proof: Straightforward from the construction. □

Lemma 3.3.2 *Let $\Gamma(C)$ be the length of the witness cycle C. Suppose that we are only allowed to apply the rule EatSmall if $\Gamma(C_1) \leq \Gamma(C_2)$. Then the rule EatSmall can only be applied finitely many times.*

Proof: By the definition of the ordering Γ, in each application of the rule *EatSmall* the shorter witness cycle is shortened and the longer cycle is extended. When two witness cycles are equal in ordering (they have the same length), by applying rule *EatSmall*, one of these cycles is shortened and one extended. Therefore by application of the rule *EatSmall* the sum of squares of cycle lengths is increased. M^2 is a trivial upper bound on the sum of squares of cycle lengths. Thus there can only be finitely many rule applications. □

Lemma 3.3.3 *Let v be a vertex of a simple undirected connected graph G. Let two different witness cycles C_1 and C_2 pass through vertex v in G. Let the cycle C_1 pass through vertex v at least twice. Then, the rule EatSmall can be applied in vertex v, using cycles C_1 and C_2.*

Proof: Denote by y the incoming arc incident with vertex v that is in the cycle C_2. Denote by x the incoming arc incident with vertex v that is in the cycle C_1 and by z the incoming arc incident with vertex v that is the next incoming arc after z to the vertex v in the cycle C_1 (notation is similar as shown in Figure 3.2). The rule *EatSmall* cannot be applied only if z is a successor of y. In such case we simply take z as the new value of x and then we find the new value of z as the next incoming arc after the arc x that is incident to the vertex v in the cycle C_1. As the cycle C_1 passes through vertex v at least twice, such arcs exists and are different. By the Lemma 2.2.1 and the fact that in this case x is successor of y, z cannot be successor of y and therefore the rule *EatSmall* can be applied. □

Lemma 3.3.4 *The time complexity of one application of the rule EatSmall in a simple undirected connected graph $G = (V, E)$ is $O(M\Delta)$, where Δ is the maximal degree of a vertex in V and $|E| = M$.*

Proof: For a chosen vertex there are $O(\Delta)$ incident edges. Each witness cycle has $O(M)$ edges and thus finding two different witness cycles has time complexity $O(M\Delta)$. Afterwards, the change of the labels in the chosen vertex, can be performed in $O(1)$ and therefore the total time complexity of the rule *EatSmall* is $O(M\Delta)$. □

3.4 Algorithm MergeCycles

Definition 3.4.1 *Pick an arbitrary vertex v. The witness cycle determined by v and the outgoing edge with label 1 will be called the kernel cycle. Our algorithm will extend this cycle by applying local rules that alter the graph. At any moment there will be exactly one kernel cycle.*

Definition 3.4.2 Rule Merge3KC: *The rule Merge3 is applied on three cycles, one of them being the kernel cycle. After the rule is applied, the new combined cycle is the current kernel cycle.*

Definition 3.4.3 Rule EatSmallKC: *The rule EatSmall is applied on two cycles C_1, C_2, with C_2 being the kernel cycle. After the rule is applied, the extended cycle C_2 is the current kernel cycle.*

The pseudocode and the detailed explanation of the *MergeCycles* algorithm follows. By `succ_v(e)` and `pred_v(e)` we will denote the operations $succ_v(e)$ and $pred_v(e)$.

```
function beginRHtraversal from vertex v via edge e
1:  step = 0;
2:  while rule Merge3KC can be applied in v do
3:    apply Merge3KC;
4:  od
5:  apply rule EatSmallKC in vertex v if possible;
6:  if rule EatSmallKC or Merge3KC was applied
7:    goto 2;
8:  else
9:    length = getLength(v,e);
10:   tryToApplyRules via edge e;

function getLength from vertex v via edge e
11: put pebble to vertex v;
12: count = 0;
13: RH-traverse via edge e,
    increase count for each edge traversed,
    stop in v when incoming edge is pred_v(e);
14: take pebble from v;
15: return count;

function tryToApplyRules in vertex v via edge e //e is incoming edge to v
16: step = step + 1;
17: if (step == length)  // kernel cycle passes all vertices
18:   RH-traverse the kernel cycle via edge succ_v(e),
      rotate the local orientation in each visited vertex, such that
```

```
      the outgoing edge will have label 1;
19:   exit;
20: else
21:   while rule Merge3KC can be applied in vertex v do
22:     apply Merge3KC;
23:   od
24:   apply EatSmallKC if it is possible;
25:   if rule EatSmallKC or Merge3KC was applied
26:     beginRHtraversal via edge succ_v(e);
27:   else
28:     tryToApplyRules via edge succ_v(e);

function start in vertex v
29: beginRHtraversal via edge with label 1
```

Now we will discuss the pseudocode in more detail. The idea of the code is that during the RH-traversal of the witness cycle, which is defined by the starting vertex v and edge with label 1, the rules *Merge3KC* and *EatSmallKC* are applied. The application of rules *Merge3KC* and *EatSmallKC* causes that the starting cycle (called kernel cycle) is extended. After the extension, by the resuming of the RH-traversal, the newly connected vertices are visited and the rules *Merge3KC* and *EatSmallKC* are applied there. The algorithm terminates once whole kernel cycle is RH-traversed and no rule is applied. To confirm this, the length of the kernel cycle `length` is computed (the length is recomputed each time after a few applications) and the number of visited vertices `step` in a row, where no rule has been applied, is also calculated.

Before the termination of the algorithm, the resulting kernel cycle is RH-traversed for the last time and the local orientation of each visited vertex is rotated so that the edge with label 1 is one of the outgoing edges of the kernel witness cycle in that vertex. Note that during the RH-traversal only the relative ordering of edges is needed. The particular values of labels are never used except for the initial step of the RH-traversal where the edge with label 1 needs to be present. Therefore, by rotating the local orientation, the ordering of the edges will stay unchanged and the RH-traversal will not be affected.

The starting point of the algorithm *MergeCycles* is the function `start`.

In function `beginRHtraversal`, the RH-traversal of the kernel cycle is launched. In lines `2-7`, rules *Merge3KC* and *EatSmallKC* are applied. When no rule can be applied (line `8`), the length of the current kernel cycle is recomputed and then the next vertex is tested for application of rules *Merge3KC* and *EatSmallKC* (lines `9-10`).

In function `tryToApplyRules`, the test of a vertex for the possibility of application of rules *Merge3KC* and *EatSmallKC* is done. In the beginning, the number of tested vertices is increased, then the check whether the algorithm terminates, is made. For the positive answer (lines `17-18`), the resulting kernel cycle is RH-traversed for the last time and in each visited vertex v the local orientation is rotated, so that the current outgoing edge of the traversal has label 1. Note that after this RH-traversal finishes, in each vertex the edge with label 1 will lie on the kernel cycle. Therefore, our RH-agent starting its traversal from any vertex by using the edge with label 1 will visit all vertices of the graph.

If the answer is negative, an attempt is made to apply rules *Merge3KC* and *EatSmallKC*. When the rule *Merge3KC* or *EatSmallKC* is applied, the length of the new kernel cycle is changed and thus a new RH-traversal begins. If no rule is applied, the RH-traversal continues via the successive edge to test the next vertex (line 28).

Note that when rule *Merge3KC* or *EatSmallKC* is applied in a vertex v, the incoming edge of kernel cycle to the vertex is known and thus the original rules *Merge3* and *EatSmall* can be easily altered so that we know one outgoing edge of the kernel cycle after the rule is applied.

It needs to be noted that the pseudocode for algorithm *MergeCycles* is simplified. Formally, the agent has a $O(\log N)$ possible number of states that need to change while performing algorithm *MergeCycles*.

3.5 Proofs of correctness and complexity

Lemma 3.5.1 *During the execution of the above algorithm the agent traverses the kernel cycle.*

Proof: It is straightforward from the pseudocode and its description. □

Lemma 3.5.2 *The rules Merge3KC and EatSmallKC can be applied only finitely many times in a simple undirected connected graph G, regardless of the order of application.*

Proof: By the statements and definitions from Sections 3.2, 3.3 and 3.4 each application of the rule *Merge3KC* or *EatSmallKC* increases the length of the kernel cycle. As the length of the kernel cycle is finite, the application of the rules is finite and therefore the statement holds. □

Lemma 3.5.3 *Whenever the original rule Merge3 can be applied during the execution of the algorithm MergeCycles, the rule Merge3KC can be applied too.*

Proof: By the Lemma 3.5.1 the agent comes into vertex v by the kernel cycle. Thus the kernel cycle passes through vertex v. As the rule *Merge3* can be applied, at least three different cycles pass through vertex v. Thus we can apply *Merge3* on the kernel cycle and any two other cycles. □

Lemma 3.5.4 *Whenever the rule EatSmall can be applied during the execution of the algorithm MergeCycles and C_1 is not the kernel cycle, the new rule EatSmallKC can be applied too.*

Proof: By the Lemma 3.5.1 the agent comes into vertex v by the kernel cycle. Thus the kernel cycle passes through vertex v. Suppose that the rule *EatSmall* is applied on two cycles different from the kernel cycle (if not, the statement holds). Then let C be the cycle that passes vertex v at least twice. Denote by z_1 and z_2 two incoming arcs of the cycle C entering vertex v. Let y be the incoming arc of the kernel cycle entering vertex v. By the Lemma 2.2.1 $succ_v(y) \neq z_1$ or $succ_v(y) \neq z_2$. Without lost of generality, let $succ_v(y) \neq z_1$. Since the arc z_1 is the arc of cycle C, there exists arc x that is preceding arc z_1 and it is incoming arc to vertex v in cycle C. Then the rule *EatSmallKC* can be applied using edges x, y and z_1. □

Theorem 3.5.5 *Algorithm MergeCycles applied to a simple undirected connected graph $G = (V, E)$ always terminates and in the final labeling the labels 1 are assigned to the outgoing arcs in the kernel cycle.*

Proof: By Lemma 3.5.2 the number of application of rules *Merge3KC* and *EatSmallKC* is finite. After each application of the rules, the length of the new kernel cycle is recomputed. This recomputation is finite as it is done by RH-traversal on a finite kernel cycle. The algorithm terminates after no rule can be applied anymore, and after a finite number of edge traversals that is identical to the length of the kernel cycle. As the rules *Merge3KC* and *EatSmallKC* can be applied only for a limited amount of times, the length extension is limited and at some point the algorithm terminates.

Second statement trivially holds by line `18`. □

Lemma 3.5.6 *Let algorithm MergeCycles terminate in a connected undirected simple graph $G = (V, E)$ resulting in the kernel witness cycle W. Then neither rule Merge3KC nor rule EatSmallKC can be applied in a vertex $v \in V$ such that $v \in W$.*

Proof: Whenever rule *Merge3KC* or *EatSmallKC* is applied in the algorithm *MergeCycles*, the length of the kernel cycle is recomputed and the calculation of the number of vertices in a row where neither rule *Merge3KC* nor *EatSmallKC* can be applied starts. The algorithm terminates when the number of vertices in a row where none of rules can be applied is same as the length of the kernel cycle and therefore the statement holds. □

Lemma 3.5.7 *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 kernel cycle (denoted by W) passes through v but not through x. Then, either the rule Merge3KC or EatSmallKC can be applied in vertex v.*

Proof: Denote by u and w the neighbouring vertices of vertex v that are before and after the vertex v on the kernel cycle W and by π_z the local orientation for the vertex $z \in V$.

Formally, the arcs $\overrightarrow{(u,v)}$ and $\overrightarrow{(v,w)}$ belong to kernel cycle and $succ_v((u,v)) = \pi_v((v,w))$. By the assumption, $x \neq u$ and $x \neq w$. As $(x,v) \in E$, $x \notin W$ and the Theorem 2.2.4, the arcs $\overrightarrow{(x,u)}$ and arc $\overrightarrow{(u,x)}$ are included in the witness cycles C_1 and C_2 ($C_1 \neq W$, $C_2 \neq W$).

If C_1 and C_2 are different, then at least three different witness cycles pass through the vertex v (W, C_1 and C_2) and therefore by Lemma 3.2.1 and by Lemma 3.5.3 the rule *Merge3KC* can be applied in the vertex v.

Let $C_1 = C_2 = C$. We will show that in this case, the cycle C passes through vertex v at least twice and therefore by the Lemma 3.3.3 and Lemma 3.5.4 the rule *EatSmallKC* can be applied.

For a proof by contradiction, let C pass through the vertex v once in each direction. Then, the arcs $\overrightarrow{(x,v)}$ and $\overrightarrow{(v,x)}$ are the two arcs that come in and come out of the vertex v. As these edges are the only ones that are passing through vertex v in the cycle C, then, by the definition of the witness ordering, arc $\overrightarrow{(x,v)}$ must be followed by the arc $\overrightarrow{(v,x)}$. By the definition of $succ_v$ function, $succ_v((x,v)) = \pi_v((v,x))$ and by Lemma 2.2.7 the degree of vertex v is 1, which contradicts the assumptions. □

Lemma 3.5.8 *Execute the algorithm MergeCycles on a simple connected undirected graph G and denote the resulting kernel witness cycle by W. Then, if C is any witness cycle in the resulting graph, there exists a vertex w such that $w \in C$ and $w \in W$.*

Proof: Suppose that C is a witness cycle which has no vertex in common with cycle W. As the graph G is connected, there exists a path P between a vertex $u \in C$ and a vertex $w \in W$ whose inner part contains vertices from neither C nor W ($u \notin W$ and $w \notin C$ from the assumption). Let e be the edge that is incident with w and is in path P (let the other vertex of e be $v \in V$) as we show in Figure 3.3.

By the definition of witness ordering and Theorem 2.2.4, there exists a witness cycle C_1 passing through arc $\overrightarrow{(v,w)}$ and a witness cycle C_2 passing through arc $\overleftarrow{(v,w)}$. By Theorem 2.2.6 and the fact that e is the edge of the path P that is not contained in W, $C_1 \neq W$ and $C_2 \neq W$.

Then, by the Lemma 3.5.7, one of the rules can be applied on w as there are two witness cycles $C_1 \neq W$ and $C_2 \neq W$. This contradicts the Lemma 3.5.6, because w belongs to W and the algorithm *MergeCycles* already terminated. □

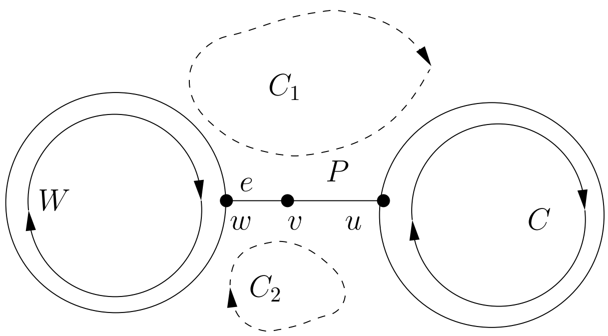

Figure 3.3: Two disjoint witness cycles W and C connected by the path P

Theorem 3.5.9 *Let $G = (V, E)$ be a simple undirected connected graph, let algorithm MergeCycles terminate in G. Then in any vertex of the entire graph G neither rule Merge3KC nor rule EatSmallKC can be applied.*

Proof: Note that this is a stronger claim than Lemma 3.5.6. Denote by W the resulting kernel witness cycle produced by our algorithm *MergeCycles* and denote by π_u the witness ordering in a vertex $u \in V$. For the sake of contradiction, let $u \in V$ be a vertex, where one of the rules *Merge3KC* or *EatSmallKC* can be applied.

By Lemma 3.5.6 we know that $w \notin W$. By Lemma 2.2.4, there exists a witness cycle $C \neq W$ such that $u \in C$. By Lemma 3.5.8 there exists a vertex w such that $w \in C$ and $w \in W$. If more than two witness cycles are passing through the vertex w, the rule *Merge3KC* can be applied, which contradicts Lemma 3.5.6. Thus only cycles W and C are passing through the vertex w.

Let $w_0 = w$, and let w_1 be an arbitrary vertex such that the arc $\overrightarrow{(w_0, w_1)}$ belongs to C. We may now construct an infinite sequence $\{w_i\}$ using the following relation:

$$succ_{w_{i+1}}((w_i, w_{i+1})) = \pi_{w_{i+1}}((w_{i+1}, w_{i+2})),\ \forall i \in \mathbb{N}.$$

Note that $\forall i \in \mathbb{N}$, $\overrightarrow{(w_i, w_{i+1})} \in C$. From the facts that $w_0 \in W$, $u \in C$ and $u \notin W$ we get that $\exists n \in \mathbb{N}, w_{n+1} \notin W \wedge (\forall i \leq n, w_i \in W)$.

By the Theorem 2.2.4 the arcs $\overrightarrow{(w_n, w_{n+1})}$ and $\overleftarrow{(w_n, w_{n+1})}$ determine witness cycles C', C''. As $w_{n+1} \notin W$, clearly $C' \neq W$ and $C'' \neq W$. If $C' \neq C''$, we may apply the rule *Merge3KC* in w_n, otherwise we may apply the rule *EatSmallKC*. Either way, we get a contradiction with Lemma 3.5.6. □

Lemma 3.5.10 *During our algorithm MergeCycles in each vertex v of a simple connected undirected graph $G = (V, E)$ we only apply the rules Merge3KC and EatSmallKC once as a batch of local changes. In other words, once we discover that none of the rules can be applied in a particular vertex v, this vertex is final – no rule applications in v will be possible in the future.*

Proof: For a proof by contradiction, let v be a vertex on the kernel cycle where neither the rule *Merge3KC* nor *EatSmallKC* can be applied at some moment during the execution of the algorithm *MergeCycles*, but it will become possible to apply one of the rules later. We start by observing that there can be at most two witness cycles passing through v, otherwise the rule *Merge3KC* would be applicable. One of these cycles is the kernel cycle and we will denote it W. The other witness cycle (if it exists at all) will be called C. As the rule *EatSmallKC* can not be applied at this moment, we know that C, if it exists, only passes through v once.

Applying the rule *Merge3KC* in some other vertex w either leaves the cycle C untouched, or it connects it to the kernel cycle. Similarly, application of the rule *EatSmallKC* in w will either leave C untouched, or it transfers a part of C to the kernel cycle. In either case the vertex v will never contain more than two witness cycles, and the cycle C will never pass through it more than once. Thus none of the rules will be applicable in v in the future. □

Theorem 3.5.11 *Let G be a simple connected undirected graph. Suppose that the algorithm MergeCycles already terminated on G. Let W be the kernel witness cycle constructed by the algorithm. Then W contains all the vertices in G.*

Proof: The statement can be proved in the same way as in Theorem 3.5.9. By contradiction, let $u \in V$ be a vertex such that $u \notin W$. Using the same reasoning as in Theorem 3.5.9, we will find $n \in \mathbb{N}$ and two vertices w_n and w_{n+1} such that $w_n \in W$, $w_n \in C$ and $w_{n+1} \notin W$, $w_{n+1} \in C$. By the same consideration, neither of the arcs $\overrightarrow{(w_n, w_{n+1})}$ and $\overleftarrow{(w_n, w_{n+1})}$ can belong to the kernel cycle W. Thus, one of the rules *Merge3KC* or *EatSmallKC* can be applied in w_n (according to Lemma 3.5.7) and this is a contradiction, with the termination of the algorithm □

Theorem 3.5.12 *The time complexity of the algorithm MergeCycles applied to a simple undirected connected graph $G = (V, E)$ is $O(M^2\Delta)$, where Δ is the maximal degree of a vertex in V and $|E| = M$.*

Proof: By Theorem 3.5.5 the algorithm *MergeCycles* terminates. The length of the kernel witness cycle increases with every use of the rules *Merge3KC* and *EatSmallKC*. The maximal length of the kernel cycle is $2M$ when all edges are used in both directions. Thus the rules can only be applied $O(M)$ times. By Lemma 3.2.4 and Lemma 3.3.4, one application of the rule can be done in $O(M\Delta)$. In the worst case, the rule is applied only once during each whole traversal of the kernel cycle. The length of the kernel cycle is $O(M)$ and thus the total time complexity is $O(M^2\Delta)$. □

Theorem 3.5.13 *The memory needed for the algorithm MergeCycles applied to a simple undirected connected graph $G = (V, E)$ is one pebble and $O(\log N)$ memory for the agent that executes the algorithm.*

Proof: The pebble is needed while applying the rule *Merge3KC*, rule *EatSmallKC* and while calculating the length of the kernel witness cycle being built. Applying these rules and calculating length is done sequentially and thus only one pebble is needed at any moment.

In the algorithm *MergeCycles* while processing a single vertex v the agent needs to keep a finite number of edge labels. A label of an edge is a number from 1 to d_v and $d_v \leq N$. Therefore a single edge label can be stored in $O(\log N)$ memory. In addition to the edge labels, our agent needs two more variables – counters for the length of kernel cycle and the number of vertices in a row, where neither rule could be applied. At any time, these variables contain numbers of size $O(N^2)$, and those can be stored in $O(\log N)$ memory. Therefore the statement is true. □

Chapter 4

Algorithm for spanning tree

In this chapter we will discuss the algorithm for constructing a (rooted) spanning tree. As the input, we will consider a network modeled as a simple undirected connected graph with a local orientation. Note that the input is a simple undirected connected graph along with the labelings of its edges and therefore by Lemma 2.2.2 the witness ordering is given. We assume that the labels of the edges incident to a vertex v are $1, 2, \ldots, d_v$, where d_v is the degree of vertex v.

The algorithm will make changes to labeling, so that when it finishes, the edge with label 1 will be the link to the vertex's father in the spanning tree (for the root of the spanning tree, the edge with the label 1 will lead to one of its direct siblings).

The changes in the labeling need to be done by an agent able to exchange two labels in a node. The vertex where the algorithm starts will be the root of the spanning tree. The extra requirement for the algorithm is to do all the precomputations with minimal memory requirements.

4.1 Main idea

The basic idea of our approach is to first use the algorithm from Chapter 3 and create a kernel witness cycle and then identify the edges of the kernel cycle that belong to the spanning tree. The root of the built spanning tree becomes the vertex where the algorithm begins and thus the vertex where the algorithm *MergeCycles* from Chapter 3 starts and terminates. When a vertex is discovered for the first time by the agent in an RH-traversal on the kernel cycle, it is added to the spanning tree by "rotating" the labels, such that the label 1 is the label of the discovering edge. Rotations of local orientations in the vertices do not harm the kernel witness cycle as the RH-traversal (except the first step) relies only on the ordering of the edges and not on the particular labels.

The decision whether the vertex v was discovered for the first time using the edge e as the incoming edge is done as follows: First, the vertex v is marked by the pebble. Then a "backward" RH-traversal (RH-traversal that is using the predecessor function) is started, using the incoming, possibly discovering edge e. The backward RH-traversal will terminate in one of these cases:

1. During the RH-traversal, the agent enters the root vertex of the spanning tree by the edge with label 1 (it is the beginning of the previous RH-traversal). In other words, the agent returns to the beginning of the RH-traversal that is adding vertices to the spanning tree. Therefore, the vertex v is new and the edge e is the discovering edge that needs to be added to the spanning tree. Below we describe how to detect this situation.
2. The agent enters a vertex with the pebble. Then, the tested vertex is on the way to the root and thus it was discovered before.

In both cases, the result is clear and the agent RH-traverses back to vertex v via edge e. If the edge e was the discovering edge, the local orientation is rotated. In both cases, the RH-traversal continues by checking the next vertex.

The only remaining question is how to decide which vertex is the root vertex of the spanning tree. Our approach is to use $O(\log N)$ memory in the agent, where N is the number of vertices in the graph. As the agent is traversing the unchanged path defined by the kernel cycle forwards (by using $succ()$ function) and backwards (using $pred()$), the RH-traversal distance from the beginning can be maintained. The length of the kernel cycle can be calculated beforehands (it does not change during the execution of our algorithm) and the termination condition of the algorithm becomes a simple comparison of two numbers – the actual distance and the length of kernel cycle. Thus, testing if the agent is in the root vertex of the spanning tree can be done easily during the backward RH-traverse.

An alternate approach would be using a second pebble (different from the first one) to mark the root vertex.

4.2 Algorithm CreateSpanningTree

This section contains a detailed discussion of the algorithm for constructing the spanning tree and its pseudocode. The agent will have five states: `Search`, `ToRoot`, `ToOldVertex`, `ToNewVertex` and `Precompute`. Figure 4.1 shows the possible transitions between these five states. The description of the states follows:

`Precompute`: This is the initial state of the agent where the length of kernel witness cycle is computed.

`Search`: This state is used during the search for the next vertex to be added to the spanning tree.

`ToRoot`: In this state, the agent traverses to the root of the spanning tree while checking whether it passes the marked vertex.

`ToOldVertex`: The state is used for RH-traversing back to the tested vertex, that is already in the spanning tree.

`ToNewVertex`: In this state the agent RH-traverses to the tested newly discovered vertex.

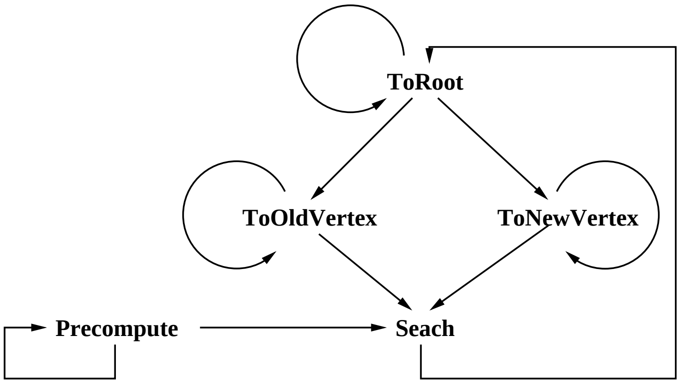

Figure 4.1: The changes of states of the agent.

The variable `state` is used for saving the state of the agent. The current distance from the starting point of the RH-travel is stored in variable `dist` and the total length of the kernel cycle is held in variable `maxLength`.

In the further text, we will assume that the input graph for the *CreateSpanningTree* algorithm is the output of the algorithm from Chapter 3. Therefore, the kernel witness cycle containing all arcs with the label 1 will be available.

The pseudocode follows: (`succ_v(e)` and `pred_v(e)` denote the operations $succ_v(e)$ and $pred_v(e)$ respectively)

```
on arrival to v via e
  if (state is Search)
    if (dist is maxLength)
      exit; // the spanning tree is done
    else
      put pebble to v;
      state = ToRoot;
      dist = dist - 1;
      go via e;
  elseif (state is ToRoot)
    if (is in vertex v pebble)
      state = ToOldVertex;
      dist = dist + 1;
      go via e;
    else
      if (dist is 0)  // root
        state = ToNewVertex;
        dist = dist + 1;
        go via e;
      else
```

```
        dist = dist - 1;
        go via pred_v(e);
  elseif (state is ToOldVertex)
    if (is in vertex v pebble)
      take pebble from vertex v;
      state = Search;
      dist = dist + 1;
      go via succ_v(e);
    else
      dist = dist + 1;
      go via succ_v(e);
  elseif (state is ToNewVertex)
    if (is in vertex v pebble)
      rotate the labeling such that label of edge e is 1;
      take pebble from vertex v;
      state = Search;
      dist = dist + 1;
      go via succ_v(e);
    else
      dist = dist + 1;
      go via succ_v(e);
  elseif (state = Precompute)
    if (pebble is in vertex v and succ(e)_v is 1)
      // end of precomputations
      state = Search;
      dist = 1;
      take pebble from v;
      go via 1;
    else
      maxLength = maxLength + 1;
      go via succ_v(e);
```

```
CreateSpanningTree in vertex v
state = Precompute;
put pebble to vertex v;
maxLength = 0;
if (vertex degree > 0)
  maxLength = maxLength + 1;
  go via 1;
else
  exit; //spanning tree is done
```

4.3 Proofs of correctness and complexity

Definition 4.3.1 *We say that vertex v is discovered when the agent enters the vertex v for the first time during an RH-traversal.*

Definition 4.3.2 *Let w be the vertex where the RH-traversal begins (via the edge with label 1). The RH-distance of some vertex v and incident edge e is the number of edges that are traversed in RH-traversal starting in the vertex w via the edge with label 1 and ending in the vertex v by incoming edge e. For the arcs that are not part of the kernel cycle, this distance is infinite and will never be used.*

Lemma 4.3.3 *Let $G = (V, E)$ be a simple undirected connected graph which will be used as the input for the CreateSpanningTree algorithm. Let $v \in V$ be a vertex where the agent starts performing the algorithm. Then, after constructing the kernel cycle by using the MergeCycles algorithm, the agent will be located in vertex v and it will be holding the pebble used in the algorithm.*

Proof: It is straightforward from the description of the algorithm *MergeCycles* from Chapter 3. □

Lemma 4.3.4 *Let $G = (V, E)$ be a simple undirected connected graph on which the algorithm CreateSpanningTree will be applied. Let $v \in V$ be the vertex where the algorithm CreateSpanningTree starts. After leaving the state* `Precompute`*, the agent is located in the vertex v, its new state is* `Search` *and the variable* `maxLength` *contains the number of edges in the kernel witness cycle.*

Proof: The algorithm *CreateSpanningTree* starts in vertex v so, according to Lemma 4.3.3, the agent is in vertex v in state `Precompute` when the algorithm *MergeCycles* terminates. If the degree of the vertex v is zero, the algorithm terminates immediately as the vertex itself is a spanning tree.

When the degree is higher, the vertex v is marked by the pebble. Then, the variable `maxLength` is increased to 1 and the RH-traversal begins via edge with label 1. Whenever the agent in the state `Precompute` enters a vertex which does not contain the pebble, the variable `maxLength` is increased and agent continues in RH-traversal by successive edge of incoming one. When the agent enters the vertex with pebble and the successive edge does not have label 1, the RH-traversal is not terminating and thus the variable `maxLength` is increased by one and the agent continues in the RH-traversal by successor edge of the incoming one. If the agent enters the vertex v (the one marked with pebble) and the successive edge of the incoming one has label 1, the RH-traversal came to the starting point and thus the state of agent is changed.

The variable `maxLength` is increased by 1 each time we traverse an edge and thus after the RH-traversal of the whole kernel cycle, the variable `maxLength` contains the length of the kernel cycle. Note that the variable `maxLength` is modified only in the state `Precompute` and as soon as the agent leaves this state, it cannot return to it at any later point. Thus the statement holds. □

Lemma 4.3.5 *Throughout the algorithm CreateSpanningTree, the agent can only change its state as follows: state* `Precompute` *to state* `Search`*, state* `Search` *to state* `ToRoot`*, state* `ToRoot` *to state* `ToNewVertex` *or* `ToOldVertex` *and states* `ToNewVertex` *and* `ToOldVertex` *to state* `Search`

Proof: The transition of the states is shown in the Figure 4.1. After the application of the *MergeCycles* part of the algorithm, the agent is in state `Precompute`. By Lemma 4.3.4, it changes its state to `Search`. The only transition from the state `Search` leads to the state `ToRoot`. Similarly, the only transitions from the state `ToRoot` lead to `ToNewVertex` and `ToOldVertex` states. Finally, the only transitions from `ToNewVertex` and `ToOldVertex` states lead to the `Search` state. As there are no other transitions possible, the statement holds. □

Lemma 4.3.6 *Let w be the vertex of a simple undirected connected graph $G = (V, E)$ in which the algorithm CreateSpanningTree starts its execution. During the execution, let v be some vertex which the agent enter using the incoming edge e. Then at this moment the value of the variable* `dist` *is the RH-distance of vertex w and vertex v using the edge e.*

Proof: We will prove the statement by structural induction on the states.

- Base: By Lemma 4.3.3, after the execution of *MergeCycles*, the agent will be located in the vertex w and its state will be `Precompute`. Afterwards, the length of the whole kernel cycle is calculated. The variable `dist` is not used in this part. The first time it is used is when the algorithm reaches the line `45`, where it is set to 1. The state of the agent is set to `Search` and the RH-traversal starts via the edge with the label 1. At this moment the statement holds.

- Inductive step: We will analyze possible states:

  – `Search`: The previous state of our agent was either `ToNewVertex` or `ToOldVertex`. In both cases, the agent RH-traversed to the next vertex by successive edge and the variable `dist` was increased by 1. The statement then holds by inductive hypothesis.

  – `ToRoot`: The previous state of the agent was either `ToRoot` or `Search`. In case of the state `Search`, the agent traversed back via the incoming edge and the variable `dist` was decreased by 1 and thus by the inductive hypothesis, the statement holds. In case of the state `ToRoot` the variable `dist` is decreased and the agent traverses back via the predecessing edge and the statement holds by the inductive hypothesis.

  – `ToNewVertex`: The previous state of the agent was either `ToNewVertex` or `ToRoot`. In case of the transition from `ToNewVertex` to `ToNewVertex`, the variable `dist` is increased by one and the traversal continues via successive edge and thus by inductive hypothesis the statement holds. In case of the change of state from `ToRoot` to `ToNewVertex`, and by the inductive hypothesis, the distance of the agent from vertex w was 0, meaning the agent was in vertex w. Then, the variable `dist` is increased by 1 and the agent traverses back by the incoming edge (this is the edge with label 1). Therefore, the statement holds.

– `ToOldVertex`: The previous state of agent was either `ToOldVertex` or `ToRoot`. In the case of the transition from `ToOldVertex` to `ToOldVertex`, the variable `dist` is increased by one and the traversal continues via successive edge and thus by inductive hypothesis the statement holds. In the case of the change of state from `ToRoot` to `ToOldVertex`, the traversal carries on by returning via the incoming edge and thus the RH-distance increases by one, just like the variable `dist`. Thus, by the the inductive hypothesis, the statement holds.

– `Precompute`: The agent cannot enter this state, so it is irrelevant for this analysis.

□

Lemma 4.3.7 *Let $G = (V, E)$ be a simple undirected connected graph where the algorithm CreateSpanningTree is executed. Let $w \in V$ be the vertex where the execution starts. Suppose that at some moment during the execution of the algorithm CreateSpanningTree the agent was in a vertex $v \in V$, the incoming edge was e and the state of the agent just changed from* `Search` *to* `ToRoot`*. Then, the agent will RH-traverse backwards on the kernel cycle, terminating in one of the following cases:*

1. *the agent enters vertex v again, then the state is changed to* `ToOldVertex`
2. *the agent enters vertex w via an edge with the label 1, then the state is changed to* `ToNewVertex`

Proof: When the state `Search` is changed to state `ToRoot` in vertex v (line 7), the vertex v is marked by a pebble and the traversal starts via the incoming edge. Whenever the agent is in the state `ToRoot` and the current vertex is neither v (it is not marked by pebble) nor the variable `dist` is zero, by the algorithm *CreateSpanningTree* the traversal is continued via predecessing edge of the incoming one. Thus, the backward RH-traversal is done.

There are two cases when the state `ToRoot` can be changed. In the first case, the vertex on the way of backward RH-traversal contains a pebble. Then this vertex has to be the vertex v and the new state is `ToOldVertex` and our statement holds. The second case occurs when the variable `dist` decreases to zero. According to Lemma 4.3.6, this occurs only if the RH-distance of the current vertex and current incoming edge is zero. Therefore, the only vertex where this case can arise is the initial vertex w and the incoming edge has label 1. Then, the new state is `ToNewVertex` and thus the statement holds. □

Lemma 4.3.8 *During the execution of the algorithm CreateSpanningTree, the variable* `dist` *is non-negative.*

Proof: During the execution of the algorithm, there are two situations where the variable `dist` is decreased.

First situation occurs during the backward RH-traversal in the state `ToRoot`. By Lemma 4.3.7, the agent RH-traverses backwards in the state `ToRoot` until the change of the state occurs. State `ToRoot` is changed when `dist` ≥ 0 to the state `ToNewVertex` or `ToOldVertex`. Then, in both cases, in the next step, `dist` is increased and thus in this case the statement holds.

The second situation when the variable `dist` is decreased by 1 is the state `Search`. The agent can come to this state from the state `Precompute` or from the states `ToOldVertex` or `ToNewVertex`. The first case occurs after the calculation of the length of the kernel cycle. The `dist` is set to 0 and subsequently to 1 (at line `45`) so the variable `dist` will remain non-negative even when it is decreased by one later and the statement holds. In the second case when the state is changed from `ToOldVertex` or `ToNewVertex` to the state `Search`, the variable `dist` is first increased by one, the state is changed and then it is decreased by one. The new state becomes state `ToRoot` where the statement from the previous paragraph holds. □

Lemma 4.3.9 *Let w be the vertex of a simple undirected connected graph $G = (V, E)$, where the algorithm CreateSpanningTree starts. Let $v \in V$, e_1 and e_2 be two edges incident to the vertex v such that e_2 is the successor of edge e_1. Let $e_1 = (u_1, v)$ and $e_2 = (v, u_2)$, $u_1, u_2 \in V$ (see in Figure 4.2). Let the agent, during the execution of the algorithm CreateSpanningTree (after the execution of MergeCycles algorithm), enter the vertex v via edge u_1 and change its state to* `Search`*. Let RH-distance of vertex u_2 and edge e_2 be smaller than the length of the whole kernel cycle. Then, the agent enters the state* `Search` *for the next time in vertex u_2 with the incoming edge e_2.*

Proof: The transition of state to the state `Search` can happened from states `Precompute`, `ToNewVertex` or `ToOldVertex`. In case of the state `Precompute`, it must have traversed via the edge with label 1, in other cases, it traversed via the successive edge. By the assumption, the successive edge is e_2 and thus by the algorithm *CreateSpanningTree* (lines `2-9`), the agent enters the vertex u_2 via the edge e_2 in the state `Search`. As the RH-distance of u_2 via edge e_2 is less than `maxLength`, the pebble is put to the vertex u_2 (the incoming edge is e_2), state is changed to `ToRoot` and the agent returns to root via edge e_2. Now, by Lemma 4.3.7, let r be the vertex which the agent enters via the incoming edge e_3 and subsequently changes its state from `ToRoot`. Then, the state is changed to either `ToNewVertex` or `ToOldVertex` and the traversal continues via edge e_3. While no pebble is passed by agent, the agent continues the RH-traversal in the states `ToNewVertex` or `ToOldVertex`. The state is changed when the agent enters the vertex with the pebble – that is, u_2 via the incoming edge e_2. Then, the state is changed back to `Search`, thus proving the statement. □

Corollary 4.3.10 *Let $G = (V, E)$ be a simple undirected connected graph. Suppose that at some moment our agent is in the state* `Search`*. Then during the entire run of the algorithm so far the value of the variable* `dist` *was never larger than it is now.*

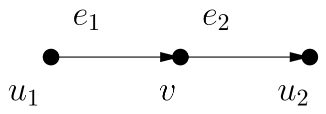

Figure 4.2: The vertex v with incoming edge e_1, successive edge e_2 and appropriate vertices u_1 and u_2

Corollary 4.3.11 *The vertices of a simple connected undirected graph G are discovered in the CreateSpanningTree algorithm in the order corresponding to increasing RH-distance from the initial vertex and its incident edge with label 1.*

Lemma 4.3.12 *A vertex can be discovered by the agent only while it is in the* `Search` *state.*

Proof: The proof is obvious from the previous statements and the pseudocode. □

Theorem 4.3.13 *Let $G = (V, E)$ be a simple connected undirected graph. The execution of the algorithm CreateSpanningTree terminates on G.*

Proof: Let the algorithm start the execution in a vertex $v \in V$. By Lemma 3.5.5 and Lemma 4.3.4, the agent enters the state `Search` in the vertex v (if it did not terminate before, that could happen when the degree of the vertex v was zero). By Lemma 4.3.5, the states are changed according to the Figure 4.1. By Lemma 4.3.6, the variable `dist` is the RH-distance of the current vertex and an the current incoming edge in the algorithm. This distance is non-negative by Lemma 4.3.8 and by the Corollary 4.3.10 the value of `dist` is increased by one in each cyclical transition of the states, starting and ending in state `Search`. Therefore, by Lemma 4.3.5 the agent will eventually reach the state `Search` and `dist` will be equal to the length of the whole kernel cycle. Then, by Lemma 4.3.4, `dist` = `maxLength` and thus the algorithm terminates in the line `4`. □

Lemma 4.3.14 *Let $G = (V, E)$ be a simple connected undirected graph. Let $w \in V$ be a vertex where the CreateSpanningTree algorithm starts. Let the agent be located in a vertex $u \in V$, change its state to* `Search`*, RH-traverse to the vertex v via the edge e in state* `Search` *($e = (u, v)$) and discover the vertex v via the edge e. Then, the state is changed to* `ToRoot`*, the agent RH-traverses backwards to vertex w and finally, changes its state to* `ToNewVertex`*.*

Proof: Since the agent discovers vertex v via edge e while in state `Search`, it follows that `dist` = `maxLenght` (line `3`) cannot hold. Thus the vertex v is marked with the pebble, the state is changed to `ToRoot` and by Lemma 4.3.7 the agent RH-traverses backward via link e. As the vertex v was discovered in the previous step, it cannot be passed in the way back from the vertex v to the vertex w and thus no pebble is found. Then by Lemma 4.3.7 the agent enters vertex w by the edge with label 1 and the state is changed to `ToNewVertex`. □

Definition 4.3.15 *Let $G = (V, E)$ be a simple connected undirected graph, let π_v be the witness ordering for the vertex $v \in V$. Let $E^* = \{e \in E \mid e = (u, v),\ u, v \in V,\ u, v$ be discovered, $\pi_u((u, v)) = 1 \ \vee\ \pi_v((u, v)) = 1\}$. Let $V^* = \{v \in V \mid \exists u \in V,\ e = (u, v) \in E^*\}$. Then, the graph $G^* = (V^*, E^*)$, which is changing during the execution of the algorithm MergeCycles according to the set of discovered vertices and the rotations of the local orientation in these vertices, will be called the kernel graph for the witness orderings π_v, $v \in V$.*

Theorem 4.3.16 *Let $G = (V, E)$ be a simple connected undirected graph. Then, whenever the agent is taking the pebble (lines* `46`, `25` *and* `35`*) during the execution of the CreateSpanningTree algorithm, the kernel graph $G^* = (V^*, E^*)$ defined by the actual witness ordering π_v, $v \in V$ is connected.*

Proof: Let $w \in V$ be the vertex where algorithm begins. Then by Lemma 4.3.3 the algorithm will continue after the *MergeCycles* part in the vertex w.

We will prove the statement by induction on the number of visited vertices in G during the execution of the algorithm *CreateSpanningTree*:

- Base: At the beginning, the vertex w is discovered, E^* is empty and thus G^* is connected. If the degree of w is greater than zero, by Lemma 4.3.4 the agent continues in the vertex w and in the state `Search`. The change of states from `Precompute` to `Search` is done at line `44`. No other vertex is discovered in the meantime, so when the pebble is taken from the vertex at line `46`, the statement holds.

  After this transition of the states, the agent continues in the state `Search` and traverses via edge with label 1 By Lemma 4.3.12 the new vertex u is discovered by the edge e and it is marked with pebble. Then, by Lemma 4.3.14 the agent returns to vertex w and changes the state to `ToNewVertex`. As the change of the states was done from `ToRoot` to `ToNewVertex`, by Lemma 4.3.5 the agent RH-traverse back to vertex u via edge e where the pebble is stored. When the agent is situated in vertex u in the state `ToNewVertex`, the line `34` is executed. The labeling in the vertex u is rotated so that the incoming (discovering) edge e has label 1. As the rotation of the ordering is done only on labels, no changes are made in the ordering for RH-traversal as here only the ordering of vertices is used. As the discovering edge has label 1 and the vertex is discovered, the kernel graph G^* is connected and consists of two vertices – the root w and the discovered vertex v, connected by an edge.

- Inductive step: According to Lemma 4.3.12, during the execution of the algorithm *CreateSpanningTree* where the transition from the state `Precompute` to state `Search` was done, the only state where a vertex can be discovered and the kernel graph G^* changed, is the state `Search`. In other state them `Search` the vertex cannot be discovered and therefore kernel graft G^* is unchanged and the statement holds.

  When the agent is in the state `Search` and enters the vertex u that was already discovered, the vertex u is marked by the pebble and state of the agent is changed to `ToRoot`. By the Lemma 4.3.7 and the fact that the vertex x is already discovered, it is found in the backwards RH-traversal. Thus the next state of the agent is `ToOldVertex`. According the Lemma 4.3.5 the agent eventually transits from the state `ToOldVertex` to the state `Search` and such a transition can occur only on the line `26`. During this cyclical transition of the states, no new vertices were discovered, so the G^* is unchanged and the statement holds.

Suppose that the agent in the state `Search` discovers the vertex u. Then `dist` $=$ `maxLenght` cannot hold (elsewhere the vertex u would not be a newly discovered one). Then the pebble is inserted to the vertex u, agent's state is changed to `ToRoot` and the agent traverses back via discovering edge e. By Lemma 4.3.14 the agent changes the state in w to `ToNewVertex`, then it starts RH-traversal from w by edge with label 1 and is RH-traversing in this state until the vertex with pebble is found. This happens in the vertex u via discovering edge e. Then by the line `34` of the algorithm *CreateSpanningTree* the labeling of discovered vertex u is rotated so that the discovering edge e has label 1. The rotations of local orientation in a vertex do not harm the RH-traversability.

Let $e = (u, v) \in E$ and let e' be the predecessing edge of e in vertex u. Then RH-distance of vertex v and edge e' is smaller than RH-distance of vertex u and edge e and thus by the Corollary 4.3.11 the vertex v was discovered before the vertex u. Therefore $v \in V^*$. After the change of labeling, the label of edge e in vertex u is changed to 1 and thus e and u becomes members of the kernel graph. By the induction hypothesis the rest of the kernel graph was connected and by adding vertex u to the kernel graph, the new kernel graph is connected.

□

Theorem 4.3.17 *Let $G = (V, E)$ be a simple connected undirected graph. Then during the execution of algorithm CreateSpanningTree (after the realization of the algorithm MergeCycles) every vertex will be discovered.*

Proof: It is straightforward, as by Theorem 3.5.11 all vertices are on the kernel cycle and all traverses are done by RH-traversal of the kernel cycle. □

Lemma 4.3.18 *Let v be the vertex of a simple undirected connected graph $G = (V, E)$, in which the algorithm CreateSpanningTree it executed. Let $e \in E$ be an edge such that $e = (u, v), u, v \in V$ and $\pi_v(e) = 1$ after the MergeCycles algorithm was executed. Then $\pi_u(e) = 1$.*

Proof: The proof is straightforward. By Lemma 4.3.3 the initial vertex after the execution of the *MergeCycles* algorithm is v and the algorithm starts in it in the `Precompute` state. Then, it changes its state to `Search` in vertex v and begins the traversal via edge e with label 1. The vertex u is discovered and therefore by Lemma 4.3.14 the agent RH-traverses backwards in the state `ToRoot` to the vertex v, changes the state to `ToNewVertex` and enters vertex u via edge e. As the vertex u was discovered, the line `34` of the algorithm is executed and the local orientation in vertex u is rotated, so that the label of the discovering edge e is set to 1. Then in the later steps of the algorithm *CreateSpanningTree*, the vertex u is already discovered and therefore the rotation of local orientation in the vertex u according the line `34` is never done and the statement holds. □

Theorem 4.3.19 *Let $G = (V, E)$ be a simple connected undirected graph and let algorithm CreateSpanningTree terminate its execution on G. Then, the kernel graph G^* is a spanning tree of graph G.*

Proof: By Theorem 4.3.16 and Theorem 4.3.17, the graph G^* is connected and consists of all vertices of G. As the vertices of G^* are connected by edges with label 1 and the labels are unique in each vertex, each vertex is connected in G^* by only one edge. In particular, this holds also for the root vertex according the Lemma 4.3.18. Therefore, G^* is connected and consists of $N - 1$ edges, thus it is a spanning tree. □

Theorem 4.3.20 *The time complexity of the algorithm CreateSpanningTree (without the algorithm MergeCycles) applied in a simple undirected connected graph $G = (V, E)$ is $O(M^2)$.*

Proof: The computation of `maxLength` in state `Precompute` is done by traversing the kernel cycle, which takes $O(M)$ edge traversals. For each vertex that is being tested whether it was discovered (in state `Search`), the agent traverses at most to the root (in state `ToRoot`). During this return the agent makes $O(M)$ steps. Then, the state is changed to `ToNewVertex` or `ToOldVertex` and the agent RH-traverses on the kernel cycle back to the vertex that is being tested. Obviously, this step amounts to $O(M)$ edge traverses as well. If the vertex was newly discovered (denoted by the state `ToNewVertex`), it is added to the kernel graph in time $O(\Delta)$ (where Δ is the maximal degree of a vertex in V), the state is changed to `Search` and in the time $O(1)$ the next vertex is entered and the test of the newly discovered vertex begins again. Whenever the agent enters a vertex during the traversal of the kernel cycle (in the state `Search`), that vertex is being tested whether it is a newly discovered one. One test takes $O(M)$ steps and the kernel cycle has $O(M)$ edges. The rotation of the local orientation is done N times as each vertex is this way added to the spanning tree. Thus the total time complexity is $O(N\Delta + M^2)$ that can be estimate to $O(M^2)$. □

Theorem 4.3.21 *The total time complexity of the algorithms MergeCycles and subsequently CreateSpanningTree applied on a simple connected undirected graph $G = (V, E)$ is $O(M^2\Delta)$, where Δ is the maximal degree of vertex in G.*

Proof: By Theorem 3.5.12, the time complexity of the algorithm *MergeCycles* is $O(M^2\Delta)$, where Δ is the maximal degree of the vertex in G. By Theorem 4.3.20, the time complexity of the algorithm *CreateSpanningTree* is $O(M^2)$ and thus the total time complexity is $O(M^2\Delta)$. □

Theorem 4.3.22 *The total memory complexity of the algorithm CreateSpanningTree applied on the simple undirected connected graph $G = (V, E)$ is one pebble and $O(\log N)$ bits of memory for the agent.*

Proof: During the execution of the algorithm, the agent uses two variables: `dist` and `maxLength`. By Lemma 4.3.4, the value of `maxLength` is the length of the kernel cycle, which is $O(M)$. Obviously, $M \leq N^2$, so the variable `maxLength` requires $O(\log(M^2)) = O(\log N)$ bits of memory. By Lemma 4.3.6, Lemma 4.3.8, Corollary 4.3.10 and the fact that the algorithm terminates, the value of `dist` is at most equal `maxLength`, thus it also needs $O(\log N)$ bits of memory. As the memory needed for algorithm *MergeCycles* is, according to Theorem 3.5.13, equal to one pebble and $O(\log N)$ bits of memory in the agent, the total memory requirements of the *CreateSpanningTree* algorithm are one pebble and $O(\log N)$ bits of the agent's memory. $\square$

Chapter 5

Algorithm MergeCycles+ using constant memory

In this chapter we will show how to adjust the algorithm *MergeCycles* from Chapter 3 to decrease memory requirements to constant memory in the agent and one pebble for marking vertices. The question whether the agent can terminate the algorithm using $O(1)$ memory and one pebble will remain open. However by increasing the amount of used memory, the termination can be solved easily.

5.1 Main idea

The main idea of the approach described in this chapter is to modify the algorithm *MergeCycles* from Chapter 3 so that the agent executing the algorithm will need just $O(1)$ memory. The modifications will be done in a few steps that are discussed in more detail in following sections.

First, we will identify the parts of the algorithm *MergeCycles* from Chapter 3 which force the agent to use $\Omega(\log N)$ memory and try to find a way to decrease that requirement. The following fragments of the algorithm raise the memory requirements to $\Omega(\log N)$:

1. *Traversing via the edges.* Obviously, the number of the incoming edge must be stored, as this information can only be obtained from the local orientation. The agent would not be able to navigate without it and the problem would become unsolvable. Therefore, this information is absolutely necessary and it is not included in the total memory calculation.

2. *Finding a witness cycle for application of the rules Merge3KC and EatSmallKC.* We will describe a method for reducing the memory usage of this activity in the following sections.

3. *Computation of the length of the kernel cycle.* Knowing the length of the kernel cycle and being able to count the number of vertices visited in a row in which neither

the rule *Merge3KC* nor *EatSmallKC* can be applied is necessary for the termination detection. The price for getting rid of these computations is a loss of termination detection – the RH-traversal of the kernel cycle will be infinite, once neither the rule *Merge3KC* nor *EatSmallKC* can be applied, the agent will not be able to detect this fact and it will travel along the kernel cycle, trying to apply the rules indefinitely.

4. *Relabeling of the edges.* This will be discussed in a separate paragraph later.

Since our goal for the agent's memory is $O(1)$, the model chosen for our algorithm and for the exchanges of the labels is crucial. In our model the agent will be able to request each of the three following operations. We do not include the necessary memory requirements in the total memory complexity.

- *apply rule Merge3KC*
- *apply rule EatSmallKC*
- *rotate local orientation*

Our model will contain six variables for performing rules *Merge3KC* and *EatSmallKC*. These variables are not stored in the agent, and the agent can only access them in a single way – when it is traversing an edge, it may decide to store it in one of the variables. This is necessary for the agent to be able to mark the edges for one of the rules, while using constant memory.

We use the notation from Sections 3.2 and 3.3, so the variables x_1, x_2 and x_3 will be used for storing the edges for rule *Merge3* as referred in Section 3.2 and x, y and z will hold the edges for rule *EatSmall* as referred in Section 3.3. Rotation of the local orientation in a vertex is important only for setting the incoming edge's label to 1 or 2. This can be implemented by using two variables, just like the rules *Merge3* and *EatSmall* discussed above but it is only a minor technical issue which we will not examine further.

We do not care about the way the operations are performed, from the agent's point of view they are atomic operations that happen instantly at the moment when they are requested. We only assume that the incoming edge stored by the agent is the same before and after the execution of each of the operations.

Note, as it was discussed before, that the particular numerical labels of the edges are not necessary for the RH-traversal. It is only their ordering that matters, so as long as we can keep the ordering intact, the RH-traversal will proceed as expected. For the sake of simplicity, we will assume that the agent is able to rotate the local orientation so that the ordering will be preserved and the label of the incoming edge will be 1 or 2 (if possible).

Now we will talk about the operations the agent is able to perform. Let v be a vertex of the network, which the agent A computing the kernel cycle is located in and let d_v denote its degree. We demand the model to provide the following operations:

- A can test whether the incoming edge e has label 1 or 2
- A can test whether the vertex v contains a pebble, it can drop and take the pebble to/from vertex v

- A can perform operations that set variables x, y, z, x_1, x_2 and x_3 as it was described in previous paragraphs
- A can rotate the local orientation so that the incoming edge will have label 1 or 2

Definition 5.1.1 *Let $G = (V, E)$ be a simple undirected connected graph. Let $u, v \in V$ be such that $e = (u, v) \in E$, let π_u, π_v be ordering in u, v respectively. We will call label $\pi_u(e)$ the* inner *label of the edge e in vertex u and the label $\pi_v(e)$ the* outer *label of the edge e in vertex v.*

5.1.1 Storing bits of information in a graph

We will consider an undirected simple connected graph $G = (V, E)$. Let $v \in V$ be a vertex of degree d. We will describe a few different approaches for storing information related to vertex v in the graph G. We will denote incident edges of vertex v by $e_1, \ldots, e_d$, such that $e_i = (v, u_i), u_i \in V$. Let π_u be the local orientation for the vertex $u \in V$.

Since the local orientation can be rotated, it is possible to set the outer label of any incident edge of vertex v to 1. Formally, set $\pi_{u_i}(e_i) = 1$, $i \in \{1, \ldots, d\}$. Moreover, in most cases, it is possible to set the outer labels of an incident edges of the vertex v to a different value e.g. 2. Naturally, this is not always possible, as some vertices might be of degree 1, but we will show how to deal with such cases later. For now, let us suppose that each vertex in the graph has degree at least two. Note that we can test whether the outer label of some edge e is 1 or not – the agent can simply traverse via the examined edge to the neighbouring vertex, use its state to remember whether the incoming edge's label is 1 or not and return back via the incoming edge.

Another important observation is that rotations of the local orientation in vertex v do not have any influence on the RH-traversal itself. Thus, we can use the label 1 to mark the edge we are currently processing.

Finally, by changing the outer labels of edges $e_1, \ldots, e_d$ according to the method described in this section, we are able to split the edges into two groups by setting their outer labels to 1 and 2 respectively. Let P be the set of edges with outer label 2. We will bound the number of elements in P by a fixed constant. As the local orientation π_v is an ordering of the edges $e_1, \ldots, e_d$, we can use the position of the agent with respect to the incoming edge to store extra information about the edges in the set P.

5.2 Changing the rule Merge3KC

In this section we describe how to transform the rule *Merge3KC* into the equivalent rule *Merge3+* with memory complexity $O(1)$.

In order to use the rule *Merge3KC* in our model, we have to set three variables described above to three arcs on which the rule will be applied according to Section 3.2. We will use the same notation as in Section 3.2. In the whole section, we will assume that all neighbouring vertices of the vertex v have degree at least 2 and thus the labels 1 and 2 are available. The case when this assumption is not satisfied is covered in Section 5.4. The general idea is to set all outer labels of the incident edges to 1 and then find and mark representative arcs of the cycles C_1, C_2 and C_3 by outer label 2 (the arc will be the edge incoming to v).

The algorithm for application of rule *Merge3+* follows: First, the outer labels of all incident edges of the vertex v will be set to 1. Inner label 1 will be used to mark the processed edge, we will move it between edges by local rotations. The processed edge will be tested whether the incoming arc defines a witness cycle different from the cycles defined by incoming arcs with outer label 2. If the outcome of this test is positive, the edge is assigned outer label 2. Then, the local orientation is rotated and the next edge is processed. When three arcs are marked with outer label 2, three different cycles were found and the rule *Merge3* can be applied. When the result is negative three different cycles do not exist and therefore rule cannot be applied. During the startup, when this rule is going to be applied, the incoming edge to the vertex v is an arc belonging to the kernel cycle. Therefore, after setting the outer labels of all edges incident with vertex v to 1, the incoming edge that was used to enter the vertex v is marked by outer label 2. This will be the first cycle C_1, which is also the kernel cycle.

The pseudocode for the rule *Merge3+* and the detailed explanation follows. By `succ_v(e)` and `pred_v(e)` we will denote the operations $succ_v(e)$ and $pred_v(e)$. By `pi_v(e)` we denote the label $\pi_v(e)$.

```
function Merge3+ for agent in vertex v incoming edge e
1:  put pebble to vertex v;
2:  rotate the ordering so that pi_v(e) = 1;
3:  set the outer label of edge e to 1;
4:  set e = succ_v(e);
5:  if (pi_v(e) is not 1)
6:    goto 3;
7:  set the outer label of edge e to 2;
8:  remember in the state that we have one marked edge;
    // now all outer labels of edges except for the one that was the our
    // incoming to v are 1
9:  set e = succ_v(e);
10: rotate the ordering so that pi_v(e) = 1;
11: if (outer label of succ_v(e) is 2)
12:   set e = succ_v(e);
13:   take pebble from v;
14:   exit "NO"; //three different cycles were not found
15: RH-traverse the witness cycle via edge succ_v(e),
    during the RH-traversal check whether some incoming edge to vertex v has
    outer label 2,
    finish when incoming edge to vertex v has inner label 1;
    // now the incoming edge e has inner label 1
```

```
16: if (no outer label of the edge was 2 during RH-traversal of an incident
    edge of vertex v)
17:   set the outer label of edge e to 2;
18:   increase by 1 the number of remembered edges of disjoint cycles that
      is remembered in state;
19:   if (the number of remembered edges is 3)
20:     set e = succ_v(e);
21:     if (outer label of edge e is not 2)
22:       goto 20;
23:     set x_1 = e; using the cycle 20-22 find the next two marked edges
        with outer label 2 and set x_2 = e and x_3 = e respectively;
24:     set e = succ_v(e);
25:     if (outer label of edge e is not 2)
26:       goto 24;
27:     take pebble from vertex v;
28:     exit "YES"; //the rule merge3 is applied
29: goto 9;
```

In the function above the only edge that is remembered is the incoming edge. Now we will discuss each step of the function to explain its functionality and implementation.

Lines `3, 7, 17`: This can be done by traversing via the incoming edge e, changing the local orientation in the neighbouring vertex and traversing back to the vertex v (the incoming edge to vertex v will be e).
Lines `4, 9, 12, 20, 24`: This can be done by traversing via $succ_v(e)$ to neighbouring vertex and returning back to v.
Lines `8, 18`: We use a constant (3) number of edges with outer label 2. Once this number is reached, the marking of edges stops, so the counter can be kept in the agent's state.
Lines `11-14`: The incident edges of vertex v are tested in order given by the local orientation, starting with the incoming edge e which is marked by outer label 2. If the successive edge of the currently processed edge has outer label 2, this means that the the successive edge is the one where the processing started and thus all incident edges were processed and three disjoint cycles were not found. Therefore algorithm terminates.
Line`15`: The RH-traversal done in this line is the check for disjoint witness cycles. The tested cycle is represented by the arc denoted by incoming edge with label 1. The cycles that have to be disjoint with our tested cycle, are represented by arcs that have outer label 2. During this RH-traversal as the pebble is in the vertex v the check for inner and outer labels can be done. In the state the agent stores the information whether some incident edge of vertex v has outer label 2 (in this case the witness cycles are not disjoint). Note that RH-traversal terminates when incoming edge has inner label 1.
Lines `16-23`: When the processed edge is the representative of a witness cycle, that is disjoint with the other cycles represented by the incoming arcs to vertex v marked by outer label 2, the counter in the state is increased and processed edge is marked by outer

label 2. Once three edges are marked, three different cycles were found and thus the rule *Merge3* can be applied by the model.
Lines `24-26`: These lines restore the original state of the incoming edge, in order to satisfy the requirement that the incoming edge must be the same before and after the execution of the rule *Merge3+*.

The first edge with outer label 2 was the incoming edge in the beginning of *Merge3+*. Note that the edges are being processed in the order defined by the local orientation. There are two cases when the *Merge3+* terminates and in both cases is the incoming edge before and after application *Merge3+* is set to the same edge:

- When there are no three disjoint cycles, the agent processes edges in a sequence and stops when an edge with outer label 2 is found. As three disjoint cycles were not found, the first edge with the outer label 2 is the one that was the incoming one before the application of *Merge3+*.
- When lines `23-26` of the algorithm *Merge3+* are executed the incoming edge is set to the one that was marked with outer label 2 first – which is precisely the incoming edge at the beginning of *Merge3+*.

As the algorithm *MergeCycles* RH-traverses the kernel cycle while trying to apply rules *Merge3* and *EatSmall*, the incoming edge to *Merge3+* is the edge of kernel cycle and thus by using *Merge3+*, the other two cycles are connected to kernel cycle. By the previous paragraph, the incoming edge to vertex v before and after the application of function *Merge3+* is the same and thus it is the edge of kernel cycle where the RH-traversal can continue. Therefore, we have designed a modification of rule *Merge3KC* requiring only $O(1)$ memory. Note that the new rule *Merge3+* is designed only for such vertices where all neighbouring vertices have degree at least two. As such, the algorithm *MergeCycles* with the rule *Merge3+* is very similar to the original one which used the rule *Merge3KC* and thus most of the statements from Chapter 3 apply to it as well.

Lemma 5.2.1 *Let $G = (V, E)$ be a simple connected undirected graph. Let $v \in V$ be a vertex such that the degree of each neighbouring vertex of v is at least two. Then, the rule Merge3 can be applied in v if and only if the rule Merge3+ can be applied in v and the result of both rules is equal.*

Proof: It is straightforward from the pseudocode and the detailed explanation. □

Lemma 5.2.2 *Let $G = (V, E)$ be a simple connected undirected graph. Let $v \in V$ be a vertex such that the degree of each neighbouring vertex of v is at least two and the rule Merge3 can be applied here. Let $e \in E$ be an incident edge of vertex v. Let the agent enter the vertex v via the incoming edge e, then the incoming edge after the application of the rule Merge3+ is e.*

Proof: It is straightforward from the pseudocode and the detailed discussion in Section 5.2. □

Corollary 5.2.3 *If the incoming edge e to a vertex where the rule Merge3+ is applied belongs to the kernel cycle, the results of the rule Merge3KC and the rule Merge3+ are equal.*

Theorem 5.2.4 *The time complexity of Merge3+ applied in a simple connected undirected graph $G = (V, E)$ is $O(E\Delta + d\Delta)$, where Δ is the maximal degree of the vertices in G and $O(d)$ is the time complexity of rotation of the local orientation in a vertex $v \in V$ such that the incoming edge's label is 1 or 2.*

Proof: Lines `1, 4-5, 8-9, 11-14, 16, 18-20, 27, 28` are performed in constant time. Lines `2, 3, 7, 10, 17` take $O(d)$ time and cycles `20-22`, `23` and `24-26` are done in time $O(\Delta)$. In cycle `3-6` the outer label of each incident edge of the vertex v is set to 1 and therefore the time is $O(d\Delta)$. The traversal done in the line `15` is $O(E)$ steps. In the cycle `9-29` that is done $O(\Delta)$ times, the rotation in line `10` and the RH-traversal from line `15` have the greatest influence on the time complexity. The time complexity of the part `1-8` is $O(d\Delta)$ and the contribution of the cycle `9-29` is $O(E\Delta + d\Delta)$. Therefore the total time complexity is $O(E\Delta + d\Delta)$. □

5.3 Changing the rule EatSmallKC

In this section, approach similar to that of Section 5.2 will be used to modify the rule *EatSmallKC*. We will adopt the notation from Section 3.3. For easier explanation, denote the initial incoming edge to vertex v as $e*$ (this is the incoming edge before the algorithm for rule *EatSmallKC* starts and thus an arc of the kernel cycle). In the whole section, we will assume that all neighbouring vertices of vertex v have degree at least 2 and thus the labels 1 and 2 are available. The case when this is not satisfied is discussed in Section 5.4. The idea is to split edges into two partitions – those that are used in the rule *EatSmallKC* (with outer label 2) and the rest (with outer label 1).

The algorithm for the application of the altered variant *EatSmall+* of the rule *EatSmallKC* follows: At first, the outer labels of all incident edges of vertex v will be set to 1 (after this operation, the incoming edge will be $e*$). Next, the incoming edge $e*$ will be marked by outer label 2 and it will become the representative of the cycle C_2 (the kernel cycle). Starting with $succ_v(e)$, by rotating the local orientation in vertex v, the inner label 1 will mark the processed edge. The processed edge will be tested whether it defines a witness cycle different from C_2 and whether it passes through vertex v at least twice. Such an edge is a good candidate for the representative of the cycle C_1. Then, this candidate is marked by outer label 2, the next incoming edge to the vertex v in the cycle C_1 is determined and the check for the specific case (when z is successor of y) in the rule *EatSmall* is done. If the rule *EatSmall* can be applied, the next edge incoming to the vertex v is marked by outer label 2.

Note that only a constant number of edges needs to be marked for application rule *EatSmall*. To distinguish edges x, y, z from each other, we can use the local orientation in the vertex v and store it in the state. As the local orientation is an ordering of incident edges, the agent is able to find which edge is the representative of cycle C_2 and which represents the cycle C_1 – see Figure 5.1.

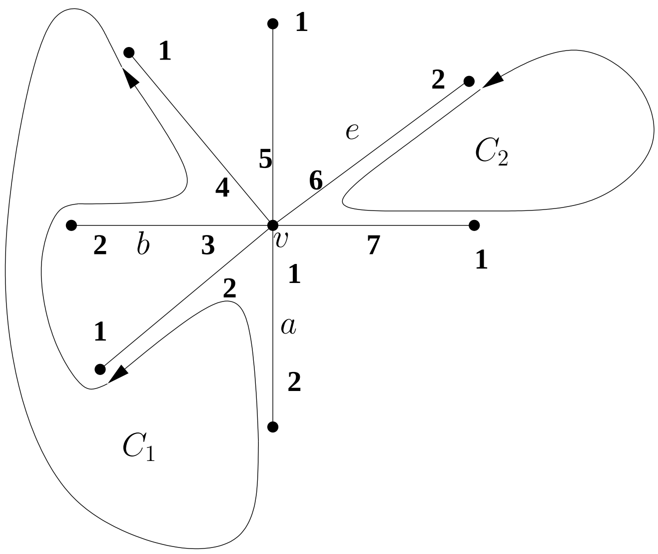

Figure 5.1: The figure is referring to the state in the middle of the execution of the rule *EatSmall+*. Cycle C_2 was marked by outer label 2 and then the edge a was discovered as the potential arc of the witness cycle (C_1) that is different from C_2 and was marked by the outer label 2. Then the cycle C_1 is tested whether the special case when the rule *EatSmall* cannot be applied occurs. In our figure the special case does not occur. After entering the vertex v via the edge b and marking the edge b with outer label 2, the incoming edge to the vertex v is set to b. Then, the agent can perform the following operation: traverse via $succ_v(b)$ and return back to set the incoming edge to $succ_v(b)$. By continuing such traversal until the incoming edge has outer label 2, the edge e or a can be found. The distinction can be done according to the inner label – a has inner label 1. Similar operations can be done with $pred_v$. Using these operations, the agent can establish its relative position with respect to b and the two marked edges (a and e). In addition to this, the agent can distinguish between the two marked edges, so it knows which one of them belongs to C_1.

The pseudocode for the rule *EatSmall+* and the detailed explanation follows. By `succ_v(e)` and `pred_v(e)` we will denote the operations $succ_v(e)$ and $pred_v(e)$. By `pi_v(e)` we denote the label $\pi_v(e)$.

```
function EatSmall+ for agent in vertex v incoming via e
    put pebble to vertex v;
    rotate the ordering pi_v so that pi_v(e) = 1;
    set the outer label of edge e to 1;
    set e = succ_v(e);
    if (pi_v(e) is not 1)
```

```
6:    goto 3;
7:  set the outer label of edge e to 2;
8:  remember in the state that one representative edge of cycle is marked;
9:  set e = succ_v(e);
10: rotate the ordering pi_v so that pi_v(e) = 1;
11: if (the outer label of succ_v(e) is 2)
12:   set e = succ_v(e);
13:   take pebble from v;
14:   exit "NO" // two cycles needed in EatSmall were not found
15: RH-traverse the witness cycle via the edge succ_v(e),
    during the RH-traversal, check whether some incoming edge to vertex v
    has outer label 2 and whether the agent visits vertex v at least
    twice. Terminate the RH-traversal when the edge with inner label 1
    is incoming to v;
16: if (no outer label of incident edge of vertex v has value 2 and
    agent has visited vertex v during the RH-traversal at least twice)
17:   set the outer label of edge e to 2;
18:   RH-traverse via edge succ_v(e) until agent again enters vertex v
19:   set e = pred_v(e);
20:   if (the outer label of e is 2 and the inner label is not 1)
      // this occurs when succ_v(y) = z and EatSmall cannot be applied
21:     set e = succ_v(e);
22:     RH-traverse backwards (using pred function) via edge e,
        terminate the RH-traversal when incoming edge to v has
        the outer label 2;
23:     set the outer label of edge e to 1;
24:     goto 9;
25:   set e = succ_v(e);
26:   set z = e;
27:   set e = succ_v(e);
28:   if (the outer label of the edge e is not 2)
29:     goto 27;
30:   if (the inner label of the edge e is 1)
31:     set x = e;
32:   else
33:     set y = e;
34:   set e = succ_v(e);
35:   if (the outer label of the edge e is not 2)
36:     goto 34;
37:   if (the inner label of the edge e is 1)
38:     set x = e;
39:   else
40:     set y = e;
```

```
41:   if (the inner label of the edge e is 1)
42:     set e = pred_v(e);
43:     if (the outer label of the edge is not 2)
44:       goto 42;
45:   apply rule EatSmall on x,y,z;
46:   take pebble from v;
47:   exit "YES";
48: else
49:   goto 9;
```

Note that the function above works only with the incoming edge e. We will now discuss and illustrate the functionality of each line of the function *EatSmall+*.

Lines `2,5,10,30,37,41:` (command `rotate the ordering pi_v that pi_v(e) = 1` and test `pi_v(e) =/!= 1`) The first command can be performed in the time $O(d)$ by means of the model, where $O(d)$ is the time the model requires for the rotation of the ordering. The second condition can be easily tested in the time $O(1)$.
Lines `3,7,11,16,17,20,23,28,35,43:` (command `set the outer label of the edge e to x` or `if (the outer label of the edge is x)`) The first command can be performed by traversing via the edge e to the neighbour, rotating the local orientation and traversing back. The time required for this operation is $O(d)$ by means of the model. The test whether the outer label of an edge e has value x is done by traversing via edge e, checking the label of the incoming edge on the other side and returning back. All that can be performed in time $O(1)$.
Lines `4,9,12,19,21,25,27:` (command `set e = succ_v(e)` or `set e = pred_v(e)`) These commands are executed in the time $O(1)$ by traversing via $succ_v(e)$ and returning directly back to v.
Lines `8,16:` The necessary information can be stored in the state of the agent.
Line `15:` RH-traversal done in this line corresponds to the test whether the incoming edge e is a representative of cycle C_1, disjoint from the one represented by the other edge with outer label 2. Also, the check whether cycle C_1 enters vertex v at least twice is done here. After the check, the only case when rule *EatSmall* cannot be applied is when $succ_v(y) = \pi_v(z)$, which will be checked later.
Lines `18,22:` The first traversal in line `18` is the search for the next edge via which the cycle C_2 enters the vertex v. The second traversal in line `22` is the opposite to the one in line `18`. When the cycle C_2 is not a good candidate for the rule *EatSmall*, by this RH-traversal agent returns back and unmark the edge representing the cycle C_2 as on this cycle the rule *EatSmall* is not applicable.
Lines `2-7:` These lines set the outer labels of the edges incident to v to 1 and the outer label of edge $e*$ to 2.
Lines `9-10:` In this line, the next edge is being processed.
Lines `11-14:` These lines deal with the case when all edges have been processed and rule *EatSmall* was not applied.

Lines `16-19`: Once the candidate for the cycle C_1 is found, the incoming edge is set to the candidate to be edge y and the check whether $succ_v(y) = \pi_v(z)$ is performed.
Lines `20-24`: If the outcome of the check $succ_v(y) = \pi_v(z)$ is positive, the rule *EatSmall* cannot be applied and thus the agent traverses backwards and unmarks the canditate edge.

Now that we have described all parts of the algorithm, we are ready to summarize the results of this section. We designed the modification of the rule *EatSmallKC* that is applicable on vertices with neighbouring vertices of degree at least two. The results of the modified rule *EatSmall+* and the original rule *EatSmallKC* are the same, while the memory consumption had been decreased to a constant.

Lemma 5.3.1 *Let $G = (V, E)$ be a simple connected undirected graph. Let $v \in V$ be a vertex such that the degree of each neighbouring vertex of v is at least two and the rule EatSmall+ can be applied in it. Let $e \in E$ be an edge incident with vertex v. If the agent enters the vertex v via the incoming edge e, the incoming edge after the application of the rule EatSmall+ is e.*

Proof: It is straightforward from the pseudocode and the detailed explanation. □

Corollary 5.3.2 *If the incoming edge e to a vertex where the rule EatSmall+ is applied is an arc belonging to the kernel cycle, the results of the rule EatSmallKC and the rule EatSmall+ are equal.*

Theorem 5.3.3 *The time complexity of EatSmall+ applied in a simple connected undirected graph $G = (V, E)$ is $O(E\Delta + d\Delta)$, where Δ is the maximal degree of the vertices in G and $O(d)$ is the time complexity of rotation of the local orientation in a vertex $v \in V$ so that the incoming edge's label becomes 1 or 2.*

Proof: Lines `1, 4, 8-9, 11-14, 16, 19-21, 25-27, 30-35, 37-43, 45-48` are performed in constant time. The time complexity of the lines `2-3, 7, 10, 17, 23` is $O(d)$ and the cycles `27-29`, `34-36` and `42-44` execute in $O(\Delta)$. The cycle `3-6` takes $O(d\Delta)$ steps. The most time consuming lines in cycles `9-24` and `9-49` that can repeat at most $O(\Delta)$ times are lines `15, 18` and `22`, all taking $O(E)$ steps and lines `10, 17, 23` that take $O(d)$ steps. Therefore the total time complexity is $O(E\Delta + d\Delta)$. □

5.4 Dealing with special cases

A careful reader must have noticed that in the functions *Merge3+* and *EatSmall+* we assumed that the outer label 2 is available for each edge of the graph. This is not necessarily true as there can be a vertex of degree 1, in which the label 2 does not exists. In this section, we will discuss the special cases that are not treated by the functions *Merge3+* and *EatSmall+*. Subsequently, the rules *Merge3+* and *EatSmall+* can be modified to handle these special cases.

We will introduce the notation for this section. Let $G = (V, E)$ be a simple undirected connected graph. Let $v \in V$ be a vertex which the agent entered via the edge e and in which the operations *Merge3* or *EatSmall* will be performed (by the means of the *Merge3+* or *EatSmall+*). Let $w \in V$ be such that $e = (v, w) \in E$. Let $u \in V$ be a neighbouring vertex of v of degree 1. As the algorithm is based on traversing the kernel cycle and using rules to prolong it, in the further text we assume that the incoming edge e to the vertex v is an arc of the kernel cycle.

We consider two types of vertices u where the special case can occur:

1. $u \neq w$:
   Vertex u must eventually be in the kernel cycle. As the degree of u is 1, both arcs $\overrightarrow{(u, v)}$ and $\overleftarrow{(u, v)}$ must be eventually arcs of the kernel cycle. When the arc (v, u) in either direction is in the kernel cycle, the vertex u is in the kernel cycle and therefore it is not interesting for us because it cannot be used in the rule *Merge3* or *EatSmall*. Thus, we can jump over it and continue in processing the edges. Our goal will be to identify such vertices and determine whether they are in the kernel cycle. If such a vertex is not in the kernel cycle, applying the rule *EatSmall* merges it to the kernel cycle, according to Lemma 3.5.4,

   The identification of a neighbouring vertex u of vertex v of degree 1 can be done by traversing the neighbouring vertex by local orientation and checking its degree. Determining whether the vertex u is in the kernel cycle is more complicated.

   We will assume that the vertex w has degree at least two. The pebble is put into the vertex v. Then, the outer label of each edge incident to vertex v can be set to 1. The arc of the kernel cycle ($\overrightarrow{(u, w)}$) can be distinguished by outer label 2. Then, the edge (u, v) can be marked by the inner label 1. The RH-traversal via the edge (u, v) (in arbitrary direction) can start and the check whether during the RH-traversal the edge with the outer label 2 incident to v is traversed can be done.

   If the vertex u is not in the kernel cycle, then at least two different witness cycles pass through vertex v. The witness cycle where the vertex u is located passes through vertex v at least twice and therefore by Lemma 3.3.3 rule *EatSmall* can be applied (there are two pairs for edge x a z and thus in one pair the rule *EatSmall* is applicable). The pseudocode (function *addDegreeOneNeighboursToKernelCycle*) can be found at the end of this section.

   In this case, all neighbours of vertex v (except for the vertex w) with degree one are added by using the rule *EatSmall* to the kernel cycle. Therefore, whenever the agent enters the vertex v via edge $e = (v, w)$ to use rules *Merge3KC* and *EatSmallKC* in the algorithm *MergeCycles*, we can start by using the function *addDegreeOneNeighboursToKernelCycle* which will guarantee that all neighbouring vertices of vertex v with degree 1 (except for vertex w) will be in the kernel cycle and therefore they can be skipped while processing active edges in rules *Merge3+* and *EatSmall+*. Using the combination of rules *Merge3+* and *EatSmall+* in this manner will keep the modified algorithm *MergeCycles* to be very similar to the original one. The only difference will be in the order of the application of the rules *Merge3KC* and *EatSmallKC* or *Merge3+* and *EatSmall+* respectively.

Note that the order of applications of the original rules *Merge3KC* and *EatSmallKC* in a vertex in the algorithm *MergeCycles* is not important. The original algorithm *MergeCycles* will work correctly for any order of applications of the rules and will always find a final witness cycle that pass through all vertices. Therefore, most of the proofs of statements from Section 3 can be adapted to this modified variant.

2. $u = w$:
This case is the most important one as all the other cases rely on it. Unfortunately, it cannot be solved in a way similar to the case $u \neq w$ as the vertex w is in the kernel cycle and both functions *Merge3+* and *EatSmall+* need it to be marked by inner label 2.

We can make the following observation according to the number of vertices that were processed during the *MergeCycles* algorithm (the same observation holds for the modification using the functions *addDegreeOneNeighboursToKernelCycle*, *Merge3+* and *EatSmall+*)

- If two or more vertices were already processed by the *MergeCycles* algorithm, the fact that the vertex w ($e = (v, w)$ is the incoming edge to v and thus it is an edge of the kernel cycle) has degree 1 implies that the vertex v was processed before. According to Lemma 3.5.10, neither of the rules *Merge3* and *EatSmall* can be applied in vertex v, so we can simply continue the RH-traversal of the kernel cycle to the next vertex by the edge with label $succ_v(e)$.

  The same reasoning can be performed for the modified algorithm *MergeCycles* where functions *addDegreeOneNeighboursToKernelCycle*, *Merge3+* and *EatSmall+* are used, as discussed in previous sections.

- If only one vertex was processed before, it was the vertex w. As the degree of vertex w is 1, the previous vertex in the kernel cycle is vertex v and the following part of the kernel cycle is interesting: $pred_v(e)$, v, $\overrightarrow{(v, w)}$, w, $\overrightarrow{(w, v)}$, $succ_v(e)$. Thus we can safely apply the algorithm by starting in vertex v by incoming edge with label $pred_v(e)$. When the agent enters the vertex w via the edge e the next time, more than two vertices would be processed and this case is already solved.

  If the neighbouring vertex p of vertex v such that $pred_v(e) = \pi_v((p, v))$ has degree 1, we can RH-traverse back in the same manner to find some neighbouring vertex with degree of at least 2. The case when such vertex does not exist is discussed later.

- If no vertices were proceeded before, we RH-traverse back in the same manner as in the previous paragraph and find a neighbouring vertex of degree at least 2 that is in the kernel cycle.

The only remaining problem in this treatment is the case when there are no neighbouring vertices of degree at least 2. In such a case, the backwards traversal searching for neighbours of vertex v of degree at least 2 lying on the kernel cycle would fail.

Then, the topology of the graph is a star (see Figure 1.1.2) and this can be easily detected at the beginning of the modified algorithm *MergeCycles*. The pseudocode that deals with the star topology is shown at the end of this section (function *isStar*).

Now, we will summarize the modification of the algorithm *MergeCycles*. First of all, the agent checks whether the input graph does not have the topology of a star. The star topology of graph with arbitrary witness ordering forms one witness cycle that is also our kernel cycle and therefore no modifications have to be done. When the topology of the graph is not a star, the agent can count in its state whether zero, one or more vertices were processed. When the agent enters a vertex v via edge $e = (v, w)$, it checks the degree of the other vertex (w) incident with the incoming edge e. If the degree of this vertex is one and the number of processed vertices is less than two, the agent sets the incoming edge to $pred_v(e)$ and continues checking. As the graph does not have the star topology, the agent checking the predecessing edges will eventually find one incident with a vertex of degree at least two.

If the degree of w is one but the number of processed vertices is at least two, the vertex v was already processed and neither the rule *Merge3* nor *EatSmall* can be applied here and thus the algorithm simply continues via the edge $succ_v(e)$. In the case when the degree of w is at least two, the neighbouring vertices of v that have degree one and are not in the kernel cycle yet are merged into the kernel cycle by function *addDegreeOneNeighboursToKernelCycle* and then, by the same manner as in the original *MergeCycles* algorithm, rules *Merge3+* and *EatSmall+* are applied (the neighbouring vertices with the degree 1 are skipped during the processing of edges).

The following pseudocode deals with the one degree neighbouring vertices different from the vertex w. (`succ_v(e)` and `pred_v(e)` denote the operations $succ_v(e)$ and $pred_v(e)$)

```
function addDegreeOneNeighboursToKernelCycle in vertex v via edge e
put pebble to vertex v;
set all outer labels to 1;
set the outer label of the edge e to 2;
set e = succ_v(e);
rotate local orientation so that the inner label of e is 1;
if (outer label of e is 2)
  take pebble from vertex v;
  exit;
check the degree of vertex u, e=(u,v)
if (degree of u > 1)
  goto 4;
```

```
12: RH-traverse via e,
    during the RH-traversal check if the outer label for
    incoming edge to vertex v is 2,
    terminate the RH-traversal in the vertex v if the inner
    label of the incoming edge is 1;
13: if (during the RH-traversal no incoming edge to vertex v had outer
    label 2)
    // the neighbouring vertex of vertex v is not in the kernel cycle
    // and has degree 1
14:   apply rule EatSmall in the vertex v, (x = pred_v(e), z = e, or
      x = e, z = succ_v(e)),
      x = the edge with the outer label 2
   // the incoming was not changed and thus the inner label is 1
15: goto 4;
```

The following pseudocode deals with the network of star topology: (`succ_v(e)` denotes the operation $succ_v(e)$, `pi_v(e)` denotes $\pi_v(e)$)

```
function ifStar in vertex v via edge e
1:  rotate the local orientation so that pi_v(e) = 1;
2:  check the degree of the neighbouring vertex u, e = (v,e);
3:  if (the degree of vertex u > 1)
4:    if (pi_v(e) = 1)
5:      exit "NO";
6:    else
7:      e = succ_v(e);
8:      goto 4;
9:  e = succ_v(e);
10: if (pi_v(e) = 1)
11:   exit "YES";
12: goto 2;
```

When all degree 1 neighbouring vertices are in the kernel cycle, we can simply ignore them while processing the edges in functions *Merge3+* and *EatSmall+* by adding the following code before the line `10` in both functions:

```
9.1: traverse via edge e to vertex u
9.2: check whether the degree of u is at least 2 and remember the answer in the
     state
9.3: traverse back to v via incoming edge to u
9.4: if (degree of vertex u is 1)
9.5:   goto 9;
```

Lemma 5.4.1 *Let $G = (V, E)$ be a simple connected undirected graph. Let $v \in V$ be a vertex with degree d_v, let e be the incoming edge through which the agent RH-traversed into v. Let $u_1, \ldots, u_n \in V$ be the vertices of degree 1 that are not in the kernel cycle. Let $e_1, \ldots, e_n \in E$ be the edges $e_i = (v, u_i)$, $i \in \{1, \ldots, n\}$, $n < d_v$. Then, the function addDegreeOneNeighboursToKernelCycle connects the vertices $u_1, \ldots, u_n$ to the kernel cycle and the agent terminates the execution of this function in the vertex v with the incoming edge e.*

Proof: Straightforward from the pseudocode and the detailed discussion above. □

Lemma 5.4.2 *Let $G = (V, E)$ be a simple undirected connected graph, denote $N = |V|$. Let $v \in V$ be a vertex. Let the time complexity of one application of rule EatSmall be $O(d)$. Then, the time complexity of the function addDegreeOneNeighboursToKernelCycle applied in the vertex v is $O(N^3 d)$.*

Proof: For each neighbouring vertex of degree 1 the check whether it is in the kernel cycle is performed. The check for the membership in the kernel cycle takes $O(N^2)$ steps as that is the maximal length of the kernel cycle. The vertex has at most $O(N)$ incident edges and thus at most $O(N)$ neighbouring edges can have degree 1. The entire check for all neighbouring vertices takes at most $O(N^3)$ steps. In each case the rule *EatSmall* will be applied and thus the lemma holds. □

Lemma 5.4.3 *Let $G = (V, E)$ be a simple connected undirected graph. Let $v \in V$ and let $e \in E$ be an edge incident to the vertex v. Let the function isStar be performed by the agent in the vertex v and with the incoming edge e. Then the function terminates in vertex v and incoming edge e with the positive result when the graph G has star topology and the vertex v is the center of this star and with negative result otherwise.*

Proof: It is obvious from the pseudocode. □

Lemma 5.4.4 *Let $G = (V, E)$ be a simple undirected connected graph, let $N = |V|$. The time complexity of the function isStar applied in vertex $v \in V$ is $O(N)$.*

Proof: Straightforward from the pseudocode. □

5.5 Putting pieces together

In this section we will summarize the modifications of the algorithm *MergeCycles* to the algorithm with constant memory for the agent – *MergeCycle+*. The only remaining part in which the modified algorithm *MergeCycles* discussed in Section 5.4 oversteps the constant memory is the detection of the termination.

The algorithm computes the length of the kernel cycle and the number of vertices in which neither the rule *Merge3KC* nor *EatSmallKC* can be applied. If neither rule can be applied in any vertex of the kernel cycle, the algorithm terminates. The length of the kernel cycle is $O(N^2)$ (N is the number of vertices in the graph) and thus the counter for the length requires $O(\log N)$ memory. We can get rid of this counter and then no lengths will be computed.

On one hand, this will solve our problem with $O(\log N)$ memory, but on the other hand the termination detection will not be possible.

At this point we have two possibilities:

1. The termination of algorithm will not be considered. Then, it follows from the description of the modified algorithm *MergeCycle+* the agent can pass through the initial vertex backwards only once. The observer in the initial vertex can notice this situation and afterward the agent starts the standard RH-traversal by means of the *MergeCycles* algorithm, the RH-agent can be inserted to the initial vertex and start the RH-traverse the same edge as the precomputation was started. Then, the RH-agent can traverse in the graph, following the agent that does the precomputations. In such a case, the RH-agent must not overrun the precomputing agent (this can be handled by checking the position of the pebble and by the assuming that the channels used in the network are FIFOs). By Lemma 3.5.10, the rules are applied once in every a processed vertex and therefore the RH-agent can traverse beyond the precomputing agent.

   The case when the agent performing the precomputation does not traverse backwards at all can be easily adapted to the above idea.

2. The second approach is to allow some additional memory in the graph for the termination detection. Obviously, by Lemma 3.5.10 marking an edge in the kernel cycle (e.g. the initial edge that was used in the beginning of the RH-traversal) solves the problem of the termination detection. Marking of the edge in the kernel cycle can be replaced by two pebbles/bits in two consecutive vertices of the kernel cycle with same result.

3. An observer able to see the global situation can be present in the network. He can then terminated the algorithm when he sees that the final kernel cycle has been created and it passes through all the vertices.

Let $G = (V, E)$ be a simple connected undirected graph where the algorithm will be executed. Let $v \in V$ be a vertex in which the algorithm starts and let d_v be its degree.

The description of the algorithm *MergeCycle+* using constant memory follows:

1. check whether the vertex v is the center of the star (in other words, run function *isStar* in v). If so, terminate – any local orientation forms the kernel cycle. If not, the incoming edge is set to the one with label 1.

2. if $d_v = 1$, traverse via the only incident edge and check whether the neighbour is the center of the star (in other words, run the function *isStar*). If so, terminate – any local orientation forms a kernel cycle. If not, return back to v via the incoming edge.

3. Now we have confirmed that the graph does not have the star topology. The agent is located in vertex v and the incoming edge is set to the one with label 1. Now, the altered variant *MergeCycle+* of the original algorithm *MergeCycles* can be executed (pseudocode of the algorithm *MergeCycle+* follows).

4. Before the agent starts checking whether the rules *Merge3+* or *EatSmall+* can be applied in the vertex, the pseudocode *addDegreeOneNeighboursToKernelCycle* has to be performed. Therefore, the degree 1 neighbours of the processed vertex are added to the kernel cycle and then the functions *Merge3+* and *EatSmall+* can be executed safely.

5. The algorithm continues the same way as the algorithm *MergeCycles* by applying rules *Merge3+* and *EatSmall+* and then moves to the successive vertex.

The pseudocode of the algorithm *MergeCycle+* follows. (`succ_v(e)` and `pred_v(e)` denote the operations $succ_v(e)$ and $pred_v(e)$ respectively)

```
function RHtraverse from vertex v via edge e
1:  if (degree of v is 1 and the number of processed vertices < 2)
2:    set e = pred_v(e);
3:    RHtraverse from v via edge e;
4:  else
5:    addDegreeOneNeighboursToKernelCycle;
6:    while rule Merge3+ can be applied in v do
7:      apply Merge3;
8:    od
9:    apply rule EatSmall+ in vertex v if possible;
10:   if (rule EatSmall+ or Merge3+ was applied)
11:      goto 6;
12:   else
13:     increase the number of processed vertices which
        is remembered in the state of the agent,
        if this number is greater than two, set it to two;
14:   RHtraverse via edge succ_v(e);

function start in vertex v
15: if (the answer of isStar(vertex v, edge with label 1) is "YES")
16:   exit;
17: traverse via edge with label 1 to some vertex u (incoming via edge e);
18: if (the answer of isStar(vertex u, edge e) is "YES")
19:   exit;
20: traverse via edge e back to vertex v;
21: set in the state that zero vertices were processed;
22: RHtraverse via edge with label 1;
```

As the agent cannot stop the traversal and executes the algorithm *MergeCycle+* forever, edges with label 1 cannot be rotated so that they will all be arcs of the kernel cycle. Whenever the agent makes an attend to use rule *Merge3+* or *EatSmall+*, it makes some rotations of local orientations and destroys the setting where edges with label 1 are arcs of the kernel cycle. Therefore it does not have to hold that an RH-agent can start an RH-traversal by using the edge with the label 1.

The question of termination detection in constant memory is still open and we are able to solve it only by increasing the amount of memory available to the algorithm as it is discussed above.

5.6 Proofs of correctness and complexity

Theorem 5.6.1 *The algorithm MergeCycle+ creates the witness cycle that is passing through all vertices of a simple connected undirected graph.*

Proof: It is obvious from the descriptions in Sections 5.2, 5.3, 5.4 and 5.5. □

Note that most of the statements from Chapter 3 can be re-stated in a very similar way for the algorithm *MergeCycle+*.

Theorem 5.6.2 *The memory complexity of the algorithm MergeCycle+ is one pebble and* $O(1)$ *for the agent.*

Proof: The agent uses one incoming edge and one pebble during the execution of the algorithm *MergeCycle+*. The incoming edge is not included in the total memory complexity as it is necessary, as the problem would be unsolvable without it. In addition to that we use six global variables x, y, z, x_1, x_2, x_3 that are only used for the application of the rules *Merge3+* and *EatSmall+*. The agent does not use these variables for temporary calculation and it is only able to write the label of the traversed edge to the chosen variable (in constant memory). Therefore this is not counted as the agent's memory complexity. The termination detection is not solved yet and therefore the only memory the agent uses is a finite number of states and a pebble. □

Chapter 6

Conclusion

In this thesis we have studied the question of using the local orientation of edges in vertices of a simple connected undirected graph for storing information.

First, we designed an algorithm for precomputing the edge labelings in order to obtain a very special ordering which allows the traversal of the graph to be performed by a finite automaton with a few states, which does not use any additional memory, nor does it store any informations in the graph itself. This finite automaton (RH-agent) obeys the following rule *"Start by taking the edge with the label 1. Then, whenever you enter a node, continue by taking the successor edge (in local orientation) to the edge you arrived through."*

In order to minimize the memory requirements, the precomputations are performed by an agent that traverses the graph and is able to change labeling of the edges in vertices it visits. In Chapter 3 we show a polynomial algorithm for this precomputation that requires one pebble for marking the vertices and $O(\log N)$ additional memory (N denotes the number of vertices in the graph, M denotes the number of edges in the graph).

The modification done in Chapter 5 tries to decrease this memory complexity down to a constant. This is achieved by changing the labelings in a specific way. As a result, we obtained a very memory efficient algorithm for all the precomputations. The only disadvantage is that this algorithm is not able to terminate without using an extra bits of memory.

The aim of the algorithm that is presented in Chapter 4 is to find a spanning tree of the graph by setting the edges with label 1 in local orientation to be the links to the parent in the spanning tree (the label 1 of the edge incident to the root vertex leads to one of its direct children in the spanning tree).

This goal is investigated in a local manner by creating an algorithm for the agent that is able to exchange labels of two incident edges of a vertex. The designed algorithm is an extension of the algorithm presented in Chapter 3. The resulting polynomial algorithm for finding the spanning tree needs one pebble and $O(\log N)$ memory for the agent.

There are many directions in which this work can be extended. The problem of termination of the algorithm from Chapter 5 without using any extra memory is still open. Solving this problem could help us decrease the memory used by the agent in the algorithm for finding the spanning tree. Then, it would not be difficult to modify this algorithm to decrease the memory complexity for the agent to a constant.

A question that is very often open for algorithms like this one is the time efficiency. All algorithms designed in this thesis are polynomial-time (their complexity is $O(N^5)$). The question is whether this bound can be improved upon and what (if any) would be the price we would have to pay.

In the algorithms from Chapters 3 and Chapter 5 we try to use as little memory as possible, but then the resulting (kernel) cycle where the RH-agent will traverse can be unnecessarily long. The length of such cycle can be $O(M)$, which is, for dense graphs, equal to $O(N^2)$. It is open whether this length can be decreased and what would be the increase in time and memory complexity. It is obvious that for obtaining a cycle of length $O(N)$, only a very small subset of the edges from the graph can be used and one needs to find the criterion for selecting this subset.

A completely different view and maybe also the solution can be obtained by considering dynamic changes of the topology. Then, it is an open problem whether it is possible to react on the topology changes in such a model and what is the best approach.

From a more general point of view, another challenging task is to explore the problem in a model where more agents are available. Then, many different questions arise: What is the balance between the number of the agents used for precomputation and the efficiency of the algorithm? Can this problem be solved better with more agents? What is the best way of the communication they can use?

Bibliography

[ABRS99] Baruch Awerbuch, Margrit Betke, Ronald L. Rivest, and Mona Singh. Piecemeal graph exploration by a mobile robot. *Information and Computation*, 152(2):155–172, 1999.

[AH97] Susanne Albers and Monika R. Henzinger. Exploring unknown environments. pages 416–425, 1997.

[Ark90] Ronald C. Arkin. Integrating behavioral, perceptual, and world knowledge in reactive navigation. *Robotics and Autonomous Systems*, 6:105–122, 1990.

[BFR+98] Michael A. Bender, Antonio Fernández, Dana Ron, Amit Sahai, and Salil Vadhan. The power of a pebble: exploring and mapping directed graphs. pages 269–278, 1998.

[Bro90] R. S. Brooks. A robust layered control system for a mobile robot. pages 204–213, 1990.

[BRS94] Magrit Betke, Ronald L. Rivest, and Mona Singh. Piecemeal learning of an unknown environment. Technical Report AIM-1474, 1994.

[BS94] Michael A. Bender and Donna K. Slonim. The power of team exploration: two robots can learn unlabeled directed graphs. In *Proceedings of the 35rd Annual Symposium on Foundations of Computer Science*, pages 75–85. IEEE Computer Society Press, Los Alamitos, CA, 1994.

[CFI+05] Reuven Cohen, Pierre Fraigniaud, David Ilcinkas, Amos Korman, and David Peleg. Label-guided graph exploration by a finite automaton. In *ICALP*, pages 335–346, 2005.

[DFKP02] K. Diks, P. Fraigniaud, E. Kranakis, and A. Pelc. Tree exploration with little memory, 2002.

[Die00] Reinhard Diestel. *Graph theory.* Springer-Verlag, New York, 2 edition, 2000.

[DJSS05] Stefan Dobrev, Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Finding short right-hand-on-the-wall walks in graphs. In *SIROCCO*, pages 127–139, 2005.

[DP90] X. Deng and C. H. Papadimitriou. Exploring an unknown graph. volume 1, pages 355–361, 1990.

[FIRT05] Pierre Fraigniaud, David Ilcinkas, Sergio Rajsbaum, and Sbastien Tixeuil. Space lower bounds for graph exploration via reduced automata. In *Structural information and communication complexity*, pages 140–154, Berlin, 1 2005. Springer.

[GKM+07] Leszek Gąsieniec, Ralf Klasing, Russell Martin, Alfredo Navarra, and Xiaohui Zhang. Fast periodic graph exploration with constant memory. In *Proceedings of the 14th Colloquium on Structural Information and Communication Complexity (SIROCCO 2007)*, Lecture Notes in Computer Science. Springer Verlag, 2007. to appear.

[Ilc06] David Ilcinkas. Setting port numbers for fast graph exploration. In *SIROCCO*, pages 59–69, 2006.

[Kou] E. Koutsoupias. Result reported in X.Deng, C. H. Papadimitriou. Exploring an unknown graph. Revised version. Proc. 31st Symp. on Foundations of Computer Science, 298-303,1991.

[Kwe97] Stephen Kwek. On a simple depth-first search strategy for exploring unknown graphs. In *Workshop on Algorithms and Data Structures*, pages 345–353, 1997.

[Par00] Lynne E. Parker. Current state of the art in distributed autonomous mobile robotics. In George Bekey Lynne E. Parker and Jacob Barhen, editors, *Distributed Autonomous Robotic System 4*, pages 3–12. Springer-Verlag, Tokyo, October 2000. ISBN: 4-431-70295-4.

[PY91] Christos H. Papadimitriou and Mihalis Yannakakis. Shortest paths without a map. *Theor. Comput. Sci.*, 84(1):127–150, 1991.

Abstrakt

Uvažujme sieť reprezentovanú ako jednoduchý, súvislý, neorientovaný graf s N vrcholmi. Vrcholom grafu nie sú priradené žiadne identifikátory (a teda vrcholy nie sú rozlíšiteľné z pohľadu vnútra grafu), no požadujeme, aby hrany incidentné s vrcholom v mali priradené z pohľadu vrcholu jednoznačné identifikátory z intervalu 1 a d_v (vrátane). Toto priradenie nazývame lokálna orientácia.

Naším hlavným cieľom je, aby agent s čo najmenším množstvom pamäte bol schopný prejsť cez všetky vrcholy takéhoto grafu. Ukazujeme, že pri vhodnom predpočítaní a zmenách v lokálnej orientácii je túto úlohu schopný splniť aj veľmi jednoduchý agent používajúci iba pravidlo pravej ruky, tzv. RH-agent. RH-agent je konečný automat s veľmi malým počtom stavov, ktorý sa riadi nasledujúcim pravidlom: „*Začni použitím hrany s číslom 1. Ďalej pokračuj vždy hranou nasledujúcou (v zmysle lokálnej orientácie) po hrane, ktorou si prišiel do vrcholu.*“

Na zmenu lokálnych orientácií v predpočítavacej fáze sme navrhli algoritmus pre agenta, ktorý toto predpočítavanie vykoná. Tento agent vykonáva zmeny v lokálnej orientácii výmenou označení dvoch susedných hrán v danom vrchole. Cieľom predpočítavania je upraviť lokálne orientácie tak, aby algoritmus RH-agenta zabezpečil, že sa ten dostane do všetkých vrcholov. Ukazujeme polynomiálny algoritmus, v ktorom je potrebné na predpočítavanie použiť jeden pebble a $O(\log N)$ pamäte pre agenta. Taktiež ukazujeme modifikáciu tohto algoritmu, v ktorom je pre veľmi podobnú predpočítavaciu fázu potrebný jeden pebble a iba konštantná pamäť pre agenta. V tomto prípade však zatiaľ nie je vyriešený problém zastavenia, a preto predpočítavateľ nevie svoj výpočet ukončiť bez ďalšej pamäte.

Ako jeden z príkladov použitia nášho algoritmu sme definovali problém nájdenia zakorenenej kostry v grafe. Náš algoritmus určený pre agenta-predpočítavateľa vytvára kostru grafu zmenami v lokálnej orientácii tak, že hrana s číslom 1 ukazuje k otcovi v kostre (hrana s číslom 1 v koreni kostry ukazuje na priameho potomka koreňa). Navrhnutý algoritmus je polynomiálny a potrebuje jeden pebble a $O(\log N)$ pamäte pre agenta.

Kľúčové slová: Mobilné výpočty, distribuované algoritmy, zmeny lokálnej orientácie