Efficient Periodic Graph Traversal on Graphs with a Given Rotation System

by

Xiao Luo

B.Sc., Simon Fraser University, 2015

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
Department of Mathematics
Faculty of Science

SIMON FRASER UNIVERSITY
Summer 2018

Approval

Name:	Xiao Luo
Degree:	**Master of Science (Mathematics)**
Title:	***Efficient Periodic Graph Traversal on Graphs with a Given Rotation System***
Examining Committee:	**Chair:** Nathan Ilten Assistant Professor

Ladislav Stacho
Senior Supervisor
Associate Professor

Jan Manuch
Co-Supervisor
Science Researcher
Department of Computer Science
University of British Columbia

Tamon Stephen
Internal Examiner
Associate Professor

Date Defended: 22 Aug 2018

Abstract

We consider *periodic graph traversal* in anonymous undirected graphs by a finite state Mealy automaton (agent). The problem is to design an automaton A and a port labeling scheme L such that A (using L) performs on any undirected graph an infinite walk that periodically visit all vertices. The goal is to minimize the revisit time of any vertex over all graphs (traversal period) π.

If the labeling scheme L is given by an adversary, it has been shown by Budach [6] that no such finite automaton A exists. If L does not need to be a permutation at every vertex, one can easily encode a spanning tree and A can perform a traversal with period $2n-2$ even if it is an oblivious agent. The problem is difficult when L is limited to be a permutation.

The best known upper bound on the traversal period in such a case is $3.5n-2$ by Czyzowicz et al. [11], where n is the number of vertices. It is an open problem whether it is possible to achieve traversal period of $2n-2$.

In this thesis, we answer this question affirmatively under the assumption that the input graph G is given with a rotation system, where a rotation system of a graph is given by lists of local orders of edges incident to each vertex the graph.

Keywords: graph exploration; periodic graph traversal; perpetual traversal; local orientation; finite state automaton; rotation system; period 2n-2; master thesis

Dedicated to

my wife, Sarah,
my son, Royson,
and my doggy, Romeo,
who bring love and joy to my life.

Acknowledgements

This research started in Dr. Ladislav Stacho's office three years ago when three mathematicians met together and began drawing things on board. I firstly give my appreciation to Dr. Jan Manuch, Dr. Stefan Dobrev as their ideas and insight are important contributors to this thesis.

My next appreciation is to Dr. Ladislav Stacho. As my professor in my first upper level math course "Linear Optimization" six years ago, then as my supervisor and mentor from three years ago until now, he has had so much influence on me, in the aspect of teaching me knowledge, shaping my learning attitude and more importantly, showing me how a mathematician thinks and works. The achievement comes from diligence, persistence and determination. I finally in the last year gave up my old habits, and learned do the work. Then the progress was accelerating. I will keep the two Ladislav's most saying words as the maxim for my life, "Is it really hard?" and "What is it exactly about?". I will ask myself the two questions whenever I encounter any difficulty in my future. A mathematician will not think about the hardness; a mathematician will always ask questions and do the work, focus on detail one at a time, and pursuit the perfection in his heart.

Additionally, I would like to give my appreciation to Dr. Tamon Stephen, Dr. Zhaosong Lu and Dr. Abraham Punnen, as my professors when I studied optimization in operational research. Since then, I have learned broad knowledge in computational mathematics, computing science and statistics, which are turned to be very beneficial to me.

Finally, I want to thank Dr. Jennifer Jing Zhao. She encouraged me to study mathematics when I walked into a crossroad in my study career. Today, we live in an algorithmic world. People are striving for optimization anywhere, in any form. And mathematicians are interested in solving problems. What a good age for us!

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Fundamental Concepts

1.1.1 Graphs

A *simple (directed) graph* is a two-tuple consisting of a *vertex set* $V(G)$ and an *edge set* $E(G)$. The elements of this relation are called *edges* and two related elements of $V(G)$ are called *endpoints of the corresponding edge*. In this thesis, $V(G)$ will be a finite set. When u and v are the endpoints of an edge, they are *adjacent* and are *neighbors*. We also say an endpoint is *incident* to its corresponding edge, and vice versa. Given a directed graph G and an ordered pair $(u, v) \in E(G)$, u is *tail* of the edge, and v is the *head*. We say u is *adjacent* to v but v is not adjacent to u.

A *path* is a simple graph whose vertices can be ordered so that two vertices are adjacent if and only if they are consecutive in this order. The number of edges in a path is its *length*. If u is the first vertex of a path and v is its last vertex, we say the path is a $u - v$ *path*. A *cycle* is a $u - v$ path of length at least two with added edge joining u and v. The length of the cycle is the number of edges on the path.

A *subgraph* of a graph G is a graph H such that $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. We then write $H \subseteq G$ and say that G contains H. A graph is *connected* if each pair of vertices in G belongs to a path; otherwise, G is disconnected.

The *degree* of vertex v in a graph G, written $d_G(v)$ or $d(v)$, is the number of edges incident to v. The maximum degree $\Delta(G)$ and the minimum degree $\delta(G)$ of a graph G are defined as: $\Delta(G) = \max_{v \in V(G)} d(v)$ and $\delta(G) = \min_{v \in V(G)} d(v)$. A graph G is *regular* if $\Delta(G) = \delta(G)$. The graph is k-regular if the common degree is k. The *neighborhood* of a vertex v, written as $N(v)$, is the set of vertices adjacent to v. For any vertex v in a directed graph $\vec{G}$, the number of head ends adjacent to v is called the indegree of v and the number of tail ends adjacent to v is called the outdegree of v.

Given a connected graph G, a *tree* is a connected subgraph of G without a cycle. A *leaf* is a vertex of degree 1 in a tree. A *spanning subgraph* of G is a subgraph with vertex set

$V(G)$. A *spanning tree* of G is a spanning subgraph of G that is a tree. Given graph G and its spanning tree T, a vertex v is called *saturated* if $d_T(v) = d_G(v)$. If an edge in G also belongs to T, we call it a *tree edge*; otherwise we call it a *non-tree edge*.

If G has a $u - v$ path, then the *distance* from u to v, written as $d_G(u, v)$ or $d(u, v)$, is the least length of a $u - v$ path. The *diameter* of connected graph G is $\max_{u,v \in V(G)} d(u, v)$.

At every vertex we assign different integer labels from $\{1, \ldots, d(v)\}$, called *port numbers* or *labels*, to edges incident to that vertex. Since every edge is incident to two vertices, it has two port numbers (labels), one at each endpoint. These two port numbers have no relation with each other.

In this thesis, we will consider simple connected graphs unless we specify otherwise.

1.1.2 Finite Automaton

A *finite Mealy automaton*, see [23], is defined by the 5-tuple

$$M = (Q, q_0, \Sigma, \Lambda, \delta)$$

where

- Q is a finite set of states,
- $q_0 \in Q$ is the initial state,
- Σ is a finite set of symbols called the input alphabet,
- Λ is a finite set of symbols called the output alphabet,
- $\delta : Q \times \Sigma \to Q \times \Lambda$ is mapping pairs (current state, input symbol) to pairs (next state, output symbol). This function is called the transition function.

The computation of such automaton starts in the initial state q_0 while reading first symbol say s_0 from an input, i.e., in a configuration (q_0, s_0). In each step of computation the transition function δ is applied to the current configuration producing a new configuration. The computation terminates when the input string is read completely.

In this thesis, we will refer to a finite Mealy automaton as an *agent*. The agent will traverse a given graph G in the following way. The transition function δ of an agent maps the current state and the input to a new state and an output. For the agent used in perpetual graph traversal, the input is the labeling sequence at a current vertex (a sequence of port numbers assigned to the edges incident to the vertex in counter-clockwise order) together with the incoming port number. The output is the outgoing port number which will be used by agent to leave the vertex. Refer for details to Section 2.1. Every transition from current configuration to the next is called a *step*.

1.2 Periodic Graph Traversal

Exploration of graphs is one of the fundamental problems in graph theory and communication networks, with applications in autonomous searching known and unknown environments, e.g., indexing data sets (Internet, file systems, etc.) and monitoring (hazardous environments, computer networks). In some specific tasks of graph exploration, every vertex has to be visited or checked periodically. This field of research is called *Periodic Graph Traversal* problems and it is particularly useful for network maintenance, data searching, etc. In this thesis, we will consider the following *graph traversal problem*.

In the graph traversal problem, we are given a simple connected undirected graph on n vertices, which is *anonymous*, that is, vertices are indistinguishable from each other, they cannot be marked or labelled. The goal is to design an agent that will visit every vertex of the graph regularly. To simplify the model, we ignore the time spent by the agent when doing local processing at a vertex (reading the permutation of port numbers on incident edges, scanning for the next tree edge, etc.) Hence, we will concentrate only on moves along edges, which we will refer to as steps. The *period*, denoted by $\pi(n)$, is the maximum number of steps between two consecutive visits at any vertex by the agent in a given graph. The period is used to measure the time efficiency of the periodic traversal.

Our first result in review is a probabilistic approach to traversal in [17]. It can be used to design an agent which revisits every vertex in expected time $O(n^3)$ regularly.

Theorem 1. *[17] The expected time for a random walk to visit all n vertices of a connected graph is at most $\frac{4}{27}n^3 + o(n^3)$.*

Proof. (sketch) We only show here the bound of $O(n^3)$. Let $G = (V, E)$ be a graph with n vertices and m edges. An agent will perform a random walk on vertices of G, where at each step the agent at vertex v has equal probability moving from v to any of its neighbors. Let u, v be any two vertices in $V(G)$. Define the *hitting time* $H(u, v)$ as the expected number of steps the agent walks from u to v, where $u, v \in V$. Define the *commute time* $C(u, v)$ as the expected number of steps the agent walks from u to v, then walks from v back to u. Clearly, $C(u, v) = H(u, v) + H(v, u)$. Define the *cover time* $EC(v)$ as the expected number of steps the agent takes starting at vertex v until it visits all other vertices. For a graph G, its hitting time is defined as $H(G) = \max_{u,v \in V(G)} H(u, v)$, its commute time is defined as $C(G) = \max_{u,v \in V(G)} C(u, v)$, and its cover time is defined as $EC(G) = \max_{v \in V(G)} EC(v)$. The *cyclic cover time* of G, denoted by $ECC(G)$, is the expected number of steps an agent takes to visit all vertices of the graph in a pre-specified cyclic order, that minimizes the number of steps. That is,

$$ECC(G) = H(v_1, v_2) + H(v_2, v_3) + \cdots + H(v_{n-1}, v_n) + H(v_n, v_1)$$

where $(v_1, v_2, \ldots, v_n)$ is a permutation of vertices of G that minimizes the sum. Clearly, $EC(G) < ECC(G)$.

There is a well known correspondence between random walks and electrical resistance [8]. View graph G as an electrical network with unit resistors as edges. For instance, consider each edge of G as an electrical resistance of 1 ohm. The *effective resistance* between vertex u and vertex v, denoted by $R(u, v)$, is the voltage that develops in v when a current of 1 amp is injected into v and the vertex u is grounded. Chandra et al. [8] have shown some important results under this analogy.

$$C(u, v) = 2mR(u, v)$$

where m is the number of edges in G.

It follows from the laws of electrical physics that the effective resistance between two endpoints of an edge of G is at most 1. Let T be a spanning tree of G. Define $R(T)$, the effective resistance of T, as the sum of the effective resistance along its edges. Then, $R(T) \leq n - 1$, since T has $n - 1$ edges. Let R_{span} denote the effective resistance of the minimum resistance spanning tree of G. It is proved by Chandra et al. [8] that

$$ECC(G) \leq 2mR_{span}$$

Observe that $2m \leq n(n-1)$ and $R_{span} \leq n - 1$. Then,

$$\begin{aligned} ECC(G) &\leq 2mR_{span} \\ &\leq n(n-1)(n-1) \\ &= n(n-1)^2 \end{aligned}$$

This gives an upper bound of $O(n^3)$. Further work can be done to improve the bound to $\frac{4}{27}n^3 + o(n^3)$ by showing there is a tradeoff between m and R_{span}, and that they cannot both attain their maximum value at the same time. □

If the edges incident to a vertex are not distinguishable, it cannot be guaranteed that the agent traverses the whole graph even for a star graph with three leaves. To overcome this trivial barrier, we will assume that the graph is pre-labelled with port numbers and the agent visiting a vertex v can read the port numbers on edges incident to v at the v-ends, cf. Figure 1.1. If these port numbers are assigned by an adversary, Budach [6] proved in 1978 that for any agent there exists a graph which can have port numbers assigned in such a way that the agent fails to traverse all its vertices. Later, Rollik [26] proved that for any finite set of automata, there is a planar graph which the automata together cannot explore (visit each vertex at least once). This result is improved by Cook and Rackoff [10], in which they introduced the concept of Jumping Automaton for Graphs (JAG). A JAG is a finite

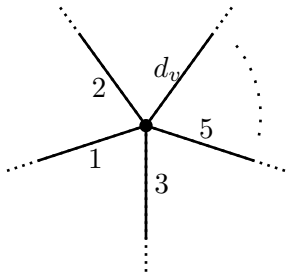

Figure 1.1: An example of port labelling at a vertex v

team of finite automata that can permanently cooperate and that can use "teleportation" to move from their current location to the location of any other automaton. In [10], it is shown that no JAG can explore all graphs.

A recent paper by Fraigniaud et al. [20] gives a lower bound on memory bits required for a traversal on anonymous d-regular d-edge-colorable graphs. As the authors in [20] note any proper d-edge-coloring of d-regular graph corresponds to a port-labelling in which the two port numbers on each edge are identical. They use this correspondence in their proof of the next theorem.

Theorem 2. *[20] For any integer $d \geq 3$ and any agent with K states, there exists an anonymous planar graph G of maximum degree d with at most $K+1$ vertices that the agent cannot explore all its vertices and traverse all its edges. In particular, any agent that can explore all n vertex graphs requires at least $\lceil \log n \rceil$ bits of memory.*

Proof. (sketch) Let T_d be a d-edge-colored infinite d-regular tree. Each color represents a port label. For any given agent with K states, we let it start on any vertex u_0 in T_d. After at most K steps, the agent has been in the same state twice. Let S be the first repeated state, and let u and u' be the first two vertices where the agent was in the state S. Assume this happened at time t and t', respectively. Since the agent is in an edge-colored regular graph, the sequence of states the agent is in during traversal becomes periodic after time t, with the period $p = t' - t$. The construction of the resulting graph G will use this subgraph of T_d that the agent explores from time t to t', plus in some more difficult cases the part of T_d that is explored by the agent in next p steps must be used. In general this subgraph will be a walk. The construction is completed by creating a cycle (possibly cyclic walk) in which the agent will be trapped without ever visiting at least one vertex. For example, suppose that the first and the last edge labels on the walk are different, i.e., the agent is leaving u on edge labelled l and coming to u' on edge labelled l' from a vertex say w. Since T_d is d regular, There must be edge incident to u labeled l'. Now remove the edges (w, u') and the edge incident to u with label l' from the subgraph and add new edge (w, u) and label it l'. This is obviously a trap for the agent since it will never leave the created cycle and never visits the vertex u'. The construction is completed by making the subgraph an almost

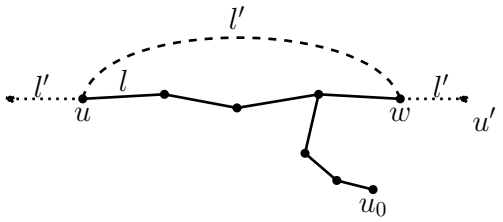

Figure 1.2: A trap subgraph is constructed by removing the two dotted edges and adding the dashed edge.

d-regular graph (the only exception will be the unvisited vertex) so that we can obtain a port labelling. □

If we can assign port numbers instead of adversary, can an agent explore given graph periodically? In the remainder of this thesis, we will use a port labelling algorithm to assign port numbers to assist the agent with the periodic traversal.

An agent is *oblivious* if it has only one state. The transition function of the oblivious agent used in [14] is very simple: arriving to a current vertex on a port i, the agent will be leaving it on the port $i+1 \mod$ the degree of the current vertex. Such a transition function is called the *Right-Hand-On-The-Wall* transition function. Dobrev et al. [14] presented a port labelling algorithm and proved that a periodic graph traversal can be achieved by an oblivious agent with a period at most $10n$ on any n vertex graph.

Theorem 3. *[14] There exists a port labelling algorithm such that an oblivious agent can traverse any simple connected undirected graph G on n vertices with a period at most* $10n$.

Proof. (sketch) For a given graph G, one can find a spanning subgraph H of G and assign directions to edges of H so that H becomes a directed closed walk $\vec{C}$ in G. The agent will strictly traverse on $\vec{C}$ and since $\vec{C}$ is a closed walk the agent can be oblivious.

The subgraph H is constructed iteratively by first adding all possible cycles of G (in any order) whose consecutive vertices are not both in the already constructed subgraph. If H is a spanning connected subgraph of G, we are done. To make H a spanning subgraph of G, we add all remaining vertices (not yet in the constructed subgraph) and all their incident edges into the subgraph. To make the subgraph connected we add additional edges bridging such components.

It can be shown that such H has at most $5n$ edges, so the length of $\vec{C}$ is at most $10n$. For an illustration of a justification of this, suppose H was connected and spanning after adding all cycles as described above. First cycle will add, say, n_1 vertices and n_1 edges into the subgraph. By the requirement on cycles added in all subsequent iterations, if they are adding, say, k new edges, they must be also adding at least $k/2$ new vertices. Therefore such H will have at most $2n$ edges as claimed. To bound number of edges of H in other cases are more involved.

To assign port numbers to edges of G, we first obtain a closed walk $\vec{C}$ (a closed Eulerian trail) from H by assigning arbitrary direction to every cycle added to H and orienting all remaining edges in both directions. Then we start assigning port numbers to edges of G first by following $\vec{C}$ and using smallest unassigned port number each time. The remaining port numbers are assigned arbitrarily. Such a port labeling will work provided $\vec{C}$ does not have edges that are oriented in both directions. In such cases some sort of reassigning of port numbers is performed. □

For an oblivious agent, in the following theorem, we have the following improvement on the upper bound, as well as first non-trivial lower bound.

Theorem 4. *[11] For an oblivious agent the upper bound for the period is* $\frac{13}{3}n - 4$ *and a lower bound is* $2.8n - 2$.

For non-oblivious agent, an obvious lower bound for period of traversal is $2n - 2$, when the graph is a n vertex tree. One might ask whether it is possible to achieve this bound. The initial bound of $10n$ on the period was improved in several papers. The common approach is similar to that of [14], one finds a subgraph of the original graph and somehow uses that subgraph in a labelling scheme and design of the agent. In particular, Ilcinkas in [22] improved the $10n$ bound to $4n - 2$ by using an arbitrary rooted spanning tree as the subgraph.

Theorem 5. *[22] For any undirected graph* G *on* n *vertices, there is a port labelling algorithm such that an agent with* 3 *states can periodically traverse the graph* G *with the period at most* $4n - 2$.

Proof. (sketch) Let T be any spanning tree of a given graph G. Choose any leaf as the root of T, denoted as r. For any vertex v, let d_v be its degree in G and let t_v be its degree in T. The port numbers are assigned by the following port labelling algorithm.

1. For any vertex $v \neq r$, assign the port number 1 to the incident edge which leads to the root, assign port numbers $2, \ldots, t_v$ to the remaining tree edges, and assign port numbers $t_v + 1, \ldots, d_v$ to the remaining non-tree edges arbitrarily;

2. For $v = r$, assign port numbers $1, \ldots, t_v$ to the tree edges and assign port numbers $t_v + 1, \ldots, d_v$ to the non-tree edges arbitrarily.

It follows from the labelling that an edge e is a tree edge if and only if at least one of its ends is labelled 1. This property can be used by an agent to perform the required traversal. The agent will start by taking an edge with the port number 1, and then it will use the Right-Hand-On-The-Wall transition function to take next edge. However it must test whether the edge it wants to take is a tree edge, or it must backtrack to the parent of the current vertex (recall T is rooted). Again, by the properties of the labelling, the edge

leaving to the parent has label 1, so backtracking is easy to perform. To test for a tree edge, the agent takes the edge and checks whether the label on its other end is 1, in which case the edge is a tree edge and agent will take it. If the label is not 1, the edge is not tree edge and the agent returns back to the current vertex and tests next edge (stopping on edge with label 1 and consequently returning to the parent of the current vertex).

The algorithm described above leads to the following bound on the period. Every tree edge will be traversed by the agent twice, and because at every vertex only one non-tree edge will be traversed (after its traversal the agent will know it needs to backtrack). The period of this traversal is $2n - 2 + 2n = 4n - 2$. □

Ilcinkas in his approach used an arbitrary spanning tree and assumes every vertex is non-saturated. If one uses a special spanning tree, the period of traversal can be further improved. Such approach was taken by Leszek et al. in [21] and resulted in a period of $3.75n$ and by Czyzowicz et al. in [11] and resulted in a period of $3.5n$.

Theorem 6. *[21] For any undirected graph G on n vertices, there is a port labelling algorithm such that an agent with* 33 *states can periodically traverse the graph G with the period at most* $3.75n - 2$.

Proof. (sketch) Given a graph G, we find a connected acyclic subgraph B, and then find a special spanning tree T based on B (B is a subtree of T). Let R, Y be two vertex sets. We call a vertex in R red and a vertex in Y yellow. Let v be any vertex in $V(G)$. Initialize $R = \{v\}$, $Y = \emptyset$, and $V(B) = R \cup Y$, $E(B) = \emptyset$. Iteratively, for any vertex u in $V(G)/V(B)$ with distance three in G to some red vertex v, add u into R, add the remaining two vertices on the $u - v$ path into Y, and add the edges on the $u - v$ path into $E(B)$. This procedure terminates when no such vertex can be found.

One can show that the acyclic subgraph B has two properties:

1. on the path between a red vertex and another red vertex closest to it, there are exactly two yellow vertices;

2. the distance between any vertex in $V(G)/V(B)$ to R is at most two. Initialize $T = (V(B), E(B))$. Then, we complete the construction of T by first making all red vertices saturated, and second connecting all remaining vertices in $V(G)$ but not in $V(T)$ to their neighbors in T. Pick any leaf r as the root of T.

Let v be any vertex in T. Assume $c_1, \ldots, c_k$ are the children of v satisfying $|T(c_1)| \geq |T(c_2)| \geq \cdots \geq |T(c_k)|$, where $T(c_i)$ is the subtree of T rooted at c_i and $|T(c_i)|$ is the order (number of vertices) of $T(c_i)$ for $i = 1, 2, \ldots, k$. The port labelling algorithm is similar to the one in Theorem 5 except instead of arbitrarily assigned port numbers at a vertex v they are assigned in an increasing order depending on the cardinality of the subtree. This enables the agent to traverse on the larger subtrees first. This together with the properties of the trees allows the agent to traverse G with period $3.75n - 2$ as claimed. □

Theorem 7. *[11] For any undirected graph G on n vertices, there is a port labelling algorithm such that an agent with finite number of states can periodically traverse the graph G with the period at most $3.5n - 2$.*

Proof. (sketch) For the proof, a new graph decomposition method called *three-layer partition* was introduced in [11]. For a given graph G, one can find three mutually disjoint vertex sets X, Y, Z and a connected cycle-free graph T_B such that:

1. $X \cup Y \cup Z = V(G)$;
2. $Y = N_G(X)$, $Z = N_G(Y) \setminus X$, where $N_G(X)$ is the set of neighbors of X in G;
3. $V(T_B) = X \cup Y$, all vertices in X are saturated;
4. edges incident to vertices in X form the edge set of T_B.

The algorithm for finding three-layer partition will successively add an unused vertex from $V(G)$ to X, and will make it saturated by adding all its incident edges to $E(T_B)$. It will be continuously updating Y and Z as above, while adding a vertex from Y or Z to X if this does not create a cycle in T_B. The algorithm terminates when there is no more vertex which can be added into X. The resulting three-layer partition (X, Y, Z, T_B) has the following properties:

1. each vertex in Y has a non-tree edge which is incident to another vertex in Y;
2. each vertex in Z has at least two neighbors in Y.

It is easy to prove above properties by contradiction. If property 1. does not hold for a vertex, this vertex must be saturated in T_B, then we can add it into X by our construction. If property 2. does not hold, let v be such a vertex in Z with one neighbor in Y. Then adding v into X and making it saturated will not create a cycle in T_B.

The spanning tree T is formed from $T(B)$ by adding every vertex in Z and one its incident edge into $T(B)$.

The port labelling algorithm is modified from previous paper [21] by a *port swap operation*. This ensures that the agent from [21] will not pay penalty steps at any vertex in X (as these are all saturated vertices) and Z. Let v be any vertex in Z and assume it will not be chosen as the root of T. From property 2, v has two neighbors, denoted by u_1 and u_2, in Y with one incident edge in T, one not in T. Let $e_1 = (v, u_1)$ be the tree edge and $e_2 = (v, u_2)$ be the non-tree edge. The port number on e_1 at $v-$end is 1. We swap 1 with the port number on e_2 at $v-$end. Now, e_1 has two port numbers not equal to 1. The agent walks on e_1 will think it is a non-tree edge and return back. This prevents the agent taking additional penalty steps at v since v has only one tree edge. To avoid the agent walking on edge e_2 and thinking it is a tree edge, if the port number on edge e_2 at u_2-end is $t+1$ (t is

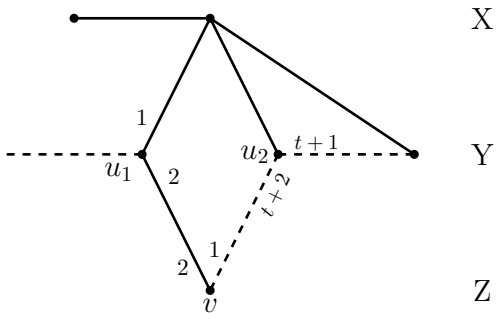

Figure 1.3: At vertex v, swap port number 1 with 2; at vertex u_2, if port number $t+1$ is on the edge (u_2, v), swap it with $t+2$.

the degree of u_2 in T), we swap it with port number $t+2$ on another non-tree edge. Such a non-tree edge exist by property 1, u_2 has a non-tree edge incident to another vertex in Y. See Figure 1.3. It can be shown that the agent can save additional portion of penalty steps thus achieving period $3.5n - 2$. □

1.3 Other Related Results

The graph exploration problem has been extensively researched under different assumptions. The first aspect of assumption is on the environment for exploration. Some papers [5, 12] assume the environment is a rectilinear region, the agent knows absolute positions of starting point s and target point t, but does not know the positions of obstacles it will encounter. The problem is to explore the unknown environment using the shortest possible total distance travelled. In particular, the objective is to minimize the ratio of the worst-case distance travelled from s to t over the optimum distance needed from s to t. In other papers, the agent is traversing an unknown graph and its movement is restricted to move along the edges. Starting from any vertex in the graph, the agent is asked to visit every vertex in the graph with a minimum number of steps.

Under the latter assumption, there are two branches of studies based on whether the graph is directed or undirected. In [1, 3, 13, 18, 19], the environment is modeled by a directed, strongly connected graph. In [1, 13], it is assumed that the agent knows all visited vertices and edges and can recognize them when encountered again; at any visited vertex, the agent knows all the unvisited edges with tails at it. An upper bound of $k^{O(\log k)} \cdot m$ is achieved by applying a divide-and-conquer approach and a greedy algorithm, where m is the number of edges in the graph and k is the minimum number of edges needed to be added to make the graph Eulerian. This result is improved to $O(k^8) \cdot m$ in [18]. The memory requirements of an agent required to perform digraphs exploration is studied in [19] and an upper bound on memory is proved to be at most $O(\log d)$, where d is the maximum degree of the digraph. Note that vertices are indistinguishable, otherwise a simple depth-first-search

can explore a graph with n vertices in $O(n \cdot m)$ memory. We also have a local assumption that the edges incident to a vertex are distinguishable, otherwise an agent cannot explore a star with three leaves. It is assumed that port numbers $1, \ldots, d_v$ can be assigned to edges incident to v at the $v-$end, where d_v is the degree of the vertex v in the graph. In the paper [3] a special additional assumption was made that the agent can drop a pebble at any visited vertex and can pick up the pebble when returns to the vertex again. This allows the agent to mark one vertex at a time. It is proved that if n is known to the agent, with only one pebble, the agent is able to explore the graph (including find the closed path which covers all the vertices and edges) in a polynomial time in terms of n and d. If n is unknown to the agent, then $\Theta(\log \log n)$ pebbles are both necessary and sufficient to perform the digraph exploration.

In remaining work, the graphs are assumed to be undirected and the agent can move in both directions of edges. For graphs with distinguishable vertices, the worst-case number of steps for traversal, which is $O(m \cdot n)$ by depth-first-search, is improved to $O(n)$ in paper [25] by monitoring the exploration in the order given by a dynamically constructed tree. In [2, 4, 15], the authors introduced the piecemeal constraint which is that the agent is required to return to its starting position every so often (for refuelling, etc.) The limitation on number of steps the agent can move in every round is $B = (2 + \alpha) \cdot D$, where $\alpha > 0$ is a constant number and D is the diameter of the graph. It is proved in [2] that the agent needs at most $O(m + n^{1+o(1)})$ steps for such piecemeal graph explorations.

1.4 Our Results

In this thesis, we study graphs with given rotation system.

Definition 1. *Rotation System: A* rotation system *of a graph is given by prescribing a local ordering of edges incident to a vertex for each vertex of the graph.*

Note that a rotation system is commonly used to describe graph's embedding into an orientable surface in a combinatorial way. Each system of such permutations gives an embedding, as each embedding is specified by a circular ordering of edges around each vertex in a counter-clockwise order as they appear on faces. In this thesis we will not use any specific properties of the embedding except the rotation system itself.

In many practical applications such a system is naturally given, for example, IO ports on a router are ordered, a cyclic order of streets at intersections, etc. See Figure 1.4 for an example of a rotation system. Graphs we consider are simple, connected, undirected, and anonymous. Moreover, at every vertex v, we assign port numbers $1, 2, \ldots, d_v$ to incident edges. The port numbers will be used by the agent. For an example of a port labeling see Figure 1.4. In previous subsection, we have summarized results on periodic traversal when no rotation system is given. We only recall here that for an oblivious agent the best

known upper bound for the period is $\frac{13}{3}n - 4$ and a non-trivial lower bound is $2.8n - 2$, see Theorem 4.

It is obvious that the minimum period of traversal on an n-vertex tree is $2n - 2$, thus an obvious lower bound of period of traversal on a general graph is $2n - 2$. In this thesis we show that this optimal period is achievable when the graph is given with a rotation system. More specifically, we answer positively the main unsolved problem whether there is a port labelling of any graph G and a corresponding agent which can perform periodic traversal on G with a period of $2n - 2$ in case when G is provided with a rotation system.

As we saw above, in the periodic graph traversal problem, special spanning tree is often constructed in order to design a traversal. In Chapter 3, we introduce a method to construct a special spanning tree that will be used by an agent.

Our main result is the following theorem, the proof of which is covered in the remainder of this thesis.

Theorem 8. *There exists a labeling scheme L and an agent (finite-state automaton) which can perform a periodic traversal with period at most $2n-2$ of any simple connected n vertex graph G given with a rotation system and labeled by L.*

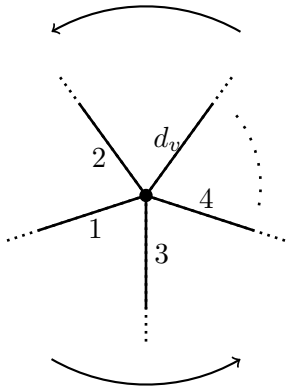

Figure 1.4: Rotation System is marked by the two arrows. The numbers represent port labelling at v.

Chapter 2

Preliminaries

2.1 Definitions and Notations

Let G be a simple connected undirected graph with a given rotation system and T be a connected spanning subgraph of G with oriented edges in such a way that every vertex has exactly one *up-tree* edge, (note that this is not necessarily rooted tree). Other edges of T are called the *down-tree* edges. Denote by $c(T, G)$ the characteristic function that assigns 1 to each edge of T, and 0 otherwise. Projecting to a vertex $v \in G$, $c(v, G, T)$ is a function assigning 1 or 0 to each port of v, depending on whether the corresponding edge is in T or not. When the context is clear, we will omit G and T and use $c(v)$ and call it a *configuration* of v. Since every vertex of G has a unique up-tree edge in T and a cyclic order on incident edges given by the rotation system, we can represent the configuration of v as a binary sequence starting with 1 which will correspond to the unique up-tree edge.

A *Labeling scheme* ls is a function that assigns to each configuration $c(v)$ of a vertex v a permutation of port numbers from 1 to d. The *labeling sequence* $l(v)$ at vertex v (as seen by the agent) is the circular permutation of permutation $ls(c(v))$, i.e., a permutation that groups all cyclic shifts of $ls(c(v))$ to one equivalence class. To represent the circular permutation $l(v)$ we will choose the permutation from the class that starts with 1. See Figure 2.1 for an example.

A vertex v is *unambiguous* if it is possible from a given labeling scheme to determine the up-tree edge and the configuration $c(v)$ from the labeling sequence $l(v)$. More precisely, a vertex v is unambiguous if the only configuration that is mapped by ls to a permutation of $l(v)$ is $c(v)$. A vertex is *ambiguous* if it is not unambiguous.

If an agent is currently at a vertex v, the *incoming port number* is the port number on the edge lastly visited by the agent at the $v-$end, i.e., the edge via which the agent moved to v, and the *outgoing port number* is the port number of the edge the agent will leave the vertex v at the $v-$end.

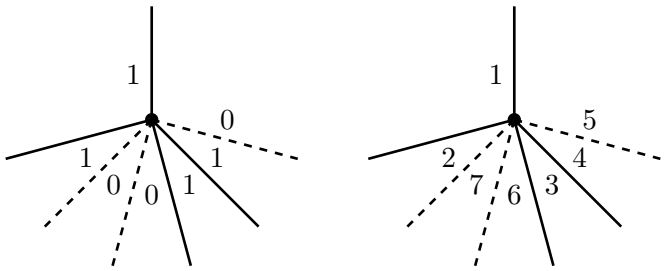

Figure 2.1: An example of configuration 1100110 and its labeling sequence 1276345 at a vertex v with the solid lines representing tree edges and the dashed lines representing the non-tree edges.

2.2 Overview of The Thesis

In order to achieve the period of $2n-2$, we first construct a special rooted spanning subgraph T of a given graph G, then use a labeling scheme to assign port numbers at every vertex such that the agent can recognize which edges are in T, and finally design a traversal algorithm for the agent so that after some preprocessing the agent can follow an Euler tour given by T and periodically visits every vertex of G with the period $2n-2$.

This approach has several challenges.

1. It is not always possible to have unambiguous configurations for small degree vertices. Therefore the agent may wrongly decode the configuration from the observed labeling sequence and may take wrong edge. However when we add another parameter which is the direction of traversal along T, ambiguous configurations become unambiguous for the agent. As the agent will have to guess the direction at the beginning, we will have to show that after finite number of moves the agent is alway able to correct its assumption if wrong.

2. Even if the labeling scheme for the configuration of a given vertex is injective, the agent must be able to decode it (or, more precisely, to determine the next edge to take) using only its limited memory (number of states). We describe a way how this can be done in $O(d)$ iterations of local computation at a vertex, where d is the degree of the vertex. Note that these are not counted towards the period.

We start with easy cases which will be covered by the following observation.

Observation 1. *There exists a labeling scheme which is injective for vertices with degree $d \geq 5$.*

Proof. As we declared earlier, every vertex has one incoming edge the up-tree edge (edge leading to a root). The configuration and labeling sequence of every vertex with $d \geq 5$ will always start with 1. For a vertex v with degree d in G, there are 2^{d-1} different possible configurations and $(d-1)!$ possible permutations of labeling sequences. We have $(d-1)! >$

2^{d-1} when $d \geq 5$. Hence, for vertices with $d \geq 5$, there are more permutations of labeling sequences than the number of possible configurations. □

It follows from Observation 1 that for the vertices with degree $d \geq 5$, problem 1 does not occur, but problem 2 does. If the agent needs to remember all the possible labeling sequences and its corresponding configurations, the size of the input alphabet is $\Delta!$. On the other hand, for the vertices of degree at most 4, the decoding can be done in $O(1)$ memory just by memorizing all pairs of labeling sequence and configurations, but the number of possible labeling sequences is insufficient to encode all possible configurations.

To cope with these problems, we use a special spanning subgraph of G as our T, instead of an arbitrary spanning tree. We also pick a special vertex as the root of T, instead of an arbitrary one. The construction of the spanning subgraph T and assigning an orientation on its edges are discussed in Chapter 3. There are two states, "up-tree" and "down-tree", denoted by $\uparrow$ and $\downarrow$ representing the current direction of the traversal by agent. Each represents the agent's assumption of traversing direction which is either going up-tree to the root or going down-tree to the leaves. The agent may change this assumption either when an inconsistency has been detected (this may happen only once), or when passing on root-path (this will be discussed later). The details of port labelling are given in Chapter 4. The transition function of the agent is discussed in Chapter 5, and the correctness is proved in Chapter 6. In Chapter 7, we conclude with a summary of our results and propose a few open questions.

Chapter 3

Constructing the tree T

Recall that G is a simple connected undirected graph with a given rotation system. Let T be a connected spanning subgraph of G. We start with some definitions.

Definition 2. *Double Rooted Tree: An acyclic graph T is called a* double-rooted tree *if it can be obtained from two disjoint rooted trees joined together by a path, referred to as* root path *between the two roots. The two roots will be considered as roots in T, and one of them will be selected as the main root. See Figure 4.7.*

Definition 3. *C-tree: T is called a* C-tree *if T consists of one cycle C and one tree attached to C. We refer to the vertex of attachment as the root of the C-tree. We call a C-tree* orientable *if in the rotation system the two cycle edges adjacent to the root are consecutive or separated by only non-tree edges. Note that if the root is of degree three in T, the C-tree is orientable. See Figure 4.6.*

Definition 4. *Eligible vertex: A vertex v is* eligible *if it is not a transit vertex (a vertex with one of the configurations (c), (d),(k), or (l) depicted in Figure 4.5), and if v is a leaf of T then must have $d_T(v) = d_G(v) = 1$, i.e., its configuration is (o) depicted in Figure 4.5.*

Lemma 1. *Any connected graph G contains either*

- *a Hamiltonian cycle, or*
- *a spanning orientable C-tree T with root r so that $d_T(r) = 3$, or*
- *a double-rooted spanning tree T in which the two roots are eligible vertices.*

Moreover, the spanning subgraph T can be chosen so that there is no edge of G joining two leaves of T.

Proof. Suppose G is not Hamiltonian. Let T be a spanning tree of G with as few leaves as possible.

If T has no branch vertex, then it is a Hamiltonian path. There are three possibilities:

- both leaves of T are of degree 1 in G: they are two eligible vertices, hence T can be double-rooted at these two vertices;
- the two leaves of T are connected by a non-tree edge, which is a contradiction since G has no Hamiltonian cycle;
- one leaf u has a non-tree edge to an internal vertex v of T. Add the edge uv into T to produce a C-tree. Since the root is of degree three, the C-tree is orientable. Obviously, $d_T(v) = 3$.

We may now assume T is not a Hamiltonian path. Next we show that the leaves of T are not joined by an edge in G. Indeed, suppose by a contradiction, u and v are two leaves of T and that $uv \in E(G)$. Now $T + uv$ has a unique cycle C and since T is not a Hamiltonian path, there must be a branch vertex $x \in C$ with $d_T(x) \geq 3$. Let $x' \in C$ be a neighbor of x on the $x - u$ path. Now the new tree $T + uv - xx'$ has at least one less leaf than T, a contradiction.

We continue our analysis of T depending on the number of branch vertices. First assume T has at least two branch vertices. Choose two branch vertices u, v that are connected in T by the path whose internal vertices are not branch vertices as the roots of the double-rooted tree T. Again, since $d_T(u) \geq 3$ and $d_T(v) \geq 3$, u, v are two eligible vertices.

Second assume T has exactly one branch vertex v. If there is a leaf u connected in G to an vertex w on the $u - v$ path, add this edge (u, w) to T to form a C-tree with root w. Obviously, $d_T(w) = 3$.

Finally, among all trees with a single branch vertex, choose T so that the lengths of branches from the branch vertex in lexicographic order is maximized, i.e., the longest branch is longest possible, with respect to this, the second longest branch is as long as possible, etc..

By excluding the previous cases, we may assume T has the following properties:

1. a leaf on branch i is not adjacent in G to a neighbor of the branch vertex, or to any vertex on branch $j > i$.
   Indeed, if a leaf is adjacent to a neighbor of the branch vertex, we can modify T to have one less leaf, a contradiction. If a leaf on branch i is adjacent to a vertex on branch $j > i$, we can modify T to a new tree having not more leaves than T, lengths of branches $< i$ same as in T, and the branch i of length greater than in T, a contradiction.
2. any leaf of T is of degree at least two in G.
   Indeed, a leaf of degree one in G is an eligible vertex and together with the branch vertex, can be the two roots in the double-rooted tree T.
3. the leaf on a branch i is not adjacent in G to an internal vertex of the branch i.
   We have proved above that a C-tree would have been constructed.

4. the leaf on the branch 1 is of degree two and its non-tree edge is adjacent to the branch vertex.

Let v be the branch vertex of T and let u be the leaf in branch 1, let w be the neighbor of the branch vertex v in branch 1. It follows from above restrictions on T, and as we consider simple graphs, $w \neq u$. Remove (v, w) from T and add (u, v). Note that the degree of the branch vertex v remains the same and vertex u is now of type (m) in Figure 4.5, i.e. an eligible vertex. Select v and u as the two roots to form a double-rooted tree T.

□

Constructing the tree T: The proof of Lemma 1 actually provides a way how to algorithmically find T. We can start with an arbitrary spanning tree T and then, using the methods described in the proof, decreasing the number of leaves and number of branch vertices as long as this is possible. It is easy to see that we either arrive to a Hamiltonian cycle, a C-tree, or a tree. In last two cases, there will be no edge joining two leaves of T. Finally, since every tree has at least two leaves, if T is a tree take the unique path between two leaves and consider two vertices that are either the leaves in G or of degree at least three in T on the path which are closest to each other. These two vertices will be the two roots of the double-rooted tree T.

Assigning the Orientation of T: Here we describe how to assign the orientation to edges in T.

1. T is a Hamiltonian cycle: It does not matter where the root is, choose the orientation of the cycle such that the up-tree direction is in the counter-clockwise order, i.e., the cycle will be oriented clockwise;

2. T is an orientable C-tree consisting of a cycle C and subtree π attaching to its root r: The orientation for edges in subtree π is from the leaves to r as the up-tree direction. The orientation for the edges in the cycle C is assigned so that the up-tree direction is the counter-clockwise direction on the cycle.

3. T is a double-rooted tree with two roots r_1 and r_2. Assign the orientation in both subtrees such that the up tree direction is from the leaves to the roots r_1 and r_2, respectively. The orientation on the root path is from r_2 to r_1.

We have explained how to construct the spanning subgraph T in G and assign the orientation of edges in T. In the next chapter, we will describe how to assign the port numbers to vertices of G. For the root vertices, we distinguish several cases how to do this. Setting port numbers for remaining vertices will be done by a general procedure, so called labelling scheme. The assignment of port numbers for given vertex v will depend on configurations $c(v)$.

Chapter 4

The Labeling Scheme

The labeling scheme will be divided into three parts: for vertices of large degree in G, i.e., $d \geq 5$, for vertices of small degree in G, i.e., $d \leq 4$, and for root vertices.

Also note that non-root vertices of the root path, if T is double-rooted tree, and the non-root vertices of the cycle, if T is a C-tree are labeled using this labelling scheme.

4.1 Labeling scheme for non-root vertices of degree $d \geq 5$

Recall that we consider the configuration and labeling sequence starting from the up-tree edge, in counter-clockwise order. We know from the Observation 1 that there is a labeling scheme that makes each vertex of degree at least 5 unambiguous. Here we give an explicit construction of labelling sequence which, in addition, allows the agent to determine the next edge without the need to look at once at the whole labeling sequence of the vertex.

Given a vertex v in G with $d = d_G(v) \geq 5$, let t be its degree in T. We will distinguish three cases based on the configuration: 10^{d-1}, $11^{t-1}0^{d-t}$ and anything else.

Case 1: Configuration 10^{d-1}. We set the labels $1, d, d-1, \ldots, 2$ starting from the up-tree edge. See Figure 4.1. The agent arriving to a vertex with this configuration will always go back to previous vertex with up-tree direction no matter what the current direction was.

Case 2: Configuration $11^{t-1}0^{d-t}$ with $1 < t < d$. We set the first $t+1$ labels starting from the up-tree edge to $1, 3, 4, \ldots, t+1, 2$ and the remaining labels to the remaining edges in decreasing order. See Figure 4.2. In case when $t = 2$, we also set the next label after 2 to 4, i.e., the labeling sequence starts with $1, 3, 2, 4$ and the rest in decreasing order.

Case 3: All other configurations. We label the tree edges in counter-clockwise order starting from the up-tree edge with labels $1, 2, \ldots, t$ and assign label $t+1$ to the first non-tree edge (if there is any). Remaining labels are assigned to remaining non-tree edges in decreasing order. See Figure 4.3.

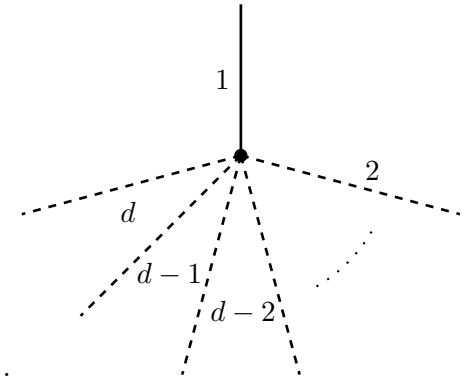

Figure 4.1: The unique up-tree edge is the only tree edge.

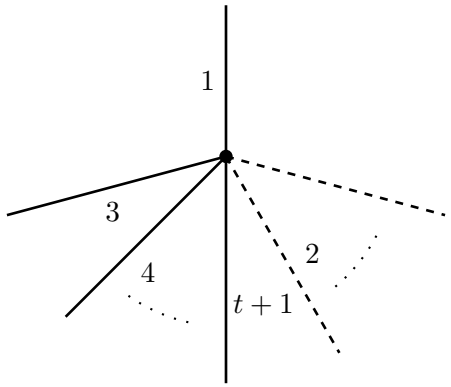

Figure 4.2: There is at least one tree edges and at least one non-tree edge in the down-tree edges and there is no non-tree edge proceeding a tree edge in the labeling sequence.

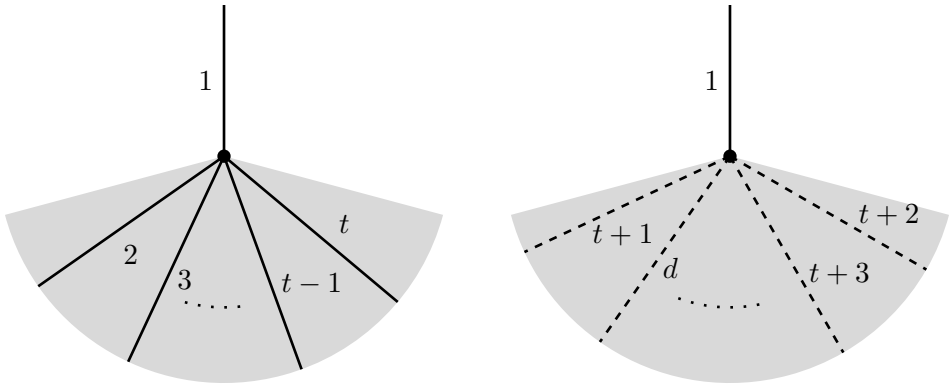

Figure 4.3: The left figure only shows the tree edges with their port numbers and hides the non-tree edges, while the right figure only shows the non-tree edges with their port numbers and hides the tree edges in the down-tree. The two figures are from a same configuration.

Lemma 2. *Consider any simple connected graph G, its spanning subgraph T and the labeling scheme described above. Then every vertex of degree at least 5 in G is unambiguous.*

Proof. Recall that v is unambiguous if for $l(v)$ there is a unique configuration $c(v)$ such that $ls(c(v)) = l(v)$. We prove by contrapositive. Let c_1 and c_2 be two different configurations and let $L_1 = ls(c_1)$ and $L_2 = ls(c_2)$ be their corresponding labeling sequences given by the labeling scheme described above. We will show that $L_1 \neq L_2$ by contradiction. Assume $L_1 = L_2 = (1, l_2, \ldots, l_d)$. It follows from the construction of the labeling scheme, that the first three labels after 1 uniquely determine the configuration type (Case 1, 2 or 3), cf. Figure 4.4. We may also assume c_1 and c_2 have the same length, otherwise $L_1 \neq L_2$. Hence, in Case 1, directly $c_1 = c_2$. In Case 2, the position of label 2 in the labeling sequence marks the first non-tree edge in both c_1 and c_2, hence, they must be equal. In Case 3, the number and position of tree edges are uniquely determined by the smallest label t for which $t + 1$ precedes t in the labeling sequence. Hence, again $c_1 = c_2$. This completes the proof. □

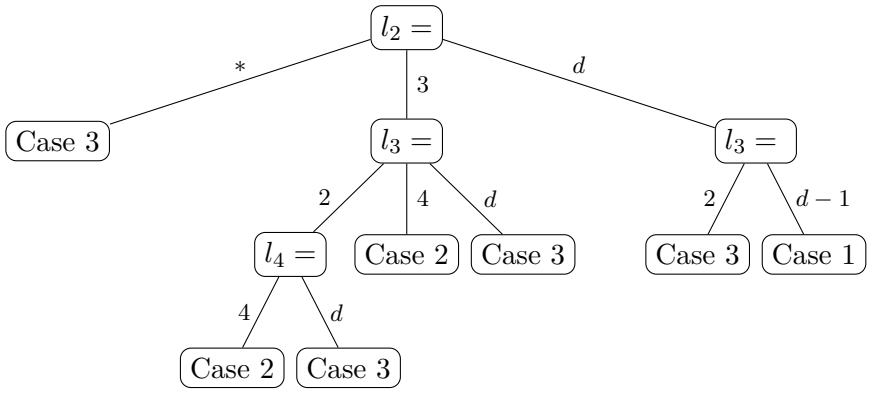

Figure 4.4: The decision tree shows that the first three labels l_2, l_3, l_4 in the labeling sequence excluding the first leading 1 determine the configuration type for vertices of degree at least 5.

4.2 Labeling scheme for non-root vertices of degree $d < 5$

The scheme is completely described in Figure 4.5, where all configurations are depicted and edges labeled with corresponding port numbers in labeling sequences.

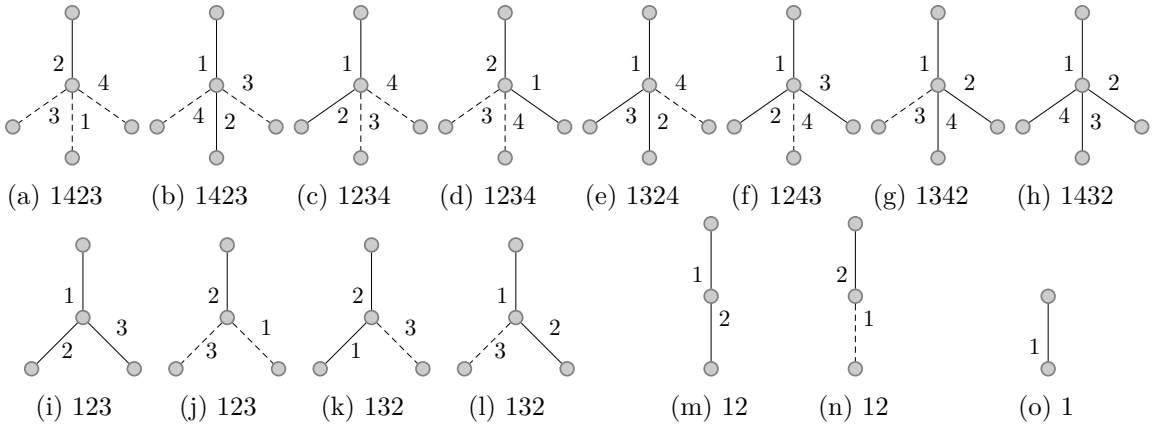

Figure 4.5: Labeling for vertices with $d < 5$. In each figure, the topmost edge is the up tree edge and the bottom edges are the down tree edges. The solid lines represent tree edges and the dashed lines depict non-tree edges. The sequence under each figure is the labeling sequence.

Unlike for $d \geq 5$, some configurations will be ambiguous, and as a consequence, the port numbers 1 and 2 will be assigned to the up-tree edges. We designed the labeling scheme with the following properties which can be readily observed:

1. (e), (f), (g), and (h) are unambiguous configurations. They can be identified by the corresponding labeling sequence and port number 1 is always assigned to the up-tree edge.

2. The configurations (a) and (b), (c) and (d), (i) and (j), (k) and (l), (m) and (n) come in pairs. They are called *dual* to each other. Each pair of dual configurations maps to one labeling sequence, hence the ambiguity is created on these configurations. Note that only port numbers 1 and 2 are assigned to the up-tree edge and port number 2 is always on the tree edge.

   Among these, there are two types of ambiguous configurations:

   - (c) and (d), (k) and (l) are called *transit configurations*, configurations for which both vertices in the pair have $t = 2$, i.e., degree of the vertex in T is two. The transition function of the agent will be designed so that, whenever the agent arrives to such vertex on one tree edge, it will directly leave on the other tree edge without changing traversing direction.

     The agent will not be able to notice any inconsistency when its assumption of traversing direction is wrong. A vertex with any of these transit configurations is called a *transit vertex*.

   - (a) and (b), (i) and (j), (m) and (n) are pairs of ambiguous *flip-flop configurations*. The transition function will be designed so that the agent will traverse these

configurations along tree edges in counterclockwise order and changing direction of traversal accordingly, arriving via up-tree edge of the configuration in down-tree direction, will continue down and coming back, eventually leaving again on the up-tree edge of the configuration in up-tree direction.

The ambiguity can result in the following behaviour. If an agent arrives to (a), (j), (n) on a tree edge with entering port number 2 and with wrong assumption on direction, i.e., in up-tree direction, it will interpret the vertex as having configuration (b), (i), (m), respectively. The agent will *flop* (will walk on a non-tree edge) according to one of these configurations. While, if an agent arrives to (b), (i), (m) on a tree edge with entering port number 2 and with wrong assumption about the direction, i.e., in down-tree direction, it will interpret it as (a), (j), (n), respectively. The agent will *flip* (bounce back to the last visited vertex). These ambiguities could create serious problems, however as we show, they cannot occur, except when the agent starts in (a), (j), and (n) and takes the non-tree edges with port 1. After this step the agent recognizes error immediately.

At vertices with configurations (b), (j), and (n), the transition function of the agent will be designed so that, the agent comes to these vertices no matter in down-tree or up-tree direction, it simply reverses its direction and returns back to the last visited vertex in up-tree direction.

3. (o) is a special unambiguous configuration. The transition function of the agent will be designed so that, the agent comes to (o) no matter in down-tree or up-tree direction, it simply returns back to the last visited vertex in up-tree direction.

4.3 Labeling scheme for root vertices

Above we have described the labeling scheme for general vertices in T that have an edge leading to the root, the up-tree edge. Recall, normally this edge will be labeled with 1, sometimes 2. In this subsection we handle remaining vertices. These are the root vertex in case T is a C-tree, and the main root in case T is a double rooted tree. Note that the non-main root is treated as a regular vertex (it has an edge leading to the main root). Also every vertex on the root-path has an edge leading to the main root and so these vertices behave as non-root vertices. The non-root vertices on the cycle C in case T is a C-tree have the unique incoming edge that is on C, and this edge will be the up-tree edge.

If T is a Hamiltonian cycle, as we mentioned above it does not matter which vertex is considered as a root, and in fact, in this special case every vertex will have an up-tree edge, the one on the cycle oriented into the vertex (incoming edge). Once we fix the up-tree edge, we determine the corresponding configuration of the vertex and assign port numbers on edges according to the configuration of the vertex which is now fixed.

If T is a C-tree, let r be its root. Then by Lemma 1, we have $d_T(r) = 3$. When r is an ambiguous vertex, it is easy to see that the configuration $c(r)$ must be type (i). Label the incoming edge into r that is on C by 2, the outgoing edge from r that is on C by 1, and the remaining tree edge by 3. If r is an unambiguous vertex, the labeling scheme is essentially the same way as in the previous case, but the port numbers on the two down-tree edges are referred to its specific port labelling rule.

Finally if T is a double-rooted tree, let r be the main root. Note that r is the only vertex we need to specify which edge is the up-tree edge. We fix the edge incident to r that is on the root-path as the up-tree edge. This fixes a configuration at r, $c(r)$ and we assign port numbers to edges accordingly. A very important note is that the edges on the root-path will be always traversed by the agent in up-tree direction.

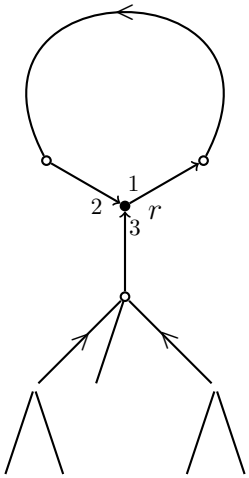

Figure 4.6: An example of a C-tree with $deg_g(r) = 3$.

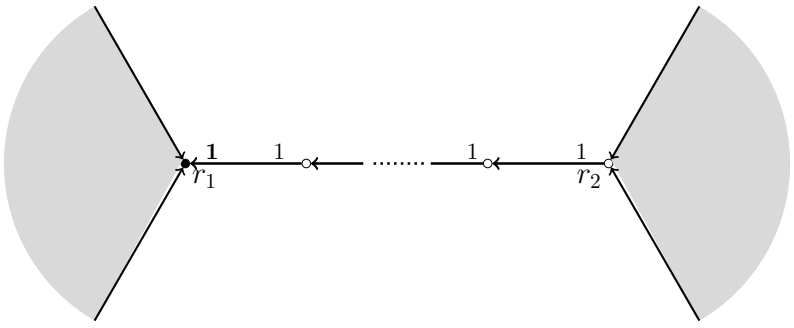

Figure 4.7: An example of double-rooted tree with r_1 as the main root, r_2 as the second root. The bold 1 depicts the one that is assigned after considering the edge as an up-tree edge when assigning ports to root vertices.

Chapter 5

The Transition Function

We first provide a high level overview of how the agent performs the traversal. The transition function will be designed so that the agent at every vertex will be traversing on tree edges only in counterclockwise order. In general, the agent enters a vertex on an edge and reads its port number. Then it reads the current state, the configuration of the vertex, and it determines the port number of the leaving edge. Usually this will be next tree edge counterclockwise around the vertex from the entering edge. The agent will distinguish tree edges from non-tree edges using the configuration and labeling. At unambiguous configurations/vertices it is always possible to distinguish tree edges from non-tree edges. At ambiguous configurations/vertices the agent can make a mistake. However, as we show, only a finite number of steps after it starts.

Note that scanning for the next tree-edge can be done in $O(d)$ steps of local processing. Indeed, the agent will scan counter-clockwise from the arrival port p and takes the first port corresponding to a tree-edge. In vertices of degree at most 4, this can be done by observing the whole labeling sequence and the edge will be determined by corresponding configuration (direction of traversal will resolve ambiguous configurations), while in vertices of degree at least 5, this will be done by first scanning counterclockwise for port with label 1, then observing the label on next three edges counterclockwise uniquely determines which of the tree cases of configuration the vertex is, see Figure 4.4. Finally scanning back to the incoming port the agent can determine the outgoing port. It is easy to see that the decision tree in Figure 4.4 can be turned into a subroutine of the agent and only three extra states will be needed to execute this subroutine.

Following is a more detailed description.

- While traversing, the agent maintains in its state its assumption about the traversal direction, it is either up-tree $\uparrow$, or down-tree $\downarrow$. The direction of traversal can only change at internal vertices of T of degree at least three.
- On start-up, the agent takes the edge with port 1, assuming the direction is up-tree, and it will start the preprocessing phase at the end of which the agent will correct

its assumptions if the assumption was wrong. This phase is not counted towards the period estimate and will take no more that n steps.

- The direction is changing as follows: when the agent arrives to a vertex v from its subtree, i.e., in direction up-tree and there is a tree edge counterclockwise around v witch leads to another subtree, the agent will leave v on that edge and changes the direction to down-tree. Eventually the agent arrives at a leaf at which it will change the direction to up-tree and bounces back. This way, eventually the agent returns to v and will be leaving it towards its parent (usually such edge is labelled by port number 1, sometimes 2 - see the configurations in previous chapter). At this point the agent will also keep the up-tree direction.

- The labeling scheme and the transition function are constructed so that the process described in item above will work as described, except when T is double rooted tree. In this case the agent will have to change direction also when passing between the two roots. The labeling scheme is designed so that the labels on edges at the two roots which are the first/last edge of the root-path are the up-tree labels in the configuration of the corresponding root. Hence the agent in up-tree direction leaving one of the roots will arrive to the other root (via the root path consisting of transit vertices) via an up-tree edge in an up-tree direction. The agent will recognize this and handles it via a Subroutine Error 2 - will change its direction to down-tree.

- Because both ports 1 and 2 are used as up-tree direction ports, as the agent starts on port 1 and assuming direction up-tree, the direction assumption may be wrong. This has two consequences.

  (a) either the agent is performing traversal with wrong direction (e.g., it is traversing T towards the leaves but its state is up-tree), or

  (b) because of wrong assumption of direction, at ambiguous vertices the agent may take a non-tree edge, thinking it is a tree edge.

  In both cases we show the agent will recover from these errors and once it does, it will never encounter these errors again. This is easy to see when the agent enters an unambiguous vertex. We will prove the same also in the case when the agent will not reach an unambiguous vertex before correction. The error in (a) is of the same flavour as Error 2, and so will be handled by the Error 2 subroutine. The error in (b) will be handled by the Error 1 subroutine.

Subroutine Error 1: This subroutine is executed when the agent determines that it passed on a non-tree edge. The agent determines this from the configuration at the arriving vertex. It will arrive to the vertex on an edge whose port will correspond to a non-tree

edge. For example, if the agent arrives to a vertex with labeling sequence 1234 via port 3, it recognizes that it is in a vertex with configuration (c) or (d), and since in both these configurations port 3 is a non-tree edge, the agent detects inconsistency. The agent will backtrack to previous vertex and takes port 2, with up-tree direction.

Subroutine Error 2: This subroutine is executed when the agent detects inconsistent direction, i.e., it will arrive to a vertex via edge whose port number corresponds to an edge in the configuration which cannot be entered via agent's direction. For example, if the agent arrives in down-tree direction to a vertex with configuration (e) via an edge with port 3. The agent corrects its assumption about the direction and calculates the port number of outgoing edge based on the configuration.

Exception 1: Exception when direction is overwritten. This subroutine overrides the standard handling in the cases when the agent it traversing in the up-tree direction and will arrive to an eligible vertex on the up-tree edge of the configuration, except when the configuration is (o). Note that this only happens on the root-path when the agent traverses between two roots. In this case the agent switched its direction to down-tree direction and continues execution of the transition function.

Exception 2: Exception when errors are not generated. For configurations that correspond to vertices that are on the cycle C in case T is a C-tree or on the root-path in case T is a double-rooted tree, we need special handling so that the agent can be traversing these in both directions without generating and handling errors. The idea is that these vertices will be always traversed in up-tree direction (direction will be changing only at the root vertices of C, resp. the root-path). For example, if the labeling sequence is 1423 and the configuration is (b). It follows that a vertex with such configuration could be on C or on the root-path. If it is on a root-path, the edge with label 1 will be the up-tree edge and when the agent enters the vertex via the edge with port 2, everything will be fine. However, eventually the agent will be passing through the vertex, again in up-tree direction, but now entering via edge with port 1. As we see, if the agent assumes the configuration is (a) it will handle this as Error 1, and if it assumes the configuration is (b), it would handle this as Error 2. However, we override this behaviour and treat this as transit vertex. Inspecting all ambiguous configurations in which the vertex has degree two (can be transit vertex on C or the root-path), we conclude that for labelling sequences 1423, and 12, if the agent is entering in up-tree direction on the edge with port number 1, we treat this particular situation as the vertex is a transit vertex - pass through and keep up-tree direction. The same adjustment is done in configurations for $d \geq 5$ when the corresponding vertex has degree two in T. Note that in the other direction (when arriving to these vertices not on the port

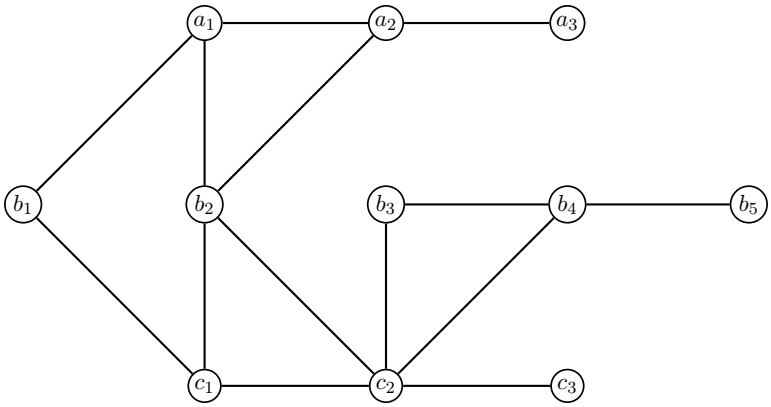

Figure 5.1: A example of given graph G.

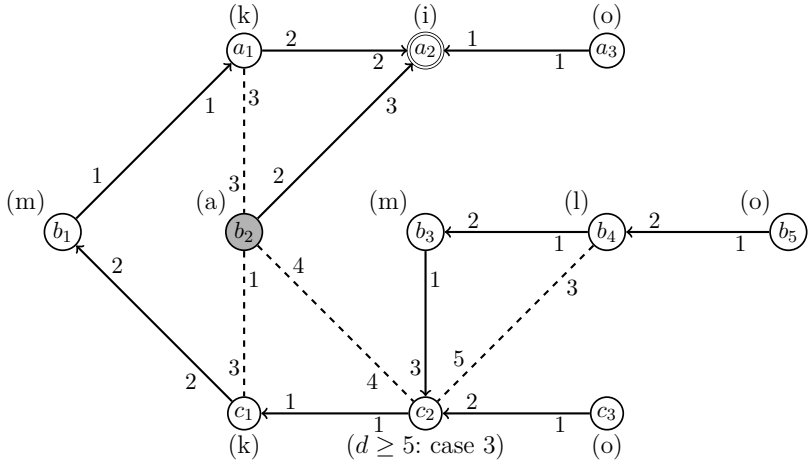

Figure 5.2: Constructed T as a double-rooted tree with a_2 as the main root. An agent starts its traversal at vertex b_2.

1) the pass is handled by regular rules. As we mentioned earlier, the transit configurations (c), (d), (k), (l) will be handled by transition function the same way.

An example: We demonstrate the computation of the agent in the following example. Figure 5.1 depicts an input graph. The rotation system is implied from its embedding into the plane. In Figure 5.2 we have chosen a spanning tree T according to Lemma 1.

In the Table 5.1, we list all steps the agent will take starting from the vertex b_2, making one round trip along T, and coming back to b_2.

Formal description of the transition function: Given current vertex v, the agent will decide about the outgoing port number j and the next state s' based on its current state s, the labeling sequence at current vertex $L = l(v)$, the degree of the current vertex $d = d_G(v)$ (d is the length of L) and the incoming port number i. Formally, the transition function δ maps 4-tuples (s, L, d, i) to pairs (s', j).

Step	Vertex	Input	Output	Description
1	b_2	$(q_0, l(b_2), d_{b_2}, \epsilon)$	$(\uparrow, 1)$	Started at b_2, by default take 1 and assume going up-tree.
2	c_1	$(\uparrow, 132, 3, 3)$	$(\uparrow, 3)$	In type (k), 3 is on a non-tree edge. The agent found Error 1: it walked on a non-tree edge. It backtracks to previous vertex b_2.
3	b_2	$(\uparrow, 1423, 4, 1)$	$(\uparrow, 2)$	The agent took port number 2 going up-tree. From step 3 to step 23, the agent would complete a Euler tour in T.
4	a_2	$(\uparrow, 123, 3, 3)$	$(\uparrow, 1)$	In type (i), port 3 is on the last down-tree edge. The agent left a_2 to up-tree by port 1.
5	a_3	$(\uparrow, 1, 1, 1)$	$(\uparrow, 1)$	Exception 1 found, agent returned back to root path.
6	a_2	$(\uparrow, 123, 3, 1)$	$(\downarrow, 2)$	Exception 1 found, set direction to down-tree, recompute the output as $(\downarrow, 2)$.
7	a_1	$(\downarrow, 132, 3, 2)$	$(\downarrow, 1)$	Crossed a transit vertex.
8	b_1	$(\downarrow, 12, 2, 1)$	$(\downarrow, 2)$	Crossed a transit vertex.
9	c_1	$(\downarrow, 132, 3, 2)$	$(\downarrow, 1)$	Crossed a transit vertex.
10	c_2	$(\downarrow, 12534, 5, 1)$	$(\downarrow, 2)$	Went to the first down-tree branch.
11	c_3	$(\downarrow, 1, 1, 1)$	$(\uparrow, 1)$	The agent reached a leaf, reversed back to up-tree.
12	c_2	$(\uparrow, 12534, 5, 2)$	$(\downarrow, 3)$	After returned back from the first brach, the agent went to the second branch at c_2.
13	b_3	$(\downarrow, 12, 2, 1)$	$(\downarrow, 2)$	Crossed a transit vertex.
14	b_4	$(\downarrow, 132, 3, 1)$	$(\downarrow, 2)$	Crossed a transit vertex.
15	b_5	$(\downarrow, 1, 1, 1)$	$(\uparrow, 1)$	The agent reached a leaf, reversed back to up-tree.
16	b_4	$(\uparrow, 132, 3, 2)$	$(\uparrow, 1)$	Crossed a transit vertex.
17	b_3	$(\uparrow, 12, 2, 2)$	$(\uparrow, 1)$	Crossed a transit vertex.
18	c_2	$(\uparrow, 12534, 5, 3)$	$(\uparrow, 1)$	The agent had traversed all the down tree vertices at c_2, then it left c_2 back to the root.
19	c_1	$(\uparrow, 132, 3, 1)$	$(\uparrow, 2)$	Crossed a transit vertex.
20	b_1	$(\uparrow, 12, 2, 2)$	$(\uparrow, 1)$	Crossed a transit vertex.
21	a_1	$(\uparrow, 132, 3, 1)$	$(\uparrow, 2)$	Crossed a transit vertex.
22	a_2	$(\uparrow, 123, 3, 2)$	$(\downarrow, 3)$	The agent turned down-tree to the second branch.
23	b_2	$(\downarrow, 1423, 4, 2)$	$(\uparrow, 2)$	A full Euler tour of T completed in 20 steps. In this graph, $n = 11$, $2n - 2 = 20$.

Table 5.1: Step by step simulation of the graph traversal process. The agent starts at the vertex b_2, and ϵ represents no input label, and q_0 is the initial state.

Let $S = \{\uparrow, \downarrow\}$ be the two states of the agent, representing traversal direction up-tree or down-tree. Let $L = (1, l_2, l_3, l_4, \ldots, l_d)$ be the labelling sequence at the current vertex. The labels in L are representing the port numbers assigned at every incident edges to v starting from the up-tree edge in a counter-clockwise order. The *cyclic order* in L means the label after the last label is the first label in L. The order is repeating cyclically.

As we mentioned above, we will not provide the code for the error processing phases. These are simple and are left for the reader. For clarity, we split the formal description of the remaining part of the transition function into procedures, depending on the degree d of the current vertex v. The first case covers cases when $d \geq 5$. We refer the reader to the Subsection 4.1 for corresponding cases to consider. When an error is returned, the agent will perform the corresponding error handling subroutine as described above. The corresponding code is in Algorithm 1.

The remaining cases cover scenarios when $d < 5$. We refer the reader to the Subsection 4.2 for corresponding cases to consider. As before, when an error is returned, the agent will perform subroutines as described above. We provide the code in Algorithm 2, Algorithm 3, and Algorithm 4.

Algorithm 1: Transition function of main subroutine when $d \geq 5$

Input : (s, L, d, i)
Output: (s', j)

$l_2, l_3, l_4 \leftarrow L$
Case Number $\leftarrow l_2, l_3, l_4$
$t \leftarrow$ Case Number,L, d
Case 1:
if $i = 1$ **then**
    if $s =\downarrow$ **then**
        return $(\uparrow, 1)$
    else
        Exception 1: set direction to $\downarrow$, process configuration, and hence output $(\uparrow, 1)$
else
    execute Error 1 subroutine (output $(\uparrow, i)$ will be returned by Error 1)
Case 2:
if $i = 1$ *or* $3 \leq i \leq t + 1$ **then**
    if $t = 2$ **then**
        if $i = 1$ **then**
            if $s =\downarrow$ **then return** $(\downarrow, 3)$;
            else Exception 2: output $(\uparrow, 3)$;
        if $i = 3$ **then**
            if $s =\uparrow$ **then return** $(\uparrow, 1)$;
            else
                execute Error 2 subroutine (output $(\uparrow, 1)$ will be returned by Error 2)
    else
        if $i = 1$ **then**
            if $s =\downarrow$ **then return** $(\downarrow, 3)$;
            else Exception 1: set direction to $\downarrow$, process configuration, and hence output $(\downarrow, 3)$;
        else if $3 \leq i \leq t$ **then**
            if $s =\uparrow$ **then return** $(\downarrow, i + 1)$;
            else execute Error 2 subroutine (output $(\downarrow, i + 1)$ will be returned by Error 2);
        else if $i = t + 1$ **then**
            if $s =\uparrow$ **then return** $(\uparrow, 1)$;
            else execute Error 2 subroutine (output $(\uparrow, 1)$ will be returned by Error 2);
else
    execute Error 1 subroutine (output $(\uparrow, i)$ will be returned by Error 1)

```
Case 3:
if 1 ≤ i ≤ t then
    if t = 2 then
        if i = 1 then
            if s =↓ then return (↓, 2);
            else Exception 2: output (↑, 2) ;
        if i = 2 then
            if s =↑ then return (↑, 1);
            else execute Error 2 subroutine (output (↑, 1) will be returned by Error 2)
            ;
    else
        if i = 1 then
            if s =↓ then return (↓, 2);
            else Exception 1: set direction to ↓, process configuration, and hence
            output (↓, 2);
        else if 2 ≤ i ≤ t − 1 then
            if s =↑ then return (↓, i + 1);
            else execute Error 2 subroutine (output (↓, i + 1) will be returned by
            Error 2);
        else if i = t then
            if s =↑ then return (↑, 1);
            else execute Error 2 subroutine (output (↑, 1) will be returned by Error 2);
else
    execute Error 1 subroutine (output (↑, i) will be returned by Error 1)
```

Algorithm 2: Transition function of main subroutine when $d \leq 3$

Input : (s, L, d, i)
Output: (s', j)

Given $d \leq 3$**:**
if $L = 123$ **then**
  if $i = 1, s = \downarrow$ **then**
    return $(\downarrow, 2)$
  else if $i = 1, s = \uparrow$ **then**
    Exception 1: set direction to $\downarrow$, process configuration, and hence output $(\downarrow, 2)$)
  else if $i = 2, s = \uparrow$ **then**
    return $(\downarrow, 3)$
  else if $i = 2, s = \downarrow$ **then**
    return $(\uparrow, 2)$
  else if $i = 3, s = \uparrow$ **then**
    return $(\uparrow, 1)$
  else if $i = 3, s = \downarrow$ **then**
    execute Error 2 subroutine (output $(\uparrow, 1)$ will be returned by Error 2)
if $L = 132$ **then**
  if $i = 1$ **then**
    return $(s, 2)$
  else if $i = 2$ **then**
    return $(s, 1)$
  else
    execute Error 1 subroutine (output $(\uparrow, i)$ will be returned by Error 1)
if $L = 12$ **then**
  if $i = 1, s = \downarrow$ **then**
    return $(\downarrow, 2)$
  else if $i = 1, s = \uparrow$ **then**
    Exception 2: output $(\uparrow, 2)$
  else if $i = 2, s = \uparrow$ **then**
    return $(\uparrow, 1)$
  else if $i = 2, s = \downarrow$ **then**
    return $(\uparrow, 2)$
if $L = 1$ **then**
  if $s = \downarrow$ **then**
    return $(\uparrow, 1)$
  else if $s = \uparrow$ **then**
    Exception 1: set direction to $\downarrow$, process configuration, and hence output $(\uparrow, 1)$)

Algorithm 3: Transition function of main subroutine when $d = 4$ and $t = 1, 2$

Input : (s, L, d, i)
Output: (s', j)

$t \leftarrow L, d$
Given $d = 4$ **and** $t = 1, 2$**:**
if $L = 1423$ **then**
    if $i = 1, s =\downarrow$ **then**
        return $(\downarrow, 2)$
    else if $i = 1, s =\uparrow$ **then**
        Exception 2: output $(\uparrow, 2)$
    else if $i = 2, s =\downarrow$ **then**
        return $(\uparrow, 2)$
    else if $i = 2, s =\uparrow$ **then**
        return $(\uparrow, 1)$
    else
        execute Error 1 subroutine (output $(\uparrow, i)$ will be returned by Error 1)
if $L = 1234$ **then**
    if $i = 1$ **then**
        return $(s, 2)$
    else if $i = 2$ **then**
        return $(s, 1)$
    else
        execute Error 1 subroutine (output $(\uparrow, i)$ will be returned by Error 1)

Algorithm 4: Transition function of main subroutine when $d = 4$ and $t = 3, 4$

Input : (s, L, d, i)
Output: (s', j)

$t \leftarrow L, d$
Given $d = 4$ **and** $t = 3, 4$**:**
if $L = 1324$ **then**
    if $i = 1, s =\downarrow$ **then**
        return $(\downarrow, 3)$
    else if $i = 1, s =\uparrow$ **then**
        Exception 1: set direction to $\downarrow$, process configuration, and hence output $(\downarrow, 3)$
    else if $i = 2, s =\uparrow$ **then**
        return $(\uparrow, 1)$
    else if $i = 2, s =\downarrow$ **then**
        execute Error 2 subroutine (output $(\uparrow, 1)$ will be returned by Error 2)
    else if $i = 3, s =\uparrow$ **then**
        return $(\downarrow, 2)$
    else if $i = 3, s =\downarrow$ **then**
        execute Error 2 subroutine (output $(\downarrow, 2)$ will be returned by Error 2)
    else
        execute Error 1 subroutine (output $(\uparrow, i)$ will be returned by Error 1)
if $L = 1243$ **then**
    if $i = 1, s =\downarrow$ **then**
        return $(\downarrow, 2)$
    else if $i = 1, s =\uparrow$ **then**
        Exception 1: set direction to $\downarrow$, process configuration, and hence output $(\downarrow, 2)$
    else if $i = 2, s =\uparrow$ **then**
        return $(\downarrow, 3)$
    else if $i = 2, s =\downarrow$ **then**
        execute Error 2 subroutine (output $(\downarrow, 3)$ will be returned by Error 2)
    else if $i = 3, s =\uparrow$ **then**
        return $(\uparrow, 1)$
    else if $i = 3, s =\downarrow$ **then**
        execute Error 2 subroutine (output $(\uparrow, 1)$ will be returned by Error 2)
    else
        execute Error 1 subroutine (output $(\uparrow, i)$ will be returned by Error 1)

```
if L = 1342 then
    if i = 1, s =↓ then
        return (↓, 4)
    else if i = 1, s =↑ then
        Exception 1: set direction to ↓, process configuration, and hence output (↓, 4)
    else if i = 2, s =↑ then
        return (↑, 1)
    else if i = 2, s =↓ then
        execute Error 2 subroutine (output (↑, 1) will be returned by Error 2)
    else if i = 4, s =↑ then
        return (↓, 2)
    else if i = 4, s =↓ then
        execute Error 2 subroutine (output (↓, 2) will be returned by Error 2)
    else
        execute Error 1 subroutine (output (↑, i) will be returned by Error 1)
if L = 1432 then
    if i = 1, s =↓ then
        return (↓, 4)
    else if i = 1, s =↑ then
        Exception 1: set direction to ↓, process configuration, and hence output (↓, 4)
    else if i = 2, s =↑ then
        return (↑, 1)
    else if i = 2, s =↓ then
        execute Error 2 subroutine (output (↑, 1) will be returned by Error 2)
    else if i = 3, s =↑ then
        return (↓, 2)
    else if i = 3, s =↓ then
        execute Error 2 subroutine (output (↓, 2) will be returned by Error 2)
    else if i = 4, s =↑ then
        return (↓, 3)
    else if i = 4, s =↓ then
        execute Error 2 subroutine (output (↓, 3) will be returned by Error 2)
```

Chapter 6

Correctness

In this chapter, we prove the correctness of our algorithm. First, by the construction of T and the labelling scheme, the agent will recognize an error at any unambiguous vertex, and will correct the error and direction. So after recognizing an error, the agent will be in perfect knowledge of current vertex (recognizes which are tree and non-tree edges, and hence also the up-tree edge). Whenever the agent is in such a situation, we say it has the *correct assumption*. We start with a useful observation which allows us to concentrate only on cases when the agent does not have the correct assumption, and some special considerations will have to be given to root vertices.

Observation 2. *If an agent has a correct assumption at a current non-root vertex, then in subsequent step it will keep correct assumption, i.e., in next vertex will recognize the up-tree edge, edges and non-edges even if the vertex is ambiguous.*

Proof. It is easy to see, by simple but tedious inspection of the transition function and configurations, that if the agent arrives to a non-root vertex (unambiguous or ambiguous) on a tree edge and with correct assumption, its transition function will output the outgoing tree edge and correct direction following labelling on the corresponding configuration of the vertex. □

Note that the Observation 2 is valid also for the transit vertices and for the vertices with exceptional handling - see paragraph "Exceptions when errors are not generated".

The next lemma will extend the previous observation and justifies the correctness of our algorithm once the agent has the correct assumption. The phase before this happens is called the initialization and moves by the agent do not count towards the period.

Lemma 3. *If the agent has a correct assumption at a current vertex, then it will perform a periodic traversal of G by following a trail along edges of T with the period at most $2n - 2$.*

Proof. The proof is essentially the same as proof of Observation 2, however we need to be more careful that all the local considerations "fit together". For this, we examine all cases of T in Lemma 1. We will assume the current direction is up-tree. The other case is similar.

1. If T is a Hamiltonian cycle, by Observation 2, the agent will follow the up-tree direction on transit vertices or vertices with configurations (b), (m), and similar special configurations when $d \geq 5$, only and will do a periodic traversal of the cycle with the period n.

2. If T is a spanning orientable C-tree, with the subtree π rooted at a vertex r. It is easy to see that r is either an unambiguous vertex, or an ambiguous vertex, in which case the only possibility for the configuration of r is of type (i). In both cases we have the following. If the current vertex is in π, by Observation 2, it follows that the agent will traverse only on edges of π and eventually will arrive at r. Now either the agent will change direction to down-tree and will continue back to π, or because C is orientable, eventually the agent will continue to up-tree edge (which is edge of the cycle C) with port number 1. Then again by Observation 2, the agent will traverse the whole cycle and returns back to r. Because the agent is still in up-tree direction and C is orientable, after arriving to r, it will switch the direction to down-tree, and will take next tree edge in counter-clockwise order about r leading back to the subtree π. Now the agent has correct assumption and is on an internal vertex of π. The argument that shows that the agent traverses only tree edges of π and eventually returns back to r is similar to the one above. Since the C-tree is spanning, it follows that the period of the traversal is $|C| + 2|\pi| - 2$ which is at most $2n - 2$.

   Similar analysis can be performed on the agent's moves when the current vertex is on C.

3. If T is a double-rooted tree with roots r_1 and r_2, let r_2 being the main root. Let π_1 and π_2 be the corresponding subtrees rooted at r_1 and r_2, respectively. By Lemma 1, both roots are eligible vertices. If the current vertex is in π_1, by Observation 2, it follows that the agent will traverse only on edges of π_1 and eventually will arrive at r_1. Now either the agent will change direction to down-tree and will continue back to π_1, or will enter the root-path and keeps direction up-tree. Then again by Observation 2, the agent will traverse the whole root-path and enters r_2. Because r_2 is an eligivle vertex and main root, the port on the incoming edge to r_2 is the up-tree port of the configuration, and the agent detects Error 2, it will flip direction to down-tree, and will traverse the subtree π_2 eventually coming back to r_2 and continuing on the root-path in up-tree direction. Since r_1 is eligible vertex and the incoming edge to r_1 has port corresponding to the up-tree edge of the configuration, the agent detects Error 2, it will flip direction to down-tree, and will traverse π_1, eventually reaching the original vertex. It follows that the agent will traverse G with period $2n - 2$.

   The cases when the current vertex is in π_2 or on the root-path are similar.

□

The next lemma guarantees that the agent will eventually have correct assumption.

Lemma 4. *If the agent starts with wrong assumption about the direction, i.e., at the initial vertex the port number 1 is either on a non-tree edge, or on a down-tree edge (of the configuration), it will correct its assumption in at most n steps.*

Proof. It follows from Lemma 3 that if the agent starts in an unambiguous vertex or its assumption about direction is correct (port 1 is on an up-tree edge, of the configuration), it will perform a periodic traversal with the period at most $2n - 2$. Thus, we may assume the agent starts in an ambiguous vertex with wrong assumption about the direction. As the agent starts by taking the edge with port number 1 and assuming the direction is up-tree, it will have wrong assumption only in configurations (a), (d), (j), (k), and (n). The first time the agent arrives to a vertex with an unambiguous configuration, i.e., a configuration for $d \geq 5$, or (e)-(h), or (o), the agent will correct its assumption. Hence the agent will start in one of the configurations (a), (d), (j), (k), and (n), and will be visiting only vertices with ambiguous configurations. We analyze all cases.

1. The agent starts at a vertex u with configuration (a). The port 1 is on the non-tree edge in (a). Since (a) represents a leaf of T and since there is no non-tree edge joining two leaves in T, the other end of the non-tree edge, must be of degree at least two in T, and its configuration must be one of the ambiguous configurations, say (x). Also in order for the agent not to recognize an error of walking on a non-tree edge, the port on the non-tree edge in (x) must be a tree edge in the corresponding dual configuration to (x), say (y). Moreover (y) must also represent a vertex of degree at least two in T. This is because (y) represents a leaf and the agent would continue back to previous vertex in up-tree direction which would now be correct direction. We inspect all possible pairs (x)-(y) of ambiguous configurations.

   The pair (a)-(b): The port 1 is on a non-tree edge in (a) and on tree edge in (b), however the edge is a down-tree edge. The port 2 is on tree edge in both (a) and (b). The ports 2 and 4 are on non-tree edges in both (a) and (b). In all cases, we either reach a contradiction or the agent recognizes an error.

   The pair (c)-(d): The only ports on non-tree edges are ports 3 and 4 and they are both on non-tree edges in both (c) and (d), hence the agent will realize an Error 1.

   The pair (i)-(j): The configuration (j) is the only one (of the two) having a non-tree edge. However, it represents a leaf, a contradiction with Lemma 1.

   The pair (k)-(l): The only port on non-tree edges in these two configurations is port 3. The agent will realize an Error 1.

   The pair (m)-(n): The only configuration (of the two) with a non-edge is (n). However, (n) represents a leaf, a contradiction with Lemma 1.

   Note that the agent will will realize an error after first step in all cases.

2. The agent starts at a vertex u with configuration (d). The port 1 is on a tree-edge in (d), so let the vertex v be the other end of the edge. Also assume the configuration of v is (x). Since the agent has wrong assumption about the direction, in T it will reach v on the up-tree edge of (x). Hence the port must be either 1 or 2. In order for the agent not to realize Error 1 or Error 2, only candidates for (x) are (b), (c), (d), (i), (k), (l), and (m). As long as the agent is traversing in this way only via vertices with configurations (c), (d), (k), and (l), it will stay on tree-edges and it will be approaching a leaf in T. Since G is finite and visited vertices cannot repeat, after at most $n-2$ steps the agent must reach a leaf of T. By our previous assumptions, the leaf must have labelling sequence 1423, 123, or 12.

   In the case of 1423, the agent will wrongly assume the vertex has configuration (b) and it has entered it on port 2, so will continue on port 1. In reality the vertex has configuration (a) and the agent will leave it on a non-tree edge with port 1.

   In the case of 123, the agent will wrongly assume the vertex has configuration (i) and it has entered it on port 2, so will continue on port 1. In reality the vertex has configuration (j) and the agent will leave it on a non-tree edge with port 1.

   In the case of 123, the agent will wrongly assume the vertex has configuration (m) and it has entered it on port 2, so will continue on port 1. In reality the vertex has configuration (n) and the agent will leave it on a non-tree edge with port 1.

   Since configurations (a), (j), and (n) represent leaves of T, one can complete the proof in the same way as in 1.

3. The agent starts at a vertex u with configuration (j). This case is the same as 1.

4. The agent starts at a vertex u with configuration (k). This case is the same as 2.

5. The agent starts at a vertex u with configuration (n). This case is the same as 1.

We have covered all possible cases and concluded that they either lead to a contradiction or the agent will recognize an error in at most n steps and correct its direction. It is clear that in this phase the agent traverses only on tree-edges, except possibly its last step when it will take a non-tree edge and immediately realizes this in next vertex. This justifies that the handling of Error 1 (backtracking one step and changing direction) is correct. □

Combining results above yields the main theorem Theorem 8.

Chapter 7

Concluding Remark

In this thesis we proved that the periodic traversal of graphs by a finite state automaton with period $2n-2$ is possible if we assume that the graph is given with a rotation system that specifies a circular ordering of edges at each vertex. Hence the agent at any vertex v in the graph G can read the permutation of port numbers on the incident edges to v. Under this assumption, an agent with two states can perform an Euler tour on the spanning subgraph T of G periodically once it has correct assumption of traveling direction; if the agent starts its traversal with wrong assumption, it can correct it within n steps.

We constructed the spanning graph T in a special way that there is no edge of G joining two leaves of T and T falls into three cases, each with strong structural properties, refer to Lemma 1. These properties overcome the shortage of labeling sequences compared to number of configurations for vertices with $d < 5$, and help to resolve the ambiguity. The choice of using this spanning tree was crucial in this research.

We gave labeling scheme, for vertices of $d \geq 5$ and for vertices of $d < 5$. The correctness of our approach has been proved in Chapter 6.

7.1 Open Questions

The labeling sequence at each vertex when compared to this circular ordering encodes $\Theta(d \log d)$ bits of information. A natural question is whether instead of a rotation system having (precomputed) bits of information available at each vertex could help the agent with the traversal. In particular, what is the least amount of bits of information at each vertex (or overall) that allows for periodic traversal with period $2n-2$? Could we achieve such a traversal with one bit of information per vertex?

Another way to look at the problem is to evaluate competitive ratio w.r.t. to best solution of TSP. There are various approximation results, from classical ones [9] to fairly recent improvements [24], including work on sub-cubic graphs [7] which as we have seen are

the difficult case. However, these results use specific walks which might not be possible to encode with the very limited information available.

Bibliography

[1] Susanne Albers and Monika R. Henzinger. Exploring unknown environments. *SIAM J. Comput.*, 29(4):1164–1188, February 2000.

[2] 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.

[3] Michael A. Bender, Antonio Fernández, Dana Ron, Amit Sahai, and Salil Vadhan. The power of a pebble: Exploring and mapping directed graphs. *Information and Computation*, 176(1):1–21, 2002.

[4] Margrit Betke, Ronald L. Rivest, and Mona Singh. Piecemeal learning of an unknown environment. *Machine Learning*, 18(2):231–254, Feb 1995.

[5] Avrim Blum, Prabhakar Raghavan, and Baruch Schieber. Navigating in unfamiliar geometric terrain. *SIAM Journal on Computing*, 26(1):110–137, 1997.

[6] Lothar Budach. Automata and labyrinths. *Mathematische Nachrichten*, 86(1):195–282, 1978.

[7] Barbora Candráková and Robert Lukotka. Cubic TSP - a 1.3-approximation. *CoRR*, abs/1506.06369, 2015.

[8] Ashok K. Chandra, Prabhakar Raghavan, Walter L. Ruzzo, Roman Smolensky, and Prasoon Tiwari. The electrical resistance of a graph captures its commute and cover times. *Computational Complexity*, 6(4):312–340, Dec 1996.

[9] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, 1976.

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

[11] Jurek Czyzowicz, Stefan Dobrev, Leszek Gąsieniec, David Ilcinkas, Jesper Jansson, Ralf Klasing, Ioannis Lignos, Russell Martin, Kunihiko Sadakane, and Wing-Kin Sung. More efficient periodic traversal in anonymous undirected graphs. *Theoretical Computer Science*, 444:60–76, 2012.

[12] Xiaotie Deng, Tiko Kameda, and Christos Papadimitriou. How to learn an unknown environment. i: The rectilinear case. *J. ACM*, 45(2):215–245, 1998.

[13] Xiaotie Deng and Christos H. Papadimitriou. Exploring an unknown graph. *Journal of Graph Theory*, 32(3), 1999.

[14] Stefan Dobrev, Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Finding short right-hand-on-the-wall walks in graphs. In *Structural Information and Communication Complexity: 12 International Colloquium, SIROCCO*, pages 127–139, Berlin, Heidelberg, 2005. Springer.

[15] Christian A. Duncan, Stephen G. Kobourov, and V. S. Anil Kumar. Optimal constrained graph exploration. *ACM Trans. Algorithms*, 2(3):380–402, July 2006.

[16] J.R. Edmonds. *A Combinatorial Representation for Oriented Polyhedral Surfaces.* 1960.

[17] Uriel Feige. A tight upper bound on the cover time for random walks on graphs. *Random Structures and Algorithms*, 6(1):51–54, 1995.

[18] Rudolf Fleischer and Gerhard Trippen. *Exploring an Unknown Graph Efficiently.* Springer, Berlin, Heidelberg, 2005.

[19] Pierre Fraigniaud and David Ilcinkas. *Digraphs Exploration with Little Memory.* Springer, Berlin, Heidelberg, 2004.

[20] Pierre Fraigniaud, David Ilcinkas, Guy Peer, Andrzej Pelc, and David Peleg. Graph exploration by a finite automaton. *Theoretical Computer Science*, 345(2):331–344, 2005.

[21] Leszek Gąsieniec, Ralf Klasing, Russell Martin, Alfredo Navarra, and Xiaohui Zhang. Fast periodic graph exploration with constant memory. In *Structural Information and Communication Complexity: 14th International Colloquium, SIROCCO, Proceedings*, pages 26–40, Berlin, Heidelberg, 2007. Springer.

[22] David Ilcinkas. Setting port numbers for fast graph exploration. *Theoretical Computer Science*, 401(1):236–242, 2008.

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

[24] Tobias Moemke and Ola Svensson. Approximating graphic tsp by matchings. In *Proceedings of the 2011 IEEE 52Nd Annual Symposium on Foundations of Computer Science*, FOCS '11, pages 560–569, Washington, DC, USA, 2011. IEEE Computer Society.

[25] PetriAor Panaite and Andrzej Pelc. Exploring unknown undirected graphs. *Journal of Algorithms*, 33(2):281 – 295, 1999.

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